Lecture Notes on Deep Learning for Computer Vision

Stephen Gould*
stephen.gould@anu.edu.au

November 21, 2024

Abstract

These lecture notes are for a one-semester (12-week) course on deep learning for computer vision. The course covers the theory and practice of deep learning with a focus on applications in computer vision. Students will learn the foundational mathematics behind deep learning and explore topics such as multi-layer perceptrons (MLPs), back-propagation and automatic differentiation, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers. These techniques play a crucial role in modern artificial intelligence (AI) systems, including image and video understanding, natural language processing, generative AI, and robotics. The course includes various practical assessments to enhance student's understanding and intuition of deep learning and its application in computer vision. Students are expected to have strong programming skills and previous exposure to linear algebra, differential calculus, and probability theory. Assessment details do not form part of these notes.

Appetizer

Consider the two-dimensional shape depicted in Figure 1. Nobody would have trouble recognising the shape as a triangle despite the lines not being completely straight and them not meeting precisely at each corner. And when asked why the shape is a triangle, most people would answer that it's because the shape has three sides. Deep learning, however, takes a different perspective. Deep learning says that the figure is a triangle because it *looks* like a triangle. In the same way, a cat is a cat because it looks like a cat, and a dog is a dog because it looks like a dog. No further justification is needed.[1] This is because deep learning methods learn from data, i.e., from training examples, rather than from prescribed rules. In this course we will study deep learning and discover how this is at all possible.

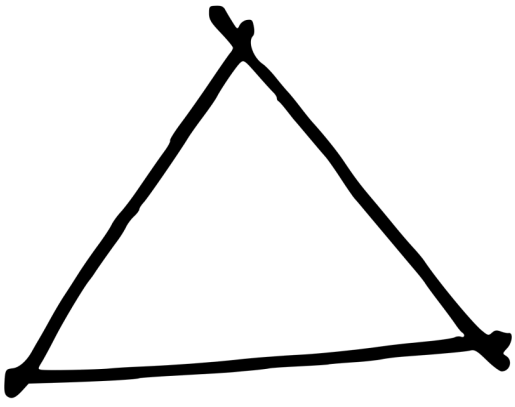


Figure 1: A triangular shape. But why? (Image credit: `https://www.kisscc0.com/`)

*A big thank you to my students and colleagues who provided invaluable feedback on earlier versions of these notes.

[1]This is not completely true—there is an entire research field devoted to the important topic of explainable AI, which requires that a human understandable justification be given for the outputs, answers, predictions, and decisions produced by an AI algorithm.

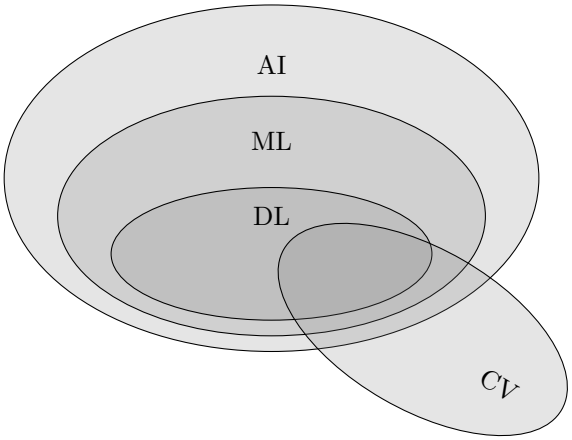


Figure 2: A Venn diagram showing where this course fits within the broader field of artificial intelligence.

1 Introduction

This lecture begins by contrasting deep learning against classical machine learning and computer vision approaches. It then gives a very quick tour of problems in computer vision and a high-level summary of machine learning. The lecture ends by discussing how data is represented and processed within deep learning systems, namely, via the tensor data structure and basic linear algebra operations. It is assumed the material in this lecture is mostly review and that most students would have seen it before.

Let us begin by positioning the field of deep learning within the broader scientific discipline of artificial intelligence (AI). Figure 2 provides a good illustration. We can loosely define AI as being about building machines that behave in an intelligent way. Machine learning (ML) is a sub-field of AI where the machines that we build improve their performance on some task by learning from data or with gained experience. Deep learning (DL) then is a sub-field of machine learning where the models that make up these machines—often called **neural networks**—are learned in an end-to-end fashion (i.e., with limited human engineering of features or model architectures).[2]

This course studies deep learning with a focus on computer vision, which has both heavily influenced the models and methods in deep learning, and as gained enormously from the results that deep learning has delivered. Computer vision (CV) is about developing algorithms for machines to understand the world behind images and videos. Some people think of this as inverse graphics. We will discuss various problems studied in computer vision later in the lecture. It is important to note that there are many areas in computer vision that are not related to deep learning (e.g., how cameras capture images of the world). Likewise, there are many other application fields that have contributed to and benefited from deep learning, the most notable of these include natural language processing, robotics, and bioinformatics.

Carl Sagan, the famous astronomer and science educator, once said, "You have to know the past to understand the present." We can pinpoint a precise moment in time when deep learning became a viable and popular research field. Although research into neural networks can be traced back to the 1960s [79, 87], it wasn't until 2012 where ideas from the past were put into a system that was truly competitive with other techniques for solving problems in computer vision and the field that we call deep learning was born. Roughly speaking, research and methods in machine learning and computer vision prior to 2012 were characterized by theory and provable algorithms. Systems involved a lot of feature engineering to get to work. Post 2012, deep learning methods have been characterized by data and compute, and building systems with end-to-end composeability. It is still not easy to get systems to work reliably all the time. But, as we will see, the knobs that we get to tune are different to the feature engineering of the past.

While deep learning is the dominant approach to solving problems in computer vision nowadays, it is important to mention that many classical ideas are still very relevant, either in their influence on deep learning techniques, in their adoption as components within larger deep learning systems, or as standalone algorithms. Moreover, developing theories and principles to better understand the behaviour and reliability of deep learning systems is becoming more and more urgent as deep learning algorithms are deployed in real-world engineering, economic, environmental, educational, societal, and health applications.

So what happened in 2012? A few years prior to 2012 a group of researchers from Stanford University led by Fei-Fei Li collected a massive dataset of images from the web (over one million). Each image in the dataset was annotated with a label that described its content employing the help of crowd-sourcing. There were images of dogs and cats of various breeds, cars, airplanes, foods, etc. One thousand different categories in total. The dataset was

[2] We will see a slightly different characterization shortly.

called ImageNet [20]. It was and remains one of the largest datasets of it's kind. To go along with the dataset, the Stanford group started the ImageNet large-scale visual recognition challenge (ILSVRC), where researchers from around the world could compete on who had the best image recognition algorithm, that is, whose algorithm would correctly recognise the most number of images in a hold-out test set. In 2012, Alex Krizhevsky, a graduate student of Geoffrey Hinton from the University of Toronto, entered the challenge with a deep learning based model [56]. It significantly beat all other models. This was a watershed moment for deep learning. The following year the top five performing entries were based on deep learning. The year after that, and every year since, all entries used deep learning and the error rate steadily dropped. Today deep learning outperforms humans on the ILSVRC challenge.

Since 2012 progress has continued to accelerate thanks various initiatives. First, the development and support of high-quality open-source software libraries (supported by industry) such as PyTorch [71], TensorFlow [1] and JAX [12], has made it easy to develop models and reproduce results from other researchers. Second, the availability of large and diverse datasets necessary to train models. Third, the existence of fast and cheap compute such as GPUs for accelerating the core calculations used in deep learning models.[3] And last, a growing community of researchers and engineers that facilitate the rapid sharing of knowledge, ideas, and results.

1.1 Tour of Computer Vision Problems

In this section we provide a very quick tour of problems studied in computer vision. The field is huge and these are just the tip of the iceberg to give you a flavour of the types of tasks being solved, and the language used to describe them. Many tasks are related and there there are many variants that differ in underlying assumptions, categories to be recognized, availability and quality of training data, additional or different imaging modalities, etc. All have been advanced in one way or another by deep learning.

1.1.1 Image Classification

Image classification tasks are characterized by assigning a single label to the entire image. Perhaps the most basic such problem is **binary image classification**. Here a single yes-no or true-false question is asked of an image. For example, *is this an image of a face?* Even for such a simple question the task is not completely clear: How much of the image does the face need to take up? Does the face need to front-on or do side views also count? What if there is more than one face? This type of ambiguity is typical of semantic-based computer vision tasks, where the specification is not rigorous but rather it is implicit in the datasets used—remember the triangle from Figure 1. In addition to the yes or no answer, algorithms often also provide a confidence score (e.g., 90% confident that the image is of a face).

Multi-class image classification or simply image classification goes beyond the binary classification task and asks the system to assign to the given image one label out of a fixed pre-defined set of labels, typically categories of objects. For example, *is this an image of a dog or a cat or a bird or ...?* Once again these tasks are somewhat ill-defined and a critical inbuilt assumption is that the image does neatly fit into one of the categories and the object (for object label sets) is dominant and well-framed within the image.

A refinement of multi-class image classification is **fine-grained image classification**. Here the task is considered to be more difficult than the vanilla multi-class image classification problem in that the objects are more visually similar. For example, *what type of bird is this?* Once again, it is assumed that objects, in this case the birds, are well-framed.

Going beyond a pre-defined set of categories, **open vocabulary image classification** allows images to be labeled with any noun, even those not seen during training of the model. A typical task of this type might be *what object is this?* This is a very challenging task and methods are usually augmented with an external knowledge base or pre-trained large vision-and-language model.

Scene classification is an specialized instance of multi-class image classification. Rather than naming the object in the image, the task is to categorize the type of scene, e.g., *is this forest or urban or ...?* Again the set of scene labels is fixed in advance and the task may be ambiguous (e.g., what distinguishes *city* from *urban*?).

Gesture recognition is another specialization of multi-class image classification, where the goal is to recognize the gesture being performed by a human in the image. These may be casual gestures such as *thumbs up* or more formal gestures such as in sign language recognition. While gesture recognition can be formulated as an image classification task, it is often more suitable to video data.

The last two tasks in this category of computer vision problems are **identity verification** (e.g., *is this Yann LeCun?*) and **identity recognition** (e.g., *who is this?*). The former assumes that the system is also given the presumed identity of the person and it simply needs to perform a binary classification task conditioned on that identity. The latter is a multi-class classification problem that may be open vocabulary, i.e., the identity of all participants may not be known in advance and need to be matched to an external knowledge base.

[3] Though, even with GPU-acceleration training large deep learning models can take several weeks or more. The original AlexNet model took five to six days to train using two NVIDIA GTX 580 GPUs [56].

1.1.2 Structured Prediction

Structured prediction describes machine learning models that output structured data rather than a single categorical label or regression value. In the context of computer vision the canonical example is **object detection** where the algorithm is expected to output a set of object labels and bounding box pairs corresponding to objects found in the image. This answers the question, *what and where are the objects in the image?* Another example is the **(human) pose estimation** task, where we are asking to *locate all body joints* such as hands, elbows, shoulders, etc. Here a skeleton model defines the joints and how they are connected in the model, which constrains the location of one joint with respect to another. A related task is to identify **landmarks** on an image of a human face, such as the eyes, nose, corners of the mouth, etc. This could be helpful as a pre-processing step for identity recognition or an generative AI approach that, for example, creates a video by animating the face in a natural way.

Human-object interaction (HOI) is a type of structured prediction task where we are asking *which human is interacting with which object (and how)?* This is typically done by performing object detection on people and objects separately and then exhaustively pairing up each human with each object, and classifying the pair with a verb describing the interaction. A special category to denote no interaction allows irrelevant pairs to be discarded.

Last, there is a very large and specialized research area on **text recognition**, which aims to *locate and read text* within the image. This has applications from number plate recognition for automating parking lot ticketing, to interpreting receipts for financial systems, to reading text in-the-wild for place recognition or language translation (e.g., of menus at foreign restaurants). Indeed, convolutional neural networks in deep learning can trace its roots to the task of digit recognition for automating the reading of addresses on envelopes and routing of mail in the US postal service.

1.1.3 Pixel Labeling

Pixel labeling tasks involve assigning one or more class labels to every pixel in the image. These labels can represent semantics, such as a sky pixel, or indicate membership such as all pixels having the same (integer) label belong to the same object. The most basic example of the latter type is **(unsupervised) image segmentation** or over-segmentation, which *breaks images up into meaningful regions* sometimes called **superpixels**. This is often a pre-processing building block for some more sophisticated downstream algorithm.

Semantic segmentation is the canonical example of the former type of pixel labeling. In this task, each and every pixel is the image is assigned a class label from a pre-defined set. You can think of this as being similar to image classification but now applied to every pixel in the image, asking the question *what class does this pixel belong to?* **Mask object detection** combines object detection with foreground-background segmentation to indicate exactly which pixels belong to each object rather than simply providing a bounding box around the object. Some pixels, i.e., those not belonging to any object, will remain unlabeled a hence considered as background.

Panoptic segmentation, which is related to multi-class multi-instance segmentation, labels each pixel with a semantic category (form a pre-defined set) and an instance identifier. This allows separation of multiple instances of the same object, e.g., multiple people in a crowd. Background categories are often considered to not have different instances. Sometimes researchers will use the terms **things** and **stuff** to distinguish between categories with distinct well-defined boundaries (and hence instances) versus those with amorphous shape and extent such as sky and grass.

Pixel labeling tasks are not restricted to semantic categories. **Monocular depth estimation** is a task that labels every pixel with its depth to the camera. If we have two cameras (or two views from the same camera) with known relative pose then the problem becomes **stereo depth estimation**, which can be solved by understanding the geometry of image formation and using traditional computer vision techniques [36].

Last, we can ask more ambiguous questions such as *what are the most important parts of the image?* resulting in a **saliency detection** task where each pixel is assigned a confidence score of belonging to the most salient object or part of the scene. This sometimes depends on context. The task can be applied iterative to give the second most salient region, and so on.

1.1.4 Matching, Sorting and Retrieval

Searching and sorting are core operations on which many downstream tasks are built. In **feature matching** we are interested in finding corresponding pixels between two (or more) images, i.e., pixels that represent the same location in 3D space. This helps with applications such as camera calibration, object tracking, and 3D reconstruction.

In **visual search** we are interested in finding information based on a query image. For example, given an image of a famous landmark such as the Sydney opera house we could ask *what is the name of this building?* **Visual place recognition** is similar in that it identifies whether two places are the same or (equivalently) estimates the location the camera in the world based on what it sees.

The last example in this category is **attribute ranking**, which is the task of sorting a set of images by some visual

attribute. For example, *sort shoes by sportiness or faces by happiness*. Here images need to be compared in a pairwise or higher-order setting to determine which more strongly aligns with the given attribute.

1.1.5 Multi-view, Geometry, 3D and Point Clouds

Geometry is fundamental to computer vision and its role in understanding the physical world cannot be understated. **Stereo vision** aims to estimate pixel disparity (how far a pixel has shifted) from two calibrated views and hence reconstruct the depth to objects in the scene. Depth is inversely related to disparity.

In **camera calibration** we wish to estimate camera intrinsics (i.e., focal length and lens distortion parameters) and extrinsics (i.e., location of the camera with respect to fixed 3D coordinate systems) from image taken of the scene. Often the images will contain a checkerboard pattern making finding correspondences between multiple cameras and determining any lens distortion easier. The related task of **camera localization** asks to find where cameras are with respect to one another.

A very important and popular research area is **structure from motion**, which aims to reconstruct 3D point clouds from multiple images. The images may be obtained from multiple cameras at the same time, a single camera taking images of a moving object, or a moving camera taking images of a static object. Dealing with a moving camera and multiple moving objects, or objects that can deform, adds further complication and is a particularly challenging task.

1.1.6 Video

There are several tasks in computer vision that require reasoning over temporally evolving scenes. In **tracking** we are asked to detect and track objects as they move through an environment where they may sometimes be occluded by other objects in the scene. **Person re-identification** is like person tracking but where the person may leave the scene for an extended period of time or appear in different camera views over time.

The task of **activity recognition** is akin to image classification but for video data. Specifically, we may ask the question *what activity is happening in this (short) video clip?* Just like image classification this can be extended to activity detection and activity segmentation for longer video clips.

Video stabilization is a post-processing task applied to a recorded video that aims to remove jitter caused by a handheld camera and motion blur caused by fast moving objects.

1.1.7 Vision and Language

One of the more impressive feats that deep learning has been able to achieve is unified reasoning over multiple data modalities such as vision and language. This manifests itself in many different tasks. In **image captioning** the goal is to describe an image in words. This is refined in **visual question answering** (VQA) where we ask specific free-form questions (i.e., not necessarily seen during training) of an image. For example, given an appropriate image where such questions make sense, *what color are her eyes? what is the mustache made of?*

Visual and language navigation (VLN) is a relatively new area of research where we ask a robot (or simulated agent) to follow a natural language navigation instruction using visual cues, e.g., *exit the dining room via the door next at the far end of the table, walk to the kitchen and stop in front of the fridge.*

Another popular vision and language task is to **locate a specific object in an image**, such as to *find the light blue truck*. Here the algorithm is expected to return a bounding box or segmentation mask of the object being localized. The task can be combined with VLN to navigate to a particular location and then identify a target object in the scene. Continuing from the example above we might ask the robot to *open the fridge and locate the bottle of milk*.

1.1.8 Denoising, Completion and Generation

Several computer vision tasks are concerned with editing or synthesizing images as the output. There has been a long history of work focused on **image denoising**, which aims to remove noise and restore corrupted images, **image colorization**, which aims to convert a black and white image to colour, and **image inpainting**, which aims to remove an object from an image and fill in the background with a plausible reconstruction.

More recently, deep learning has enabled tasks such as **novel view synthesis**, which depicts a scene from a viewpoint not previously seen by the model, i.e., different to where the image (or set of images) was originally taken. Neural radiance fields (NeRFs) are a popular class of deep learning model for novel view synthesis. Another task is **restyling**, which changes the style but not the viewpoint or content of a scene, e.g., *depict an adult as a child*. We can also create completely novel (fake) images and videos using **generative AI** from text prompts such as *an image of an astronaut riding a horse* and *a video of a woolly mammoth*.

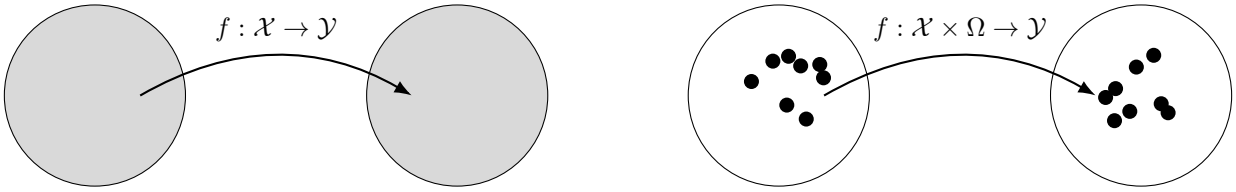


Figure 3: Machine learning from 10,000ft.

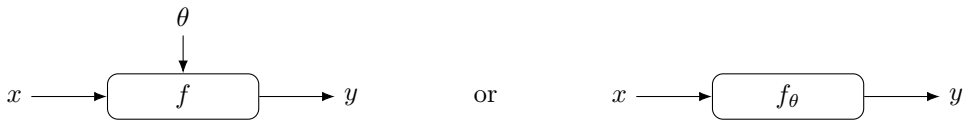
1.2 Machine Learning from 10,000ft

At its core machine learning is about finding a function f that maps from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$ (see Figure 3). Since we can't practically search over all possible mappings we define a function class parametrized by θ and train a model on a dataset of samples from $\mathcal{X}$ and $\mathcal{Y}$, denoted $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$. For example, the function class may be restricted to only linear functions or only polynomials or neural networks of a given architecture. The goal is for the model perform well over the entire space—it is no good to just memorize the training samples. This is, of course, very difficult since much of the space is unseen (i.e., not covered by samples in $\mathcal{D}$). The mapping f is learned by minimizing (over the parameters θ of the model) a loss function L, that in some sense measures the inaccuracy of the model. Typically, the loss function decomposes over sampled input-output pairs,

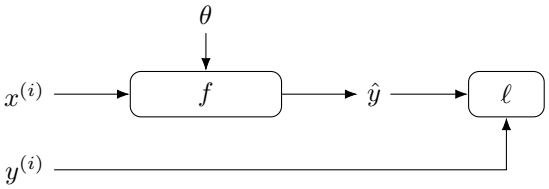
$$\text{minimize}_\theta \quad L(\theta; \mathcal{D}) \triangleq \textstyle\sum_{(x,y)\in\mathcal{D}} \ell(f(x;\theta), y) \tag{1}$$

This is called **empirical risk minimization** [84].[4] Here the loss function L tells us **what** to do, and the parametrized function $f(\cdot;\theta)$ tells us **how** to do it.[5] The objective (composed of the loss L and the mapping function f) is, in general, nonconvex in the parameters θ. As such, we can only hope to find a local minima of the learning problem, which we do using **gradient descent** or one of its variants.[6]

It is common to represent the machine learning model diagrammatically as,[7]



We can also incorporate the loss function in the diagram when we want to stress how the model is trained,



To summarise, empirical risk minimization (ERM) [84] is a principled framework for choosing model parameters in machine learning. The theory of empirical risk minimization states that we should choose the parameters of the model θ that minimize L over all possible parameters (i.e., the "hypothesis" class). To prevent overfitting on the training dataset or to incorporate prior information into the model we often include a **regularization term** on θ,

$$L(\theta; \mathcal{D}) = \sum_{i=1}^{N} \ell(f(x^{(i)};\theta), y^{(i)}) + \lambda R(\theta) \tag{2}$$

where λ is a hyperparameter that controls the regularization strength. This is known as regularized empirical risk minimization. The most common regularization function is the squared-norm of the parameters, $R(\theta) = \frac{1}{2}\|\theta\|_2^2$, which has a nice theoretical interpretation and also results in a simple modification to gradient descent based update rules.

[4] The function L is often scaled by $1/N$. Technically it is then the expected loss called **risk** with the function ℓ being the **loss**. However, most researchers will use the terms interchangeably, or simply refer to any function being minimized in deep learning a loss function.

[5] Often you'll see notation f_θ, indicating that θ indexes f from some function class $\mathcal{F}$. However, in these notes we prefer $f(\cdot;\theta)$ to make it clear that the function takes parameters as an argument and avoid confusion when discussing function compositions in later lectures.

[6] In practical applications suboptimal solutions are often desirable since finding the globally optimal solution can result in poor generalization of the model to unseen test samples. Techniques such as regularization, data-augmentation, and model selection on a hold-out validation set all help in this regard, and will be discussed in later lectures.

[7] In the sequel we will sometimes use symbol y to denote the target output (i.e., groundtruth) and sometimes the estimated output of the network (i.e., as a shorthand for $f(x;\theta)$). At other times we will use $\hat{y}$ for the estimated output. The meaning should be clear from the context.

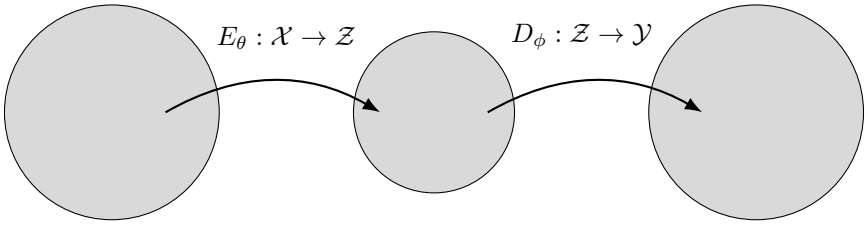


Figure 4: A popular paradigm in deep learning involves an encoder E_θ that maps the input into some latent space followed by a decoder D_ϕ that maps from the latent space to the output space. This has numerous applications from language translation to image generation.

A very common pattern in deep learning is the **encoder-decoder** paradigm. Here one model, the encoder, maps from the input space $\mathcal{X}$ to some latent space $\mathcal{Z}$. A second model then maps from the latent space to the output space $\mathcal{Y}$, as shown in Figure 4. The encoder and decoder models each have their own parameters and can be trained together or separately. Usually the latent space is smaller than the input or output spaces, and hence acts to compress the representation of the data. As a graphical illustration of this, we often draw encoder functions (E) and decoder functions (D) as follows,

Parameters are often omitted from these diagrams for brevity. Owing to the shape when the diagrams are combined, encoder-decoder models are sometimes called **hourglass** models.

Encoder-decoder models can be divided into **auto-encoders**, where the input and output spaces are the same and **cross-encoders**, where the input and output spaces are different. One use of an auto-encoder is to learn a low-dimensional (and sometimes sparse) representation of the data. We do not need explicit supervision for this task since the aim is to reconstruct the input x from its encoded version $z = E(z; \theta)$, i.e., we want $y = D(E(x; \theta); \phi)$ to be close to x. The latent variable z is called a **bottleneck** between x and y.

Cross-encoders have numerous applications. Indeed, many of the recent advances in deep learning and generative AI can be considered as a cross-encoder. With convolutional neural network (CNN) encoders and decoders we can implement semantic segmentation and style transfer algorithms; with recurrent neural network (RNN) encoders and decoders we can implement language translation models; and with a mixed CNN encoder and RNN decoder we can implement image captioning. Multiple different encoders can also be used to learn **joint embedding** spaces for images and language (e.g., CLIP [74] and BLIP [60]), allowing us to map from one semantic space to another.

1.2.1 Ingredients of a Machine Learning System

There are several ingredients that make up a machine learning system. They should be considered carefully whenever designing, debugging or deploying a system for a specific task. Perhaps the most obvious ingredient is the **data**, which includes the input signals (i.e., set of images) and the target labels for supervised tasks. The data is used for training as well as evaluating the system, and we need to be sure there is sufficient quantity and quality for both of these roles. How these are **represented**, that is what datastructures are used for storing and processing, plays an important part of the system design. For example, labels can be represented as text strings, integers (each indexing a list of categories), or one-hot vectors; videos can be stored in a compressed encoded format or as a sequence of extracted frames.

Next, the **model architectures** (also known as networks) and **training objectives** (i.e., losses and priors), which we touched on above, are central to the workings of the system. Equally important is the **learning algorithm** used to tune the model's parameters using the training data and guided by the training objective. A crucial element of the learning algorithm for deep learning systems is how the parameters are initialised as this can have a big affect on the resulting performance and convergence rate of the algorithm.

Last, is the **evaluation metrics**. These map our judgment of how a system should perform into quantifiable scores. Using a single measure of performance can often be misleading. For example, reporting detection rates for a positive class (e.g., detecting cancer) and ignoring false positives can be misled by a system that always returns positive. It is important to think critically of the evaluation metrics, what they mean and where they may fail.

Each one of these ingredients—data, representations, architectures, objectives, and metrics—encode **(inductive)**

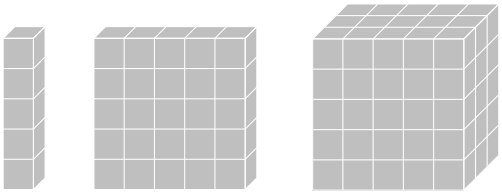


Figure 5: The basic datastructures in deep learning are vectors, matrices and tensors.

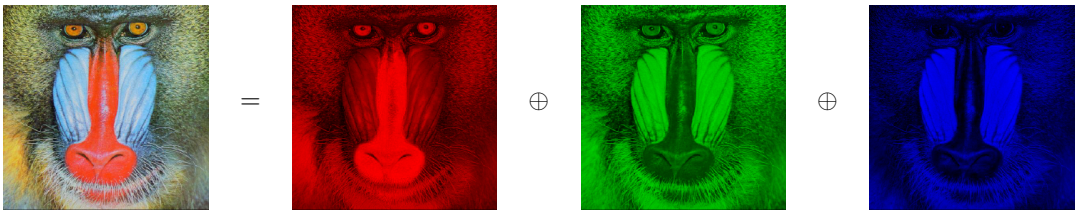


Figure 6: A colour image is represented using a third-order tensor composed of red, green and blue channels. (Mandrill image from USC Image Database, `https://sipi.usc.edu/database/`)

biases that affect system behaviour and provide levers to change behaviour. We will study each of them in this course in the context of computer vision and deep learning.

1.3 Data Representation and Linear Algebra Basics

Deep learning builds heavily on linear algebra (and a bit of differential calculus) for much of its numerical processing. Linear algebra is an incredibly rich area of mathematics and we will only review a few key topics here. We recommend the following online resources for students interested in delving more deeply into the field:

- Gilbert Strang's MIT lectures, `https://www.youtube.com/playlist?list=PL49CF3715CB9EF31D`
- 3Blue1Brown animated math videos, `https://www.3blue1brown.com/topics/linear-algebra`

The first thing we need to consider is how data is represented and stored. For the purposes of this course (mostly), a (column) **vector** is a one-dimensional array of numbers, a **matrix** is a two-dimensional array of numbers, and a **tensor** is an arbitrary-dimensional array of numbers (see Figure 5). The number of dimensions is called the **order** of the tensor, so an first-order tensor is just a vector.

The term **dimension** can also be used to specify the number of entries a vector, e.g., a 3-dimensional vector, but the meaning should be clear from the context. In deep learning we tend to describe vectors, matrices and tensors by their **size** or **shape**, e.g., a matrix with two rows and three columns, that is, a 2-by-3 matrix, has shape `(2, 3)`.

An **image** is an array of red, green and blue (RGB) colour pixels. As such it can be represented by a third-order tensor with integer values in the range 0–255 or floating-point values in the range 0.0–1.0 (see Figure 6). The order in which data is stored is not consistent across software frameworks: In OpenCV and `numpy` images have shape $(H, W, 3)$, whereas in PyTorch they have shape $(3, H, W)$, where H and W and the height and width of the image, respectively.

We will often refer to a colour image as a 3-**channel** or 3-**plane** image. Instead of colour, which can be thought of as a 3-dimensional vector, we can generalize to C-dimensional feature vectors for each pixel assembled into a **feature map** of size (C, H, W), also called a C-channel feature map. A video then can be represented using a fourth-order tensor with shape $(3, H, W, T)$, where T denotes the number of frames in the video.

1.3.1 Transpose

Given an $m \times n$ matrix, A, we can define its transpose, A^T, as the $n \times m$ matrix where the rows and columns of A have been swapped. For example,

$$A = \begin{bmatrix} 8 & 5 \\ 3 & 6 \end{bmatrix} \qquad A^T = \begin{bmatrix} 8 & 3 \\ 5 & 6 \end{bmatrix} \tag{3}$$

It should be clear from this definition that the transpose of a column vector is a row vector and vice versa, e.g.,

$$a = \begin{bmatrix} 8 \\ 5 \\ 3 \\ 6 \end{bmatrix} \qquad a^T = \begin{bmatrix} 8 & 5 & 3 & 6 \end{bmatrix} \tag{4}$$

The following properties of transpose are true:

- $\left(A^T\right)^T = A$
- $(AB)^T = B^T A^T$
- $(A+B)^T = A^T + B^T$

The idea of transpose extends to tensors where we then also need to specify exactly which channels are being swapped.

1.3.2 Addition and Multiplication

Two matrices of the *same size* can be added together in a componentwise fashion so that $C = A + B$ means

$$C_{ij} = A_{ij} + B_{ij} \quad \forall i = 1, \dots m \text{ and } j = 1, \dots n \tag{5}$$

The product of a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^n$, written $y = Ax$, is an m-dimensional vector with elements

$$y_i = \sum_{j=1}^{n} A_{ij} x_j \tag{6}$$

for $i = 1, \dots, m$. The cost of matrix-vector multiplication is $O(mn)$.

The product of two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ is an $m \times p$ matrix with elements

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{7}$$

for $i = 1, \dots, m$ and $j = 1, \dots, p$.

Note that the order of the matrices matters and for the product to exist we need the number of columns of A to equal the number of rows of B. However, matrix multiplication is associative, $(AB)C = A(BC)$, and distributive, $A(B+C) = AB + BC$. Matrix multiplication is not, in general, commutative, $AB \neq BA$.

The cost of matrix multiplication in $O(mnp)$.

Consider a matrix $A \in \mathbb{R}^{m \times n}$ and vector $x \in \mathbb{R}^n$. We can interpret the product $y = Ax \in \mathbb{R}^m$ in the following ways:

- the i-th entry of y is the inner product of the i-th row of A and x

$$y \quad = \quad \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} x \quad = \quad \begin{bmatrix} a_1^T x \\ a_2^T x \\ \vdots \\ a_m^T x \end{bmatrix}$$

- y is a linear combination of the columns of A

$$y \quad = \quad \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix} x \quad = \quad \sum_{i=1}^{n} x_i a_i$$

1.3.3 Inner Product and Euclidean Norm

The **inner product** between two vectors (of equal length) is defined by

$$\langle x, y \rangle = x_1 y_1 + \cdots + x_n y_n \tag{8}$$

$$= x^T y \tag{9}$$

Some important properties of inner product are:

- $\langle \alpha x, y \rangle = \alpha \langle x, y \rangle$
- $\langle x + y, z \rangle = \langle x, z \rangle + \langle y, z \rangle$
- $\langle x, y \rangle = \langle y, x \rangle$
- $\langle x, x \rangle \geq 0$

- $\langle x, x \rangle = 0$ if and only if $x = 0$

The row vector x^T represents a linear function $\mathbb{R}^n \to \mathbb{R}$.

For $x \in \mathbb{R}^n$, we define the (Euclidean) **norm** as

$$\|x\|_2 = \sqrt{x_1^2 + \cdots + x_n^2} \tag{10}$$

$$= \sqrt{x^T x} \tag{11}$$

The quantity $\|x\|_2$ measures the length of the vector (from the origin).

Some important properties of norm are:

- $\|\alpha x\| = |\alpha| \|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$ (triangle inequality)
- $\|x\| \geq 0$
- $\|x\| = 0$ if and only if $x = 0$

Other important (vector) norms are $\|x\|_1 = |x_1| + \cdots + |x_n|$ and $\|x\|_\infty = \max\{x_1, \cdots, x_n\}$.

1.3.4 Batch Processing and Broadcasting

Deep learning frameworks process data in batches, passed as tensors. The first dimension of the tensor is the batch dimension. So, for example, a batch of n-dimensional vectors has shape (B, N), and a batch of colour images has shape $(B, 3, H, W)$.

Example. For the operation $y = Ax + b$ we might have

$$X = \{x^{(1)}, \ldots, x^{(B)}\} \qquad \text{(input)} \tag{12}$$

$$Y = \{Ax^{(1)} + b, \ldots, Ax^{(B)} + b\} \qquad \text{(output)} \tag{13}$$

Many PyTorch functions are batch-aware and support broadcasting where data along singleton dimensions is automatically replicated to match arguments within an operation, e.g., for $(M \times N)$-matrix A, M-vector b, and batch of N-vectors x,

```
y = torch.matmul(A.view(1, M, N), x.view(B, N, 1)).view(B, M) + b.view(1, M)
# could also be done with: y = torch.einsum("ij,kj->ki", A, x) + b
```

computes $y^{(k)} = Ax^{(k)} + b$ on each element $k = 1, \ldots, B$ of the batch.

1.3.5 Tensors in PyTorch

In addition to their shape (`size`), tensors in PyTorch have several other properties:

- `dtype` to specify numerical precision, e.g., 32-bit float
- `device`, e.g., on CPU or GPU
- `requires_grad` will be discussed later
- `grad` holds gradient for training (will be discussed later)

There are many ways to create a tensor. For example,

```
A = torch.tensor([[8, 5], [3, 6]])
```

creates the matrix

$$A = \begin{bmatrix} 8 & 5 \\ 3 & 6 \end{bmatrix} \tag{14}$$

A `Parameter` is a type of learnable `Tensor` used within PyTorch modules that you will encounter later in the course.

It will often prove useful to be able to **reshape** a tensor when implementing deep learning models. Reshaping is always possible so long as the total number of elements does not change, e.g.,

```
A = torch.tensor([[8, 5, 1], [3, 6, 2]])
B = A.reshape([3, 2])
```

creates matrices

$$A = \begin{bmatrix} 8 & 5 & 1 \\ 3 & 6 & 2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 8 & 5 \\ 1 & 3 \\ 6 & 2 \end{bmatrix} \tag{15}$$

Note that this is different to transpose, which would create

$$A^T = \begin{bmatrix} 8 & 3 \\ 5 & 6 \\ 1 & 2 \end{bmatrix} \tag{16}$$

from PyTorch code

```
A.transpose(0, 1)
```

The method **view** can also be used whenever the memory layout of the underlying data does not need to change, e.g., `B = A.view([3, 5])`.

A very common reshaping is to **flatten** the tensor (sometimes called vectorizing), e.g., `C = A.flatten()` produces

$$C = \begin{bmatrix} 8 & 5 & 1 & 3 & 6 & 2 \end{bmatrix} \tag{17}$$

Note that elements are rearranged rowwise (called row-major ordering).

1.3.6 Coordinate Systems

A final comment on indexing and coordinate systems. In mathematics we generally index from 1, whereas in code we generally index from 0. This is a source of endless confusion and bugs. Now you know you wont fall into this trap.

To add further confusion, in linear algebra we index matrices and tensors from top-left to bottom-right, i.e., the top-left element of an $m \times n$ matrix is the $(1, 1)$-th entry and the bottom-right element is the (m, n)-th entry. Since we represent images as tensors, this will also be the default indexing for pixels. So the top-left pixel is the first pixel in the image. However, in geometry, coordinate systems increase upwards from the origin. In this coordinate system it is the bottom-left pixel that is the first pixel in the image. Be aware that different tools and different authors may use different coordinate systems. If you view an image and everything is upside down then flip the coordinate system.

1.4 Further Reading and Resources

The following is a list of recent textbooks that augments that material covered in these lecture notes:

- Goodfellow et al., *Deep Learning*, 2016 (`http://www.deeplearningbook.org/`)
- Zhang et al., *Dive into Deep Learning*, 2023 (`http://https://d2l.ai/`)
- Prince, *Understanding Deep Learning*, 2023 (`https://udlbook.github.io/udlbook/`)
- Scardapane, *Alice's Adventures in a Differentiable Wonderland*, 2024 (`https://www.sscardapane.it/alice-book`)

You should also familiarize yourself with the PyTorch deep learning library (`https://pytorch.org/`), which has excellent documentation and tutorials.

Finally, there are a number of very high quality lectures and courses online from various institutions. We list a few of them here:

- Stanford (Fei-Fei, Karpathy, et al.), `https://cs231n.stanford.edu/`
- Michigan (Johnson), `https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/`
- York University (Derpanis), `https://www.eecs.yorku.ca/~kosta/Courses/EECS6322/`
- University of Amsterdam (Lippe), `https://uvadlc-notebooks.readthedocs.io/en/latest/`

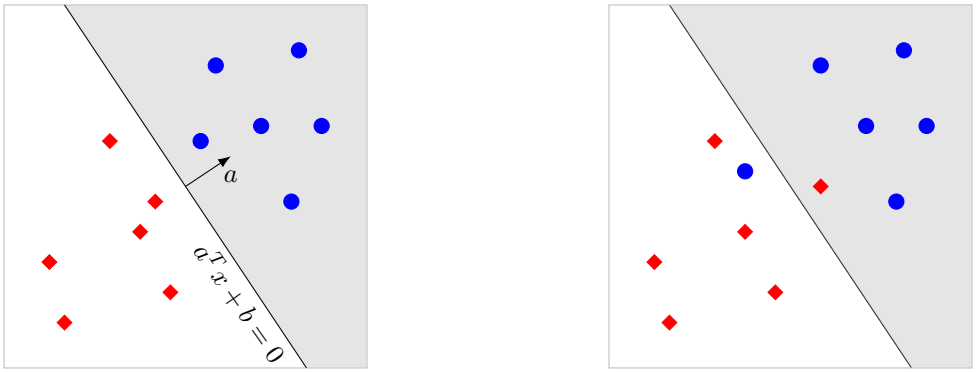


Figure 7: Example of linearly separable (left) and non-separable (right) data for binary classification in 2D.

2 Linear Classification and Multilayer Perceptrons

In this lecture we introduce the notion of logistic regression in the context of binary classification. We discuss the limitations of logistic regression for non-linearly separated data. This motivates the multi-layer perceptron (MLP), which can be shown (under mild assumptions) to be a universal function approximator. Building on the basic framework of MLPs, we discuss multi-class classification and gradient descent for fitting model parameters.

Binary classification involves data instances that can be divided into two distinct categories. For example, images of faces and images not of faces. Typically we are given a set of training examples $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$ composed of n-dimensional input features $x^{(i)} \in \mathbb{R}^n$ representing each data instance, and target labels $y^{(i)} \in \{0, 1\}$. Instances with label `1` are called **positive** (e.g., faces), while those with label `0` are called **negative** (e.g., not faces).

In this setting we can think of two different tasks. First, the **regression task** is to learn a density function $P(y \mid x)$ from the training examples, which estimates the probability of `0` or `1` on a new input x. Second is the **classifier task**, which learns a classifier function $f : \mathbb{R}^n \to \mathbb{R}$ from the training examples that predicts `1` on an input x if $f(x) \geq 0$ and `0` otherwise. Often the classifier function is a **linear**, or more correctly, affine, function of the features, $f(x) = a^T x + b$. These tasks are related as we will later see.

Figure 7 depicts two examples of binary classification in a two-dimensional features space. In the first example (left), a linear function can be found to correctly separate the positive examples from the negative examples. In the second example (right), no such linear function exists, and the data is said to be **inseparable**. Note that the affine function $a^T x + b$ measures (signed) distance from the hyperplane that divides the space.

2.1 Logistic Regression

Roughly speaking, logistic regression turns an affine function of the input into a probability distribution over outputs. Let $y \in \{0, 1\}$ be a random variable with distribution defined by

$$P(y = 1 \mid x) = \frac{\exp(a^T x + b)}{1 + \exp(a^T x + b)} \tag{18}$$

where a and b are parameters and $x \in \mathbb{R}^n$ is the observed feature vector. The **maximum likelihood principle** states that we should choose parameters a and b that maximizes the probability of the observed data under the model,

$$P(\mathcal{D}) = \prod_{i=1}^{N} P\left(y^{(i)} \mid x^{(i)}\right) \tag{19}$$

In practice, we (equivalently) minimize the negative log-likelihood function instead,

$$\ell(a, b; \mathcal{D}) = -\log\left(\prod_{i:y^{(i)}=1} \frac{\exp(a^T x^{(i)} + b)}{1 + \exp(a^T x^{(i)} + b)} \prod_{i:y^{(i)}=0} \frac{1}{1 + \exp(a^T x^{(i)} + b)}\right) \tag{20}$$

$$= -\sum_{i:y^{(i)}=1} (a^T x^{(i)} + b) + \sum_{i=1}^{N} \log\left(1 + \exp(a^T x^{(i)} + b)\right) \tag{21}$$

which is convex in variables $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$.

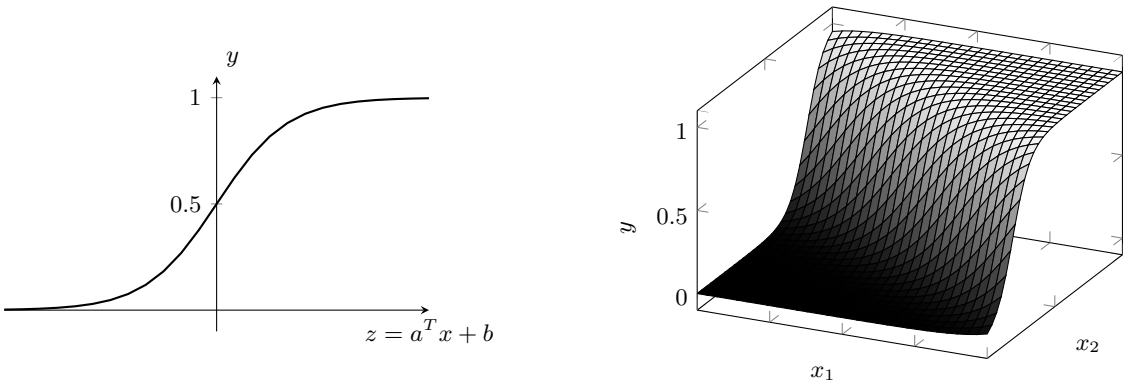


Figure 8: The logistic function, $y = (1 + \exp(-a^T x - b))^{-1}$, for 1D and 2D features.

The **logistic function** is an s-shaped curve (sigmoid) that maps from real numbers to the interval $[0, 1]$. It is therefore useful for modeling binary probability distributions. An extension of the logistic function is the so-called softmax function, which we will see later. The definition of the logistic function for scalar variable $z \in \mathbb{R}$ is,

$$\sigma(z) = \frac{\exp(z)}{1 + \exp(z)} \tag{22}$$

$$= \frac{1}{1 + \exp(-z)} \tag{23}$$

Replacing z with $a^T x + b$ allows us to learn a probability distribution conditioned on vector-valued features $x \in \mathbb{R}^n$,

$$P(y = 1 \mid x; a, b) = \sigma(a^T x + b) \tag{24}$$

as we have done above. A property of the logistic function is that $\sigma(z) \geq 0.5$ if, and only if, $z \geq 0$. So a classification rule $a^T x + b \geq 0$ corresponds to $P(y \mid x) \geq 0.5$, i.e., predict positive on an input x if the probability is above 50%.

An illustration of the logistic function for scalar $z = a^T x + b$ and two-dimensional features (x_1, x_2) is shown in Figure 8. Observe that the function is bound between zero and one. Note also that for the two-dimensional case (and higher-dimensions) when we look along the a direction we get a one-dimensional logistic curve.

Revisiting our example of binary classification over two-dimensional data, we can now see what a logistic function fitted to the data might look like as illustrated in Figure 9. Notice that in the separable case the logsitic curve is much sharper than in the non-separable case. In fact, in the separable case the function can be made arbitrarily sharp as we continue to train the model. In the non-separable case, the value of the logistic function gives us an estimate of the probability of an example being labeled positive.

Having our classifier too confident around the decision boundary, even in the separable case, is undesirable. Remember we are fitting to a set of training data, and what we really want is for our model to generalise to unseen test data. One way to reduce the confidence of the classifier on positive and negative training examples is through **regularization**. Here we add a penalty on the magnitude of the model's parameters, and minimize the resulting combination of loss function and regularizer,

$$\text{minimize}_{a,b} \quad \ell(a, b; \mathcal{D}) + \frac{\lambda}{2}\|a\|^2 \tag{25}$$

Note that we have not put any regularization on the offset, or bias, parameter b.

This type of regularization is also called **weight decay** because of it resulting in pushing the parameter values towards zero as evident when considering the gradient of the regularized loss,

$$\nabla_a \left(\ell(a, b; \mathcal{D}) + \frac{\lambda}{2}\|a\|^2 \right) = \nabla_a \ell + \lambda a \tag{26}$$

Ignoring the $\nabla_a \ell$ term for the moment, then taking a step in the negative gradient direction will change parameter a to $(1 - \eta\lambda)a$, where η is the gradient step size (or learning rate), to be discussed later in the lecture. That is, regularization tends to push a to zero. From the definition of the logistic function in Equation 23 it should be clear that as parameter a approaches zero, the value of the logistic $\sigma(a^T x + b)$ approaches a constant, i.e., $\frac{1}{1+e^{-b}}$. Hence, regularization tends to flatten the fitted logistic curve as show in Figure 10.

There are several other techniques for preventing over fitting that we'll see throughout the course, e.g., data augmentation. One such technique is **temperature scaling** that can be applied post-hoc. Here we introduce a temperature

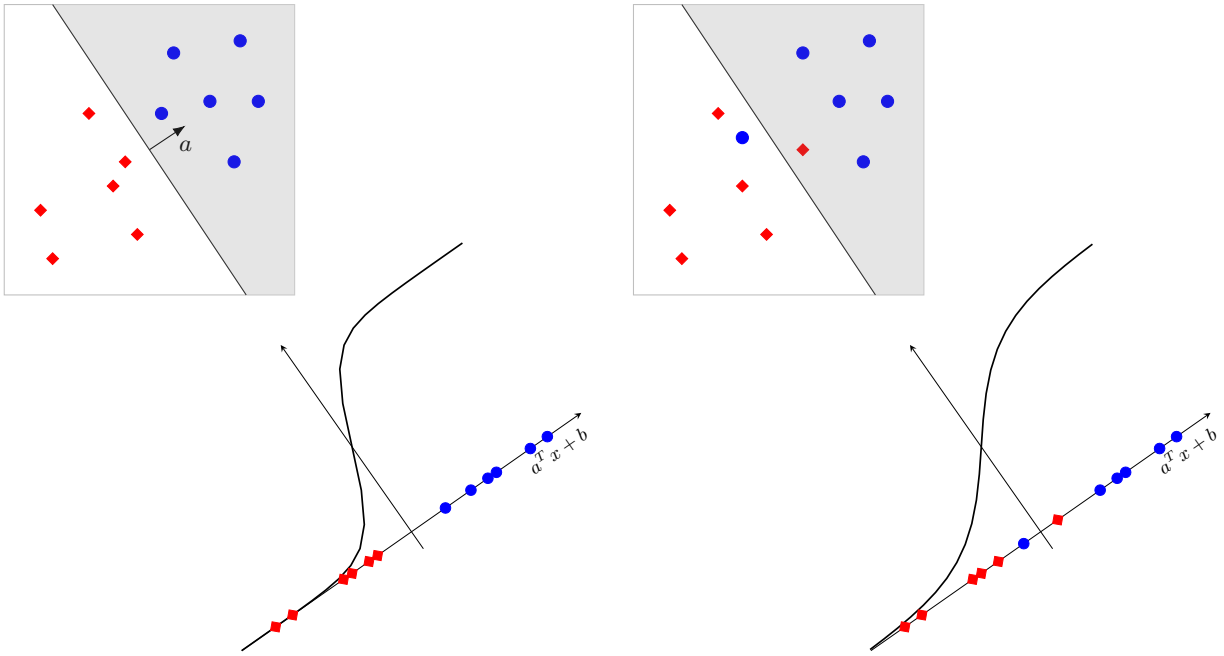


Figure 9: Illustration of logistic functions fitted to two-dimensional separable and non-separable data. The logistic curves are drawn aligned to the direction of a.

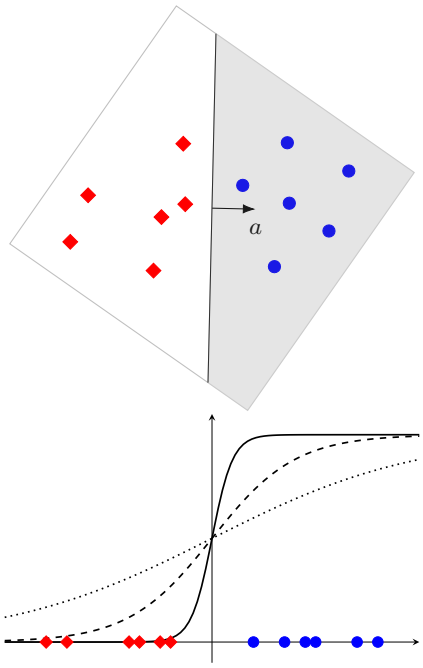


Figure 10: Regularization can be used to flatten a logistic function to reduce confidence around the decision boundary. We have rotated the feature space compared to Figure 9 for easier comparison of the different logistic curves.

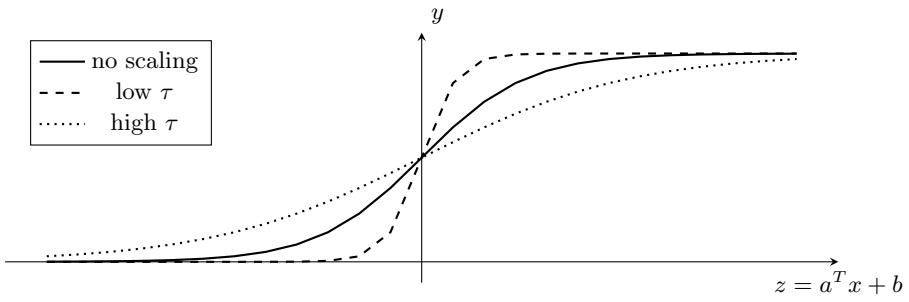


Figure 11: Temperature scaling. The parameter $\tau > 0$ controls the sharpness of the logistic curve.

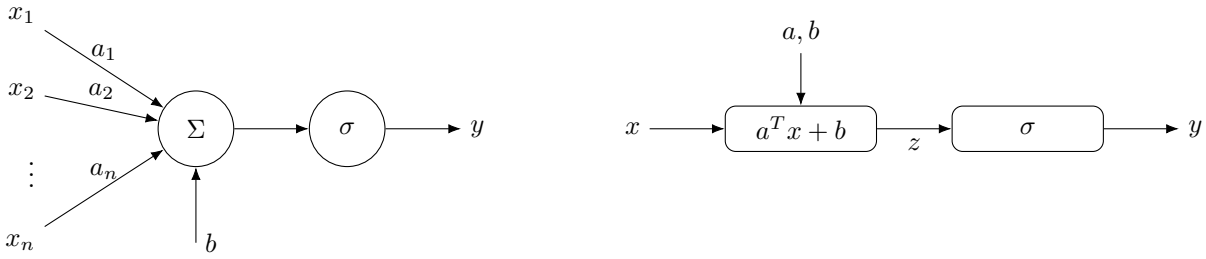


Figure 12: Classic view (left) and modern view (right) of a single neuron as either the explicit expression $y = \sigma\left(\sum_{i=1}^{n} a_i x_i + b\right)$ or more compact expression $y = \sigma(a^T x + b)$.

parameter $\tau > 0$ to the logistic function,

$$y = \sigma\left(\frac{z}{\tau}\right) = \frac{1}{1 + \exp\left(-z/\tau\right)} \tag{27}$$

The effect of τ is shown in Figure 11. A high value of τ has a similar affect to regularization, tending to flatten the curve. A value of τ less than one will sharpen the curve.

2.1.1 Logistic Regression as a Single Neuron

A very naive model of a biological neuron in the brain is a cell that fires its output if its accumulated input exceeds some threshold. We can very roughly view logistic regression as a single artificial neuron, as shown in Figure 12(left). The output y will be high if the weighted sum of the inputs $z = a^T x + b$ is greater than zero, and low otherwise. The arrows represent scalar signals. This is the classical depiction of a neuron in an artificial neural network, and is somewhat biologically inspired. The more modern view in deep learning is as a computation graph, illustrated more compactly in Figure 12(right). Here, the arrows can represent vector (or higher-order tensor) signals.

2.1.2 The XOR Problem and Multi-layer Perceptrons

As we saw earlier, a logistic regression classifier is only able to correctly classify linearly separable data. A famous example of a problem where the data cannot be separated by a single linear decision boundary is the XOR problem. See Figure 13(left). Here examples from the positive class (blue circles) are distributed in two clusters, one close to the point $(0, 1)$ and the other close to the point $(1, 0)$. Likewise, examples from the negative class (red diamonds) are distributed in two clusters, one close to the point $(1, 1)$ and the other close to the point $(0, 0)$. No straight line separates the two positive clusters from the two negative clusters.

The positive and negative examples can, however, be separated if we allow an additional layer of processing, as illustrated in Figure 13(right). In the first layer we construct a (linear) decision boundary $a_1^T x + b_1$ to separate the negative cluster close to $(1, 1)$ from the remaining three clusters. This can be thought of as implementing a NAND logic gate, $\xi_1 = \sigma(a_1^T x + b_1)$, where values close to 0 map to `False` and values close to 1 map to `True`. Still in the first layer, we construct another (linear) decision boundary $a_2^T x + b_2$ to separate the negative cluster close to $(0, 0)$ from the remaining three clusters. Similar to the first decision boundary, this can be thought of as implementing an OR logic gate, $\xi_2 = \sigma(a_2^T x + b_2)$. If we plotted ξ_1 versus ξ_2 we would find that the positive examples cluster around $(\xi_1, \xi_2) = (1, 1)$, whereas the negative examples cluster around either $(\xi_1, \xi_2) = (0, 1)$ or $(\xi_1, \xi_2) = (1, 0)$. As such, we can construct a second layer (linear) decision boundary $c^T z + d$, which acts as an AND logic gate on the first layer's output $z = (\xi_1, \xi_2)$ to solve the XOR problem.

In the preceding example the first layer of processing defines new features for the second layer. This motivates the **multi-layer perceptron** [87, 79]. More formally, a multi-layer perceptron is defined by composing multiple layers of

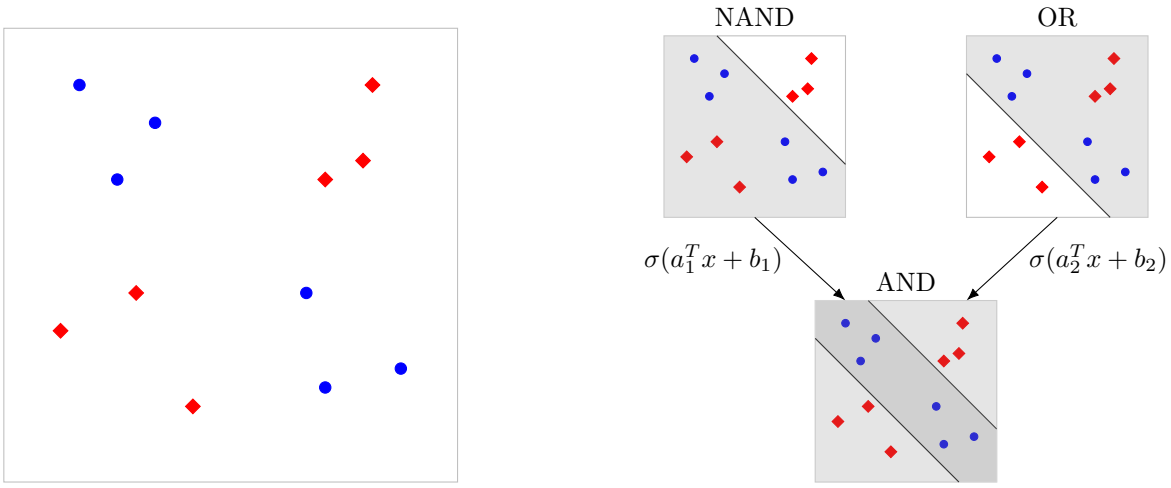


Figure 13: The XOR problem (left) can be solved with a two-layer network (right).

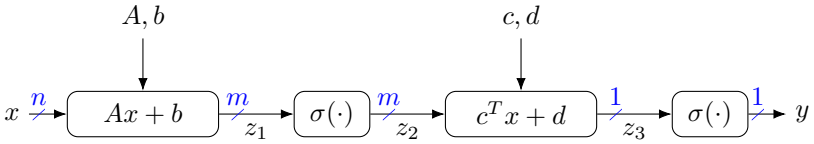


Figure 14: A two-layer percepton, implementing $y = \sigma(c^T \sigma(Ax + b) + d)$. Signal shape is indicated on the arrows.

logistic functions. For example, a two-layer perceptron is the composed function

$$y = \sigma(c^T \sigma(Ax + b) + d) \tag{28}$$

with parameters $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, and $d \in \mathbb{R}$, and where the logistic function σ is applied elementwise on $z_1 = Ax + b \in \mathbb{R}^m$. Parameters of multi-layer perceptrons (and other linear layers in deep learning) are often called **weights**. Figure 14 shows a graphical depiction of the two-layer perceptron.

To see concretely how the two-dimensional XOR problem fits the multi-layer perceptron formulation, observe that we can stack the parameters from the two neurons from the first layer as

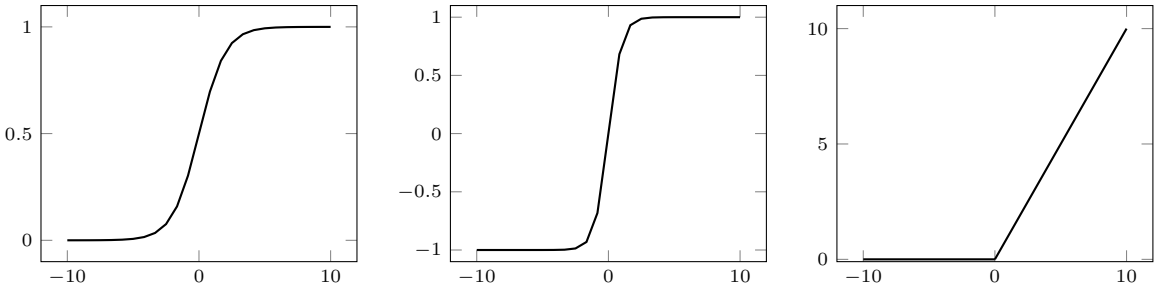
$$A = \begin{bmatrix} a_1^T \\ a_2^T \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{29}$$

giving

$$\begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} = \sigma\left(Ax + b\right) = \sigma\left(\begin{bmatrix} a_1^T x + b_1 \\ a_2^T x + b_2 \end{bmatrix}\right) = \begin{bmatrix} \sigma(a_1^T x + b_1) \\ \sigma(a_2^T x + b_2) \end{bmatrix}. \tag{30}$$

2.1.3 Activation Functions

Thus far we have only considered the logistic function as operating on linear combinations of the input features. The effect of the logistic function is to produce an elementwise non-linear transformation of the features. Transformations



(a) logistic (sigmoid), $1/(1 + e^{-z})$

(b) hyperbolic tangent, $\tanh(z) = (e^{2z} - 1)/(e^{2z} + 1)$

(c) rectified linear unit (ReLU), $\max\{0, z\}$

Figure 15: Common activation functions.

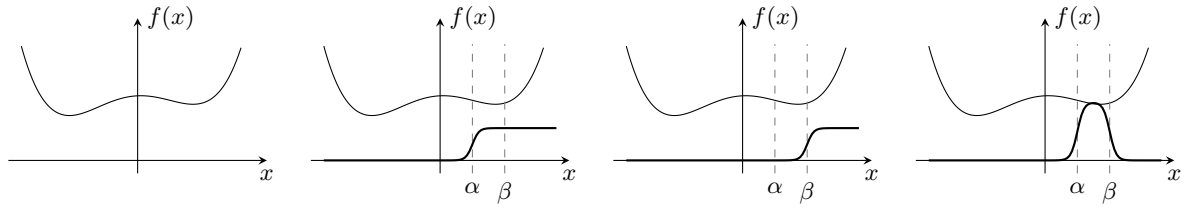


Figure 16: Visual argument for the universal approximation theorem. Shown from left to right: (a) A function that we wish to approximate. (b) An interval in the domain and sigmoid curve shifted to the start of the interval. (c) Another sigmoid shifted to the end of the interval. (d) Subtracting the second sigmoid from the first and scaling to the value of the function in the interval give an hump-shaped approximation to the function within the interval. This can be repeated by subdividing the entire domain into small intervals to approximate the whole function.

other than the logistic function are also possible. These go under the collective name of **activation functions**. Other common activation functions include the hyperbolic tangent function,

$$\sigma(z) = \tanh(z) \tag{31}$$

$$= \frac{\exp(2z) - 1}{\exp(2z) + 1} \tag{32}$$

and the rectified linear unit (ReLU),

$$\sigma(z) = \max\{0, z\} \tag{33}$$

These activation functions are plotted in Figure 15. Many other activation functions have been proposed in the literature, e.g., leaky ReLU, gated linear units (GLU), etc.

2.2 The Universal Approximation Theorem

The celebrated universal approximation theorem [18, 44] states that for any continuous function $f : \mathbb{R}^n \to \mathbb{R}$ and well-behaved activation function $\sigma : \mathbb{R} \to \mathbb{R}$ (such as the logistic function), then there exists parameters $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, and $d \in \mathbb{R}$ such that the function $\hat{f}$ defined by

$$\hat{f}(x) = c^T \sigma(Ax + b) + d \tag{34}$$

approximates the function f everywhere. That is, $|\hat{f}(x) - f(x)| \leq \epsilon$ for all x.

A visual argument for why the universal approximation theorem holds true is shown in Figure 16. The argument goes as follows. Suppose we have some function f. For simplicity we will assume that the function is defined over the reals (i.e., $n = 1$). Then we can break the domain of the function into small mutually exclusive and exhaustive intervals. For each interval $(\alpha, \beta]$ we can define two logistic functions, the first shifted to the start of the interval and the second shifted to the end of the interval. For example,

$$\sigma(x - \alpha) \quad \text{and} \quad \sigma(x - \beta) \tag{35}$$

Subtracting the second logistic from the first produces a hump-shaped function centred on the interval. We can scale this hump to approximate f over the interval. Let γ be the value of the function, say, at the midpoint of the interval, i.e., $\gamma = f\left(\frac{\alpha+\beta}{2}\right)$. Then the expression,

$$\gamma \left(\sigma(x - \alpha) - \sigma(x - \beta)\right) \tag{36}$$

is a good approximation to the function f in the interval $(\alpha, \beta]$. We can make the edges of the hump arbitrarily sharp using temperature scaling,

$$\gamma \left(\sigma\left(\frac{x - \alpha}{\tau}\right) - \sigma\left(\frac{x - \beta}{\tau}\right)\right) \tag{37}$$

Repeating this process for all of the intervals give the approximation for f everywhere, completing the argument.

The universal approximation theorem is a nice theoretical result—we can approximate any reasonable function with a two-layer network (without the final activation function). But the theorem suffers from two practical shortcomings. First, it does not tell us how many parameters we need in the network, and experience suggests that for two-layer networks we need a very large number of parameters to approximate functions well. Second, it does not tell us how

to find the parameter values in an efficient way. In turns out that we can reduce the number of parameters by going to deeper networks, i.e., more than two layers. Moreover, learning appears to be easier in deeper networks, although the theory here is not currently well understood. In practice, we often choose activation functions that violate the assumptions of the universal approximation theorem, such as ReLU, and this doesn't seem to affect the ability for deep networks to learn. Indeed, it often learns better.

A topic that is not taught much, but which is important for understanding learning and comparing trained networks is **identifiability**. This refers to whether we are able to uniquely identify a multi-layer perceptron's parameters from the function that it produces, i.e., its input-output relationship. The answer is that we cannot, i.e., the parameters of a neural network are never unique in that there exists a different set of parameters that produce exactly the same input-output mapping. We can see one specific example of how this may happen as follows. Let $P \in \mathbb{R}^{m \times m}$ be a permutation matrix and consider the two-layer perceptron from Figure 14, where we omit the final activation function for brevity. Then,

$$
\begin{aligned}
y &= c^T \sigma(Ax + b) + d && && (38)\\
&= c^T \sigma(P^{-1}PAx + P^{-1}Pb) + d && \text{(since } P^{-1}P = I\text{)} && (39)\\
&= c^T P^{-1} \sigma(PAx + Pb) + d && \text{(since } \sigma \text{ is applied elementwise)} && (40)\\
&= \tilde{c}^T \sigma(\tilde{A}x + \tilde{b}) + d && && (41)
\end{aligned}
$$

Therefore, parameters $\{A, b, c, d\}$ and $\{\tilde{A}, \tilde{b}, \tilde{c}, d\}$ produce exactly the same outputs.

2.3 Multi-class and Multi-label Classification

We now extend our discussion from binary classification to multi-class and multi-label classification. In **multi-class** or K-way classification we are interested in assigning one label out of a predefined discrete set to each data instance, such image classification on the ImageNet dataset. For this task we are given a set of training examples $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$ composed of features $x^{(i)} \in \mathbb{R}^n$ and target labels $y^{(i)} \in \{1, \ldots, K\}$. Note that while the target labels are often expressed as integers for compactness, they map to semantic categories such as *dog*, *cat* or *duck*. Another popular encoding is one-hot encoding where the targets are represented by a K-length vector with a one for the element associated with the class and zeros elsewhere, e.g., $(1, 0, 0)$ for *dog*, $(0, 1, 0)$ for *cat*, etc. Be aware that authors are often sloppy and the encodings are used interchangeably so that $y^{(i)}$ is sometimes an integer and sometimes a one-hot encoding depending on the context.

We can extend the logistic function to the multi-class logistic,

$$
P(y = k \mid x) = \frac{\exp(a_k^T x + b_k)}{\sum_{j=1}^K \exp(a_j^T x + b_j)} \qquad \text{for } k = 1, \ldots, K \tag{42}
$$

parametrized by $\{(a_k, b_k) \mid k = 1, \ldots, K\}$. The function that takes a vector $z = (z_1, \ldots, z_K) \in \mathbb{R}^K$ and returns $\left(\frac{\exp z_1}{Z}, \ldots, \frac{\exp z_K}{Z}\right)$ with $Z = \sum_{k=1}^K \exp z_k$ is called **softmax** and the z_k are called **logits**. The variable Z is called the **partition function** in machine learning and ensures that the output vector sums to one. In the context of the multi-class logistic we have $z_k = a_k^T x + b_k$. Softmax has the property that $\arg\max_k \left\{\frac{\exp z_k}{Z}\right\} = \arg\max_k \{z_k\}$, i.e., the index of the component that maximizes softmax is the largest logit. This means that when we want to use the model to make predictions (rather than estimate probabilities) then we can simply take the most likely prediction as the label corresponding to the largest logit.

If instead of restricting a data instance to just one label, we allowed the instance to take a set of labels, then the task becomes a **multi-label** classification problem. This is the same as having a set of K binary classification problems, and is typical of tasks like attribute classification. Once again we are given a set of training examples, $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$, composed of features $x^{(i)} \in \mathbb{R}^n$, but now the target labels are binary vectors, $y^{(i)} \in \{0, 1\}^K$ indicating true or false for each of the categories. We model the problem using a set of K sigmoid functions,

$$
P(y_k = 1 \mid x) = \frac{\exp(a_k^T x + b_k)}{1 + \exp(a_k^T x + b_k)} \qquad \text{for } k = 1, \ldots, K \tag{43}
$$

parametrized by $\{(a_k, b_k) \mid k = 1, \ldots, K\}$.

Both multi-class and multi-label problems can be solved using multi-layer perceptrons where the only difference is the final activation function, i.e., softmax or sigmoid. With parameters $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $C \in \mathbb{R}^{K \times m}$, and $d \in \mathbb{R}^K$, we can write,

$$
\begin{aligned}
y^{\text{multi-class}} &= \mathbf{softmax}(C\sigma(Ax + b) + d)\\
y^{\text{multi-label}} &= \mathbf{sigmoid}(C\sigma(Ax + b) + d)
\end{aligned}
$$

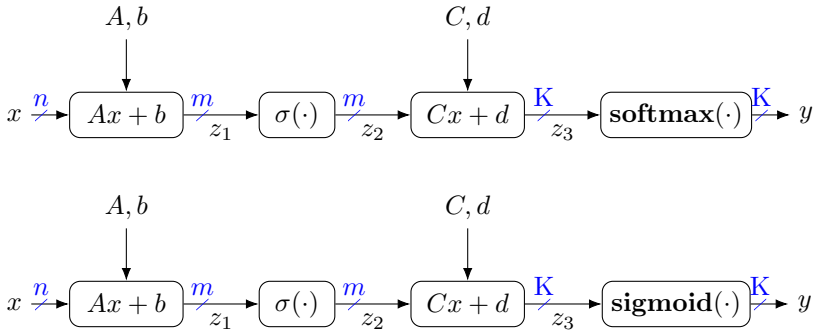


Figure 17: Multi-class (top) and multi-label (bottom) problems solved using a multi-class perceptron.

and depict graphically as shown in Figure 17.

There are several options for loss functions that train the models. Remember it is the loss function that tells the optimizer what to do and is a function of the model's parameters, collectively denoted by θ in the sequel. Unfortunately, the thing that we really care about, e.g., reducing the number of misclassified training examples, is not translatable into a loss that's easily optimized. Specifically, we seek a loss function that is differentiable. As such, a surrogate loss function is used.

The loss function that counts the number of misclassified examples is called the **0-1 loss**. It can be expressed mathematically as,

$$\ell^{\text{0-1}}(\theta) = \begin{cases} 0, & \text{if } f(x;\theta) \geq 0 \text{ and } y = 1 \\ 0, & \text{if } f(x;\theta) < 0 \text{ and } y = 0 \\ 1, & \text{otherwise} \end{cases} \tag{44}$$

but as mentioned above it is non-differentiable so difficult to optimize.

Standard loss functions that are differentiable are **mean square error** (MSE),

$$\ell^{\text{mse}}(\theta) = \frac{1}{2}\|f(x;\theta) - y\|^2 \tag{45}$$

for regression problems, and **negative log-likelihood** (NLL),

$$\ell^{\text{nll}}(\theta) = -\log P(y \mid x;\theta) \tag{46}$$

for classification problems. The latter is often called **cross entropy** loss when the probability is modeled by a multi-class logistic,

$$P(y \mid x;\theta) \propto \exp(f(x;\theta)) \tag{47}$$

or **binary cross entropy** loss when applied to K independent binary variables such as for multi-label problems,

$$-\sum_{k=1}^{K} y_k \log P_k(1 \mid x;\theta) + (1 - y_k) \log P_k(0 \mid x;\theta) \tag{48}$$

Note that all these loss functions decompose as a summation over training examples, $(x, y) \sim \mathcal{D}$.

2.4 Gradient Descent Optimization

We now turn our attention to the algorithm used to optimize the loss function—gradient descent. Let us begin by reviewing the definition of a gradient from multi-variate calculus. Let $f : \mathbb{R}^n \to \mathbb{R}$ be some function, fix some $x \in \mathbb{R}^n$, and consider the expression

$$\lim_{\alpha \to 0} \frac{f(x + \alpha e_i) - f(x)}{\alpha} \tag{49}$$

where $e_i = (0, \ldots, 1, 0, \ldots)$ is the i-th canonical vector.

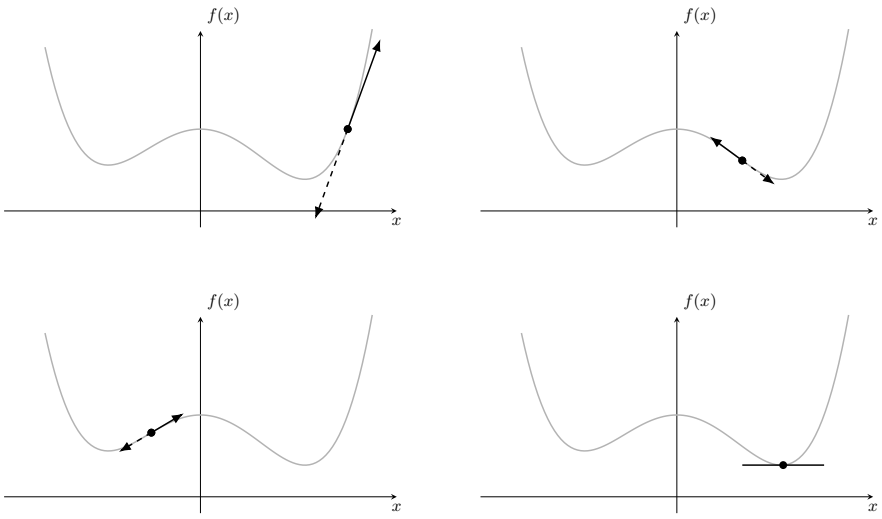


Figure 18: Gradient (solid arrow) shown at various points on the function. The negative gradient (dashed arrow) always points in a direction that reduces the function value, except at stationary points (i.e., local maxima, local minima and saddle points), where the gradient is zero. (Technically speaking, the gradient is the direction indicated in the figure projected onto the x-axis, and the arrow depicts an affine approximation to the function at the given point.)

If the limit exists, it is called the i-th **partial derivative** of f at x and denoted by $\frac{\partial f(x)}{\partial x_i}$. The **gradient** of $f(x)$ with respect to $x \in \mathbb{R}^n$ is the vector of partial derivatives,

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{50}$$

Figure 18 shows an example one-dimensional function. Several points on the function are highlighted, and the gradient at those points depicted by a solid arrow. The function decreases in the negative gradient direction. This is always true unless the gradient is zero (depicted by a horizontal line in the figure), which will occur at local maxima, local minima and saddle points. We can make use of this property to optimize the function.

The **gradient descent algorithm** is an iterative algorithm that takes steps in the negative gradient direction (of the loss function with respect to parameters) to seek a (local) minimum for the function. It works for smooth functions with any number of parameters. The steps can be expressed mathematically as,

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla \ell(\theta^{(t)}) \tag{51}$$

where η is the (iteration dependent) step size or step length or learning rate that controls how big a change is made to the parameters during each iteration. Taking too big a step (i.e., very large η) may result in overshooting the minimum, whereas taking too small a step may result in very slow progress. Ideally, η is chosen by searching along the negative gradient direction for a minimum value of the function, but this is expensive to do, amounting to solving a one-dimensional optimization problem. So instead of finding the best η, in deep learning we resort to choosing η via a fixed **learning rate schedule**.

Pseudo-code for the gradient descent algorithm is shown below:

```
def gradient_descent(loss, theta0):
    """Generic gradient descent method."""

    theta = theta0
    for t in range(max_iters):
        dtheta = -1.0 * gradient(loss, theta)   # descent direction is negative gradient
        eta = line_search(loss, theta, dtheta)  # choose step size by line search
        theta = theta + eta * dtheta            # update

    if (converged()):                           # check stopping criteria
        break

    return theta
```

The algorithm terminates when some **stopping criterion** is obtained, usually of the form $\|\nabla \ell(\theta)\|_2 \leq \epsilon$, or when a maximum number of iterations is reached.

Gradient descent is very simple, we just need the gradient of the loss ℓ with respect to the parameters θ at each iteration. However, it can be very slow in its vanilla form. In later lectures we will discuss methods used in deep learning to speed up the algorithm.

The following code shows what gradient descent code looks like in PyTorch.

```
1     dataloader = ...     # way of loading batches of input-label pairs for training
2     model = ...          # definition of our prediction function
3
4     criterion = torch.nn.CrossEntropyLoss(reduction='mean')
5     optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
6
7     for epoch in range(max_epochs):
8         for iteration, batch in enumerate(dataloader, 0):
9             inputs, labels = batch
10
11            # zero gradient buffers
12            optimizer.zero_grad()
13
14            # compute model outputs for given inputs
15            outputs = model(inputs)
16
17            # compute and print the loss
18            loss = criterion(outputs, labels)
19            print(loss.item())
20
21            # compute gradient of the loss wrt model parameters
22            loss.backward()
23
24            # take a gradient step
25            optimizer.step()
```

The code starts with some routines for loading the data, defining the model, specifying the criterion or objective function, which is the cross entropy loss in this case, and choosing the optimization algorithm (Lines 1–5). The code the enters the optimization loop (Lines 7–25) where is processes data in batches. We will discuss the mechanics of epochs (once through the dataset) and iterations (one gradient update step) in the next lecture. For now it is sufficient to understand that each iteration the code zeros out the memory used to store gradients (Line 12), computes the models output from its input (Line 15), and calculates the loss by comparing the model's output to the desired or target output (Line 18). This is called the forward pass. It then computes all necessary gradients via a so-called backward pass (Line 22). These gradients are accumulated into buffers associated with the model parameters, which is why it is important to zero out these buffers at the start of the loop. Last, the code updates the model parameters by taking a gradient step (Line 25).

A simple example of gradient descent from Boyd and Vandenberghe [11], that also demonstrated why it can be slow, is illustrated in Figure 19. In this example we are optimizing the function,

$$\ell(\theta) = \frac{1}{2}\left(\theta_1^2 + \gamma\theta_2^2\right) \qquad (\gamma > 0) \tag{52}$$

over a two-dimensional variable $\theta \in \mathbb{R}^2$. The constant γ gives us a family of functions. We start at $\theta^{(0)} = (\gamma, 1)$ and use exact line search. This allows us to calculate the iterates determined by the gradient descent algorithm in closed-form:

$$\theta_1^{(t)} = \gamma\left(\frac{\gamma-1}{\gamma+1}\right)^t, \quad \theta_2^{(t)} = \left(-\frac{\gamma-1}{\gamma+1}\right)^t \tag{53}$$

Note that convergence will be very slow if $\gamma \gg 1$ or $\gamma \ll 1$. In the figure we use $\gamma = 10$.

One question remains: how do we calculate the gradients? To finish the lecture we give a worked example for the two-layer perceptron with logistic activation functions. In the next lecture we will discuss gradient calculation for deep learning in much more detail and introduce automatic methods for performing the calculations for us. Nevertheless, it is a valuable exercise to work through computing gradients by hand at least once.

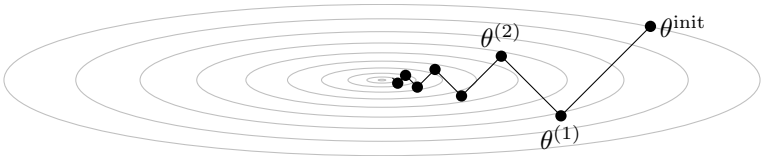


Figure 19: Example of gradient descent for two-dimensional quadratic problem, $\ell(\theta) = \frac{1}{2}(\theta_1^2 + \gamma\theta_2^2)$.

Consider the two-layer perceptron expressed as

$$z = \sigma(Ax + b) \qquad \text{(layer one)} \tag{54}$$

$$y = \sigma(c^T z + d) \qquad \text{(layer two)} \tag{55}$$

where $\sigma(\xi) = (1 + e^{-\xi})^{-1}$ is applied elementwise. We will assume an arbitrary differentiable loss function ℓ that we wish to minimize. As such, we can assume that $\frac{\mathrm{d}\ell}{\mathrm{d}y}$ is given. But we need to compute gradients of the loss for all of our parameters, i.e., $\frac{\mathrm{d}\ell}{\mathrm{d}A}$, $\frac{\mathrm{d}\ell}{\mathrm{d}b}$, $\frac{\mathrm{d}\ell}{\mathrm{d}c}$ and $\frac{\mathrm{d}\ell}{\mathrm{d}d}$, so that we can perform gradient descent.

First, observe that

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\xi} = (-e^{-\xi})(-1)(1 + e^{-\xi})^{-2} \tag{56}$$

$$= \left(\frac{1}{1 + e^{-\xi}}\right)\left(\frac{e^{-\xi}}{1 + e^{-\xi}}\right) \tag{57}$$

$$= \sigma(\xi)(1 - \sigma(\xi)) \tag{58}$$

Starting at the second layer, let $\xi = c^T z + d$. We have, by the chain rule of differentiation,

$$\frac{\partial \ell}{\partial d} = \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\partial y}{\partial d} \tag{59}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\frac{\partial \xi}{\partial d} \tag{60}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1 - y) \tag{61}$$

since $\frac{\mathrm{d}y}{\mathrm{d}\xi} = y(1 - y)$ by our observation and $\frac{\mathrm{d}\xi}{\mathrm{d}d} = \frac{\mathrm{d}c^T z + d}{\mathrm{d}d} = 1$, and

$$\nabla_c \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y}\nabla_c y \tag{62}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\nabla_c \xi \tag{63}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1 - y) z \tag{64}$$

where in the last line we, again, used the fact that $y = \sigma(\xi)$ and $\nabla_c \xi = \nabla_c (c^T z + d) = z$.

Moving backward onto the first layer, we have

$$\nabla_z \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1 - y) c \tag{65}$$

following the same pattern as $\nabla_c \ell$. Then, for the first layer's parameters,

$$\nabla_b \ell = \nabla_z \ell \circ z \circ (1 - z) \tag{66}$$

$$\nabla_A \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y} y(1 - y)\left(c \circ z \circ (1 - z)\right) x^T \tag{67}$$

where $\circ$ denotes elementwise product. These expressions use vector operations and can be a little intimidating when seen for the first time. If you find these difficult to interpret, try computing gradients on individual parameters using only scalar-valued derivatives, e.g.,

$$\frac{\mathrm{d}\ell}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\left(\sum_k \frac{\mathrm{d}\xi}{\mathrm{d}z_k}\frac{\mathrm{d}z_k}{\mathrm{d}A_{ij}}\right) = \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}z_i}\frac{\mathrm{d}z_i}{\mathrm{d}A_{ij}} \tag{68}$$

where $z_k = [\sigma(Ax+b)]_k = \sigma(\sum_p A_{kp}x_p + b_k)$ and noting that $\frac{\mathrm{d}z_k}{\mathrm{d}A_{ij}} = 0$ if $k \neq i$.

Notice that we have reused calculations from evaluation of ℓ, i.e., y and z, in our expressions for the gradients. We could just have legitimately written the gradients in terms of the input x only. How we write these expressions leads to an important consideration of memory-vs-compute trade-off that will be discussed in later lectures. Stay tuned.

3 Back-propagation and Learning

This lecture covers the technology at the heart of deep learning: back-propagation. We begin with a brief review of supervised machine learning and optimization by gradient descent. Then by considering deep learning models as computation graphs, or equivalently function compositions, we show how gradients can be automatically computed. We cover some practical considerations and conclude with an example of putting it all together to learn a simple multi-class classifier.

Recall that in supervised learning we are given a parametrized model $y = f(x;\theta)$ and a set of training examples $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$. Our goal is to find the parameters θ so that $f(x^{(i)};\theta)$ is a good predictor of $y^{(i)}$ for all $i = 1,\dots,N$. We do this by defining a regularized loss function L that measures how well the model fits the training data. The loss function usually decomposed over the training data plus a prior on the parameters,

$$L(\theta) \triangleq \sum_{i=1}^{N} \ell\Big(f(x^{(i)};\theta), y^{(i)}\Big) + R(\theta) \tag{69}$$

We then optimize the loss function with respect to the parameters to find the best model

$$\theta^\star = \arg\min_\theta L(\theta) \tag{70}$$

The simplest algorithm for doing this is gradient descent, which iteratively updates θ as

$$\theta \leftarrow \theta - \eta \nabla_\theta L \tag{71}$$

where η is a step size, either fixed or chosen via some step size schedule.

The most basic model we can think of in deep learning is the multi-layer perceptron (MLP), which is a composition of affine transformations with elementwise non-linear transforms. Concretely, let $x \in \mathbb{R}^{p_0}$ be the input data, $z_j \in \mathbb{R}^{p_j}$ be the hidden layer features, and $y \in \mathbb{R}^{p_n}$ be the output. Each layer computes its output as

$$z_j = f_j(z_{j-1}) = \sigma_j(A_j z_{j-1} + b_j) \tag{72}$$

where $A_j \in \mathbb{R}^{p_j \times p_{j-1}}$ and $b_j \in \mathbb{R}^{p_j}$ are parameters, and σ_j is a non-linear activation function. To simplify this expression so that it applies to all layers in the network, we have defined $x \triangleq z_0$ and $y \triangleq z_n$. Hence, the multi-layer perceptron defines the composition

$$y = (f_n \circ \cdots \circ f_2 \circ f_1)(x) \tag{73}$$

$$= \sigma_n(A_n \sigma_{n-1}(\cdots \sigma_1(A_1 x + b_1)) + b_n) \tag{74}$$

The idea of composing affine functions and non-linear activations in multi-layer perceptrons can be generalised in the sense that the function $f(\cdot;\theta)$ can be an arbitrarily composition of simple differentiable parametrized sub-functions, and where the parameters of these functions are optimised end-to-end. The composed function can be represented by a computation graph as illustrated in Figure 20, where the sub-functions are depicted as nodes in the graph. This type of representation is very popular for describing deep learning architectures where nodes can denote anything from a very simple arithmetic operation to very complicated algorithmic procedures and data transformations.

To compute the output of the function, we sort the nodes in topological order and evaluate each sub-function in turn,

$$\begin{aligned} z_1 &= f_1(x;\theta_1) \\ z_2 &= f_2(z_1;\theta_2) \\ &\vdots \\ z_7 &= f_7(z_6;\theta_7) \\ y &= f_8(z_4, z_7;\theta_8) \end{aligned} \tag{75}$$

To compute the derivative of a loss function at the output of the graph, with respect to any parameter or input of the graph, we simply apply the chain rule of differentiation by following the arrows backwards through the graph. This is known as **back propagation**. Two examples are shown in Figure 21 for computing the derivative of the loss L with respect to parameter θ_7 and parameter θ_1, respectively. Here writing out the chain rule we have

$$\frac{\partial L}{\partial \theta_7} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial \theta_7} \qquad \text{and} \qquad \frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial y}\left(\frac{\partial y}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1} + \frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial z_6}\frac{\partial z_6}{\partial z_5}\frac{\partial z_5}{\partial z_4}\right)\frac{\partial z_1}{\partial \theta_1} \tag{76}$$

where, for the latter derivative $\frac{\partial L}{\partial \theta_1}$, the first term in the summation is from the top branch and second term in the summation is from the bottom branch of the graph.

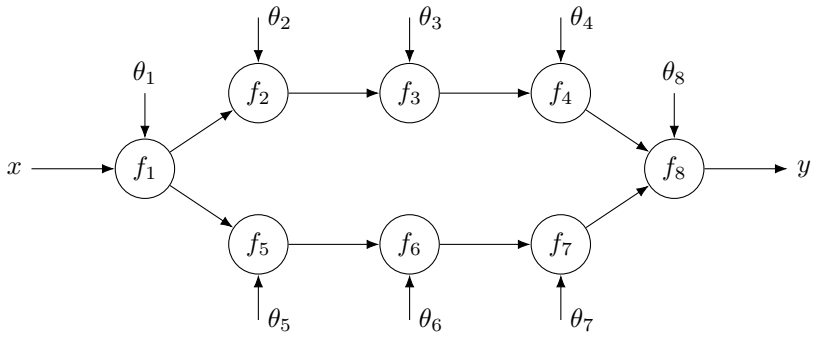
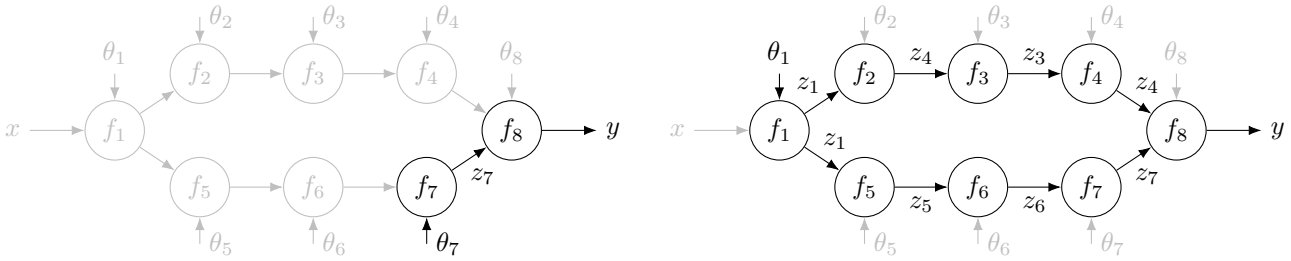


Figure 20: Example of a deep learning model as an end-to-end computation graph. This graph implements the composed function $y = f_8(f_4(f_3(f_2(f_1(x)))), f_7(f_6(f_5(f_1(x)))))$ where each f_i's parameters have been omitted for brevity. Note here that f_8 takes three arguments: one the output from f_4, another the output from f_7, and the last parameters θ_8.



(a) Computing $\frac{\partial L}{\partial \theta_7}$.

(b) Computing $\frac{\partial L}{\partial \theta_1}$.

Figure 21: Back-propagation of gradients through the computation graph. See text for details.

3.1 Notation

Before proceeding it is worth clarifying notation for gradients. For scalar-valued functions $f : \mathbb{R} \to \mathbb{R}$ we denote by

$$\frac{\mathrm{d}f}{\mathrm{d}x} \tag{77}$$

the total derivative of function f with respect to argument x. If a function takes more than one argument we can differentiate with respect to each argument separately by taking partial derivatives denoted by, for example,

$$\frac{\partial f(x,y)}{\partial x}. \tag{78}$$

What is often confusing is when y is also a function of x. Then calculus dictates that the total derivative with respect to x is the sum of direct and indirect terms,

$$\frac{\mathrm{d}f(x,y)}{\mathrm{d}x} = \frac{\partial f(x,y)}{\partial x} + \frac{\partial f(x,y)}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x}. \tag{79}$$

We already saw in the previous lecture that for multi-variate functions $f : \mathbb{R}^n \to \mathbb{R}$ we denote the gradient by

$$\nabla f(x) = \left(\frac{\mathrm{d}f}{\mathrm{d}x_1}, \ldots, \frac{\mathrm{d}f}{\mathrm{d}x_n}\right) \tag{80}$$

which is an n-dimensional (column) vector. More generally, for multi-variate vector-valued functions, $f : \mathbb{R}^n \to \mathbb{R}^m$ we define

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x_n} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_m}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_m}{\mathrm{d}x_n} \end{bmatrix} \tag{81}$$

as the m-by-n matrix of total derivatives, sometimes called the Jacobian matrix. Note that this is the transpose of ∇f for scalar-valued functions. For functions with signature $f : \mathbb{R}^n \times \mathbb{R}^{\tilde{n}} \to \mathbb{R}^m$ we can also define the matrix of partial derivatives,

$$\frac{\partial}{\partial x} f(x,y) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \tag{82}$$

This convention makes the chain rule particularly easy to express by, for example, replacing the scalar products in Equation 79 with matrix multiplications and following the same ordering of expressions. We also have that the affine function $y = Ax + b$ has derivative $\frac{\mathrm{d}y}{\mathrm{d}x} = A \in \mathbb{R}^{m \times n}$ without having to introduce transposes, which is nice.[8]

Symbols D and D_X are also used to denote the derivative operators, which is cleaner that writing $\frac{\mathrm{d}}{\mathrm{d}x}$ and $\frac{\partial}{\partial x}$, especially with inline text, but is less familiar to computer science students so won't be used here. Some authors use D_1 for partial derivatives with respect to the first argument, D_2 for the second, etc.

Last, and perhaps most importantly, many authors (including ourselves) are sometimes sloppy with notation and the reader should carefully check the intent from the context.

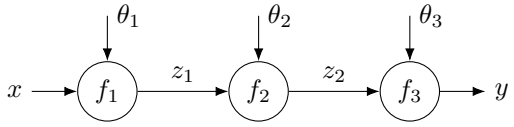
3.2 Automatic Differentiation

Automatic differentiation (AD) is an algorithmic procedure that produces code for computing exact derivatives (up to machine precision). This is in contrast to numerical methods for estimating gradients such as with finite differences. The approach assumes that all calculations are composed from a small set of elementary operations that we know how to differentiate, e.g., arithmetic operations, exponentiation, logarithms, and trigonometric functions.

It is not an exaggeration to say that modern deep learning would not be possible without automatic differentiation—it greatly reduces development and testing effort, and allows for rapid prototyping of new ideas.

Automatic differentiation comes in two flavours. **Forward mode** automatic differentiation fixes an independent variable u and computes derivatives $\frac{\mathrm{d}v}{\mathrm{d}u}$ for all dependent variables v. In contrast, **reverse mode** automatic differentiation fixes a dependent variable v and computes derivatives $\frac{\mathrm{d}v}{\mathrm{d}u}$ for all independent variables u.

Consider the simple three-node network below, where a loss function L is computed on the output y, itself a function of the input x and parameters θ_1, θ_2 and θ_3,



The chain rule of differentiation allows us to write the derivative of the loss with respect to parameter θ_1 as,

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}. \tag{83}$$

Forward mode automatic differentiation evaluates this expression from right to left, first computing $\frac{\mathrm{d}z_2}{\mathrm{d}\theta_1} = \frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}$, then $\frac{\mathrm{d}y}{\mathrm{d}\theta_1} = \frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}\theta_1}$, and finally $\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\theta_1}$,

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\left(\frac{\mathrm{d}y}{\mathrm{d}z_2}\left(\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{\mathrm{d}z_2/\mathrm{d}\theta_1}\right)\right) \tag{84}$$

so by the end we have computed $\frac{\mathrm{d}v}{\mathrm{d}\theta_1}$ for all variables $v \in \{z_1, z_2, y, L\}$.

Reverse mode automatic differentiation evaluates the other way around, i.e., from left to right,

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \left(\left(\underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{\mathrm{d}L/\mathrm{d}z_2}\right)\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\right)\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1} \tag{85}$$

so that by the end of the calculation we have computed $\frac{\mathrm{d}L}{\mathrm{d}u}$ for all variables $u \in \{y, z_2, z_1, \theta_1\}$.

3.2.1 Dual Numbers for Forward Mode AD

A useful representation for computing forward mode automatic gradients on-the-fly is the **dual number** representation. Here each variable v is replaced with $v + \Delta v$, where Δv represents an infinitesimal change in v. As such, quantity Δv has the property that $(\Delta v)^2 = 0$. Setting $\Delta u = 1$ for an independent variable u and then computing all dependent variables will result in $\Delta v = \frac{\mathrm{d}v}{\mathrm{d}u}$. We illustrate this with three examples.

[8] One drawback of this convention is that the gradients propagated through deep learning networks are with respect to a scalar-valued loss function L and, in all frameworks, stored transposed so that they have the same dimensionality/shape as the variable with respect to which the derivative is taken.

Example 1. In this first example we show how to compute gradients with dual numbers for the affine transformation $y = ax + b$,

$$y + \Delta y = a(x + \Delta x) + b \tag{86}$$
$$\therefore\ \Delta y = a\Delta x \tag{87}$$

which for $\Delta x = 1$ gives the expected result $\frac{\mathrm{d}y}{\mathrm{d}x} = a$.

Example 2. Next we look at the more complicated example of $y = \exp(x)$. Here we first expand the exponential function into its infinite polynomial series, $\exp(x) = \prod_{n=0}^{\infty} \frac{x^n}{n!}$, so that we can write,

$$y + \Delta y = \exp(x + \Delta x) \tag{88}$$
$$= 1 + (x + \Delta x) + \frac{(x + \Delta x)^2}{2} + \frac{(x + \Delta x)^3}{3!} + \ldots \tag{89}$$
$$= 1 + x + \Delta x + \frac{x^2 + 2x\Delta x + \Delta x^2}{2} + \frac{x^3 + 3x^2\Delta x + 3x\Delta x^2 + \Delta x^3}{3!} + \ldots \tag{90}$$
$$= 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots + \Delta x \left(1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots\right) \tag{91}$$
$$= \exp(x) + \Delta x \exp(x) \tag{92}$$
$$\therefore\ \Delta y = \Delta x \exp(x) \tag{93}$$

where in Line 91 we have grouped together terms involving Δx (all higher-order terms of Δx having been dropped using the property that $\Delta x^2 = 0$), and then in the following line replaced the infinite series with the corresponding $\exp(x)$ terms. Once again setting $\Delta x = 1$ we arrive at $\frac{\mathrm{d}y}{\mathrm{d}x} = \exp(x)$ as expected.

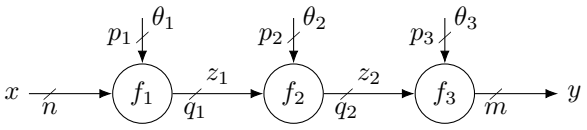
Example 3. Last, let us compute the gradient of $y = 1/x$,

$$y + \Delta y = \frac{1}{x + \Delta x} \tag{94}$$
$$= \frac{1}{(x + \Delta x)} \frac{(x - \Delta x)}{(x - \Delta x)} \qquad (\text{since } \tfrac{(x-\Delta x)}{(x-\Delta x)} = 1) \tag{95}$$
$$= \frac{x - \Delta x}{x^2 - \Delta x^2} \tag{96}$$
$$= \frac{x - \Delta x}{x^2} \qquad (\text{since } \Delta x^2 = 0) \tag{97}$$
$$\therefore\ \Delta y = -\frac{\Delta x}{x^2} \tag{98}$$

giving $\frac{\mathrm{d}y}{\mathrm{d}x} = -1/x^2$. These three examples are all we need for computing all the gradients we need for MLP training.

3.2.2 Cost of Gradient Evaluation Ordering

In training deep learning models we usually want to update all parameters at the same time. Consider the simple three-node deep learning model from earlier, reproduced below with the size of the intermediate variables annotated,



and with parameters $\theta = (\theta_1, \theta_2, \theta_3)$ of total size $p = p_1 + p_2 + p_3$. Both forward and reverse mode automatic differentiation compute the quantity,

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \begin{bmatrix} \frac{\mathrm{d}L}{\mathrm{d}\theta_1} & \frac{\mathrm{d}L}{\mathrm{d}\theta_2} & \frac{\mathrm{d}L}{\mathrm{d}\theta_3} \end{bmatrix} \in \mathbb{R}^{1 \times p} \tag{99}$$

Focusing on the first term we have,

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m} \underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2} \underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1} \underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta}}_{q_1\times p_1} \tag{100}$$

where we have written the size of each gradient matrix under it in the expression.

Forward mode automatic differentiation evaluates the expression from the right, first computing $\frac{\mathrm{d}z_2}{\mathrm{d}\theta_1} = \frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}$ of size q_2-by-p_1, then $\frac{\mathrm{d}y}{\mathrm{d}\theta_1} = \frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}\theta_1}$ of size m-by-p_1, and finally $\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\theta_1}$ of size 1-by-p_1. The total computational cost is therefore,

$$O(q_2 q_1 p_1) + O(m q_2 p_1) + O(m p_1) \tag{101}$$

Reverse mode automatic differentiation on the other hand evaluates the expression from the left, first computing $\frac{\mathrm{d}L}{\mathrm{d}z_2} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z_2}$ of size 1-by-q_2, then $\frac{\mathrm{d}L}{\mathrm{d}z_1} = \frac{\mathrm{d}L}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1}$ of size 1-by-q_1, and finally $\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}$ of size 1-by-p_1. The total computation cost for reverse mode is therefore,

$$O(m q_2) + O(q_2 q_1) + O(q_1 p_1) \tag{102}$$

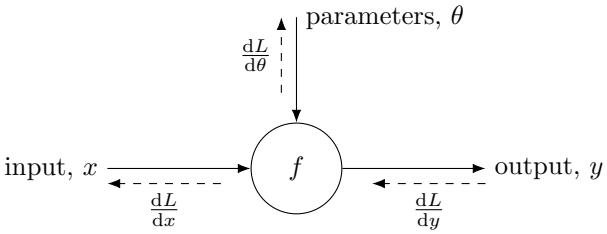
which can be significantly faster than the forward mode.

The operations for computing intermediate terms during reverse mode automatic differentiation are often called vector-Jacobian products for obvious reasons—we are always multiplying the gradient of the loss, which is a vector, by a so-called Jacobian matrix. This is generally much more efficient in deep learning where the function we are differentiating, i.e., L, is scalar-valued and there are a large number of parameters θ.

3.3 Back Propagation

Back-propagation is the term used in deep learning for computing gradients, essentially using reverse mode automatic differentiation together with caching of intermediate results. Different deep learning frameworks use slightly different approaches (explicit graph construction versus eager evaluation and operator tracking). But in all frameworks the developer only needs to implement the forward pass operation for a node and the backward pass is automatically generated. Conceptually, for each line of the forward pass code, `P, Q = foo(A, B, C)`, automatic differentiation needs to produce a line `dLdA, dLdB, dLdC = foo_vjp(dLdP, dLdQ)` in the backward pass code. Here `vjp` stands for vector-Jacobian product. Usually the inputs `A`, `B` and `C`, and outputs `P` and `Q` are also made available to `foo_vjp` in the backward pass.

So, to summarize, a deep learning node (sometimes called a **layer**) in a deep learning network,



needs to implement two operations. In the **forward pass** the node computes the output y as a function of the input x and parameters θ. In the **backward pass** the node computes the derivative of the loss function with respect to the input x and the parameters θ given the derivative of the loss with respect to the output y. In PyTorch these operations appear as follows:

```
y = f(x, theta)      # evaluate function
L = loss(y, y_true) # compute loss
```

```
L.backward() # compute all gradients
```

Let us look at a simple node that we may want to include within a deep learning model, that of computing the inverse square-root of a positive number x,

$$y = \frac{1}{\sqrt{x}} \tag{103}$$

This has application, for example, in 3D scene rendering where x represents the magnitude of a normal vector, and the aim is to produce a unit normal. The calculation in the forward pass is composed of elementary operations, namely square-root and inverse, and so can be automatically differentiated. Alternatively, we could provide a hand-coded implementation of the derivative. We have,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\frac{1}{2}x^{-3/2} \tag{104}$$

$$= -\frac{1}{2}y^3 \tag{105}$$

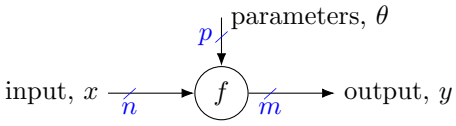
So in the backward pass, given $\mathrm{d}L/\mathrm{d}y$, the node would compute,

$$\frac{\mathrm{d}L}{\mathrm{d}x} = -\frac{1}{2}y^3\frac{\mathrm{d}L}{\mathrm{d}y} \tag{106}$$

Notice that we have chosen to use the output of the node, y, in the implementation of the expression for the gradient in the backward pass. This requires storing the value in the forward pass for re-use later on, and is a typical pattern in deep learning frameworks. In this example, we could also have implemented the backward pass in terms of x, but that would have required taking its square-root a second time.

3.3.1 Batched Operations

Consider an abstract deep learning node with n-dimensional input x, m-dimensional output y, and p-dimensional parameters θ, shown here with the shape of inputs, output, and parameters annotated,



The Jacobian matrices $\frac{\mathrm{d}y}{\mathrm{d}x}$ and $\frac{\mathrm{d}y}{\mathrm{d}\theta}$ are (m-by-n)-dimensional and (m-by-p)-dimensional matrices of partial derivatives y with respect to x and θ, respectively. The gradients $\frac{\mathrm{d}L}{\mathrm{d}y}$, $\frac{\mathrm{d}L}{\mathrm{d}x}$ and $\frac{\mathrm{d}L}{\mathrm{d}p}$ are a m-, n-, and p-dimensional (row) vectors, respectively. Now recall that in deep learning we process data in batches of, say, N samples. As such, the gradient quantities computed by the node in the backward pass are

$$\underbrace{\frac{\mathrm{d}L}{\mathrm{d}x^{(i)}}}_{1\times n} = \overbrace{\frac{\mathrm{d}L}{\mathrm{d}y^{(i)}}}^{1\times m}\underbrace{\frac{\mathrm{d}y^{(i)}}{\mathrm{d}x^{(i)}}}_{m\times n} \qquad \text{for } i = 1, \ldots, N \tag{107}$$

and

$$\underbrace{\frac{\mathrm{d}L}{\mathrm{d}\theta}}_{1\times p} = \sum_{i=1}^{N}\overbrace{\frac{\mathrm{d}L}{\mathrm{d}y^{(i)}}}^{1\times m}\underbrace{\frac{\mathrm{d}y^{(i)}}{\mathrm{d}\theta}}_{m\times p} \tag{108}$$

where the shape of each term has been marked. Notice that for the input we have N independent vector-Jacobian products, whereas for the parameters we sum the vector-Jacobian products over training instances. In practice, instead of forming $\frac{\mathrm{d}y}{\mathrm{d}x}$ explicitly, code can compute $\frac{\mathrm{d}L}{\mathrm{d}x}$ directly from $\frac{\mathrm{d}L}{\mathrm{d}y}$ if more efficient to do so.

3.3.2 Vanishing and Exploding Gradients

Training deep learning models involves evaluating long chains of gradient multiplications, e.g.,

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta} \tag{109}$$

This can result in the so-called vanishing and exploding gradient problem, i.e., successive multiplication of small numbers making the gradient go to zero, or successive multiplication of large numbers making the gradient calculation overflow.

The problem is particularly pronounced with sigmoid-like activation functions, which have flat regions, hence zero gradient, away from the origin. A great deal of effort has been devoted to developing techniques to mitigate against vanishing or exploding gradients. We will see some of these later in the lecture.

3.3.3 Memory Layout and Usage in Deep Learning

Deep learning models consume vast quantities of data to train. The data is often processed in batches to reduce computational load and memory requirements. A batch of data may have shape something like $B \times C \times H \times W$ for a batch of size B elements of C-channel 2D features maps. An example illustrating memory layout as a batch of data is processed through a model is shown in Figure 22. Blue boxes indicate tensors in the forward pass, gray boxes indicate memory that stores model parameters, and red boxes depict gradient tensors in the backward pass. The batch size stays constant through the network but the dimensionality of the rest of the tensor may change. Parameters are not batched, they are shared across batch elements.

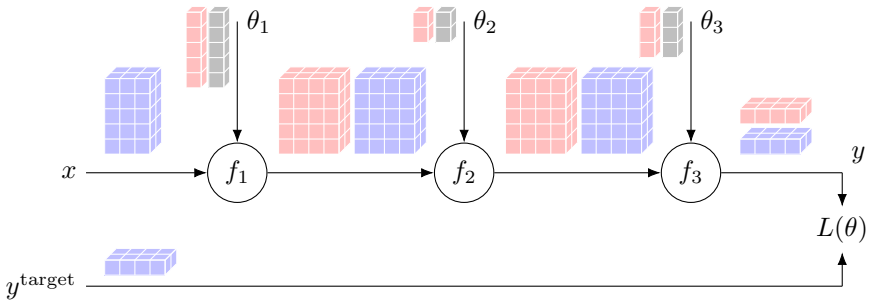


Figure 22: Illustration of memory requirements and layout in a deep learning model.

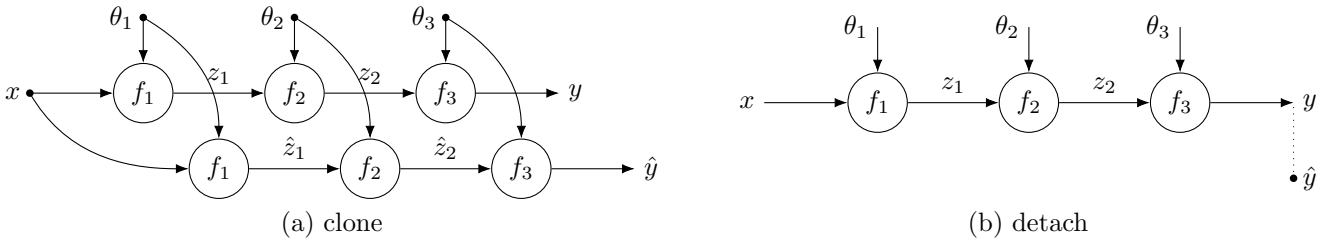
There are a few of things to observe. First, parameters only take a small amount of memory (relative to data).[9] Second, gradients take the same amount of space as the features from the forward pass. Gradients are stored transposed so that their shape exactly matches the forward pass tensor. Third, in-place operations may save memory in the forward pass. However, during training intermediate results from the forward pass may be needed to compute gradients. In-place operations must not destroy this information. Re-using memory buffers may save some memory in the backward pass, but this can be challenging to implement and of minor utility since feature size (and hence memory requirements) usually dominate at the early layers of the network. Last, at test time only the forward pass is needed and hence intermediate results are not stored, significantly reducing the memory needed. This is why we need large data centers to train deep learning models, but the final model can be deployed on edge devices for many applications.

3.3.4 PyTorch Autograd Function

Mostly you will be able to compose models from existing functions already implemented in your deep learning software library. However, there may be times when you will want to add functionality beyond what is available in the library or implement your own variants for speed or memory efficiency. Below we show an example PyTorch class for an automatically differentiable function that implements the inverse square-root operation from earlier. The class contains two important methods. The first method, `forward`, is called in the forward pass and computes $y = 1/\sqrt{x}$. The second method, `backward`, is called during the backward pass and computes $\mathrm{d}L/\mathrm{d}x = -\frac{1}{2}y^3\mathrm{d}L/\mathrm{d}y$. Scalar input (and output) is assumed in this example, and some housekeeping code is included to pass context information between the forward and backward pass and check whether gradients are actually needed.

```
class InvSqrtFcn(torch.autograd.Function):
    """Differentiable inverse square-root."""

    @staticmethod
    def forward(ctx, x):
        with torch.no_grad():
            y = 1.0 / torch.sqrt(x)

        # save state for backward pass
        ctx.save_for_backward(y)

        # return result
        return y

    @staticmethod
    def backward(ctx, dLdy):
        # check for None tensors
        if dLdy is None:
            return None

        # unpack cached tensors
        y = ctx.saved_tensors

        # compute and return gradients
        dLdx = None
        if ctx.needs_input_grad[0]:
            dLdx = -0.5 * torch.pow(y, 3.0) * dLdy
        return dLdx
```

[9] There are some very large foundation models these days where the number of parameters can be significant. But even for these models the amount of data needed to train the model is massive compared to the parameter count.



(a) clone (b) detach

Figure 23: Graphical illustration of clone and detach operations.

Of course, this is a trivial example and the code `y = 1.0 / torch.sqrt(x)` can be automatically differentiated by PyTorch already without the need to implement our own `autograd.Function`. However, the example gives you a template for more sophisticated operations where PyTorch may not know how to compute gradients, e.g., in deep declarative networks [35].

3.3.5 Cloning and Detaching

There may be times when we want to copy a variable in an implementation of a deep learning model. This is rarely something we think much about when coding for the forward pass[10], but further processing of copied variables has big implications on what gradients get calculated in the backward pass. There are two general mechanisms by which copying can be done. The first, **cloning**, creates a copy of the entire computation graph. This means that gradients will propagate backwards through both the original variable and the cloned variable. Cloning is performed using the `clone()` method on tensors,

```
y_hat = y.clone() # creates copy of computation graph, gradients backpropagate
```

The second approach is to **detach** a variable. This creates a new variable that shares memory with the original tensor. However, no gradients are propagated through the new detached variable, and therefore any further processing on this variable has no effect on training (assuming the underlying shared memory is not modified, i.e., no in-place operations). Detaching is performed using the `detach()` method,

```
y_hat = y.detach() # shares memory with y but no gradients backpropagate
```

It is possible perform both detach and clone, as in

```
y_hat = y.detach().clone() # creates a copy of y but no gradients backpropagate
```

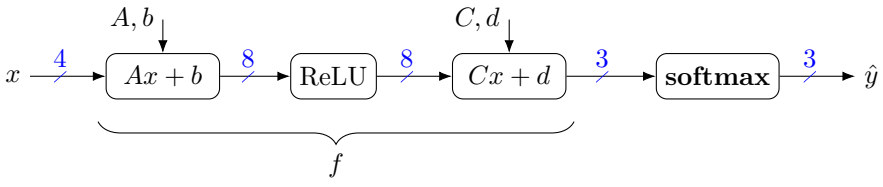
which creates a copy of the original variable without the computation graph and without sharing memory.

An illustration of the clone and detach operations is shown in Figure 23.

3.4 Putting it all Together

Let's put what we have learned so far together to build an flower classifier. We won't be extracting features directly from images but rather making use of an existing dataset where features of example flowers have already been measured for us manually. The dataset is the Iris dataset collected by the famous statistician Sir Roland Fisher in 1936. It contains 150 examples of Iris flowers. The flowers are divided into three classes—Setosa, Versicolour, and Virginica—with 50 examples per class. Each example is characterized by a 4-dimensional feature vector representing the flower's sepal length, sepal width, petal length and petal width. Examples of the different classes is shown in Figure 24.

The existence of the pre-measured features significantly simplifies the problem. Nevertheless, it will make for an instructive case study based on what we have learned thus far in the course. To classify examples we will train a multi-layer perceptron with four inputs, eight hidden nodes with ReLU activation function, and three outputs normalized by softmax. The model architecture, which should be quite familiar to you by now, is shown below.



[10] Beyond the cost of additional memory that may be needed to store the copied data.

Figure 24: The Iris dataset (Fisher, 1936) is a small classical dataset used to develop and evaluate classification algorithms. It consists of 150 examples of three different classes of Iris flower—Setosa, Versicolour, and Virginica. Each instances has four features measuring properties of the flower.

The parameters of the model, $\theta = \{A, b, C, d\}$, will be tuned using gradient descent with cross-entropy loss,

$$\ell(\theta; x, y) = -\log\left[\mathbf{softmax}(f(x;\theta))\right]_y . \tag{110}$$

Here, $[\cdot]_y$ indicates taking the entry from $\mathbf{softmax}(f(x;\theta))$ corresponding to the ground-truth label y. In this case study we will use all 150 examples as our training data, so the loss L is summed over the entire dataset,

$$L(\theta) = \sum_{i=1}^{150} \ell(\theta; x^{(i)}, y^{(i)}) \tag{111}$$

with gradient updates,

$$\theta \leftarrow \theta - \eta \nabla L(\theta). \tag{112}$$

However, for real applications it is better to split the data into training, validation and test subsets, and only update the parameters based on the training subset. We will discuss the reason for doing this later in the lecture.

PyTorch's `nn.CrossEntropyLoss` function computes **softmax** internally so we can removed it from the software implementation of our model. This does not matter for prediction since taking the maximum element after the softmax is the same as taking the maximum element before softmax (i.e., on the logits).

There are two ways to implement our multi-layer perceptron in PyTorch. Since the model is a simple chain of processing nodes we can defined it using the `nn.Sequential` container. This container takes a list of nodes (also called modules in PyTorch) and connects the output of each node to the input of the next node in the sequence.

```
    model = nn.Sequential(
        nn.Linear(4, 8, bias=True),
        nn.ReLU(True),
        nn.Linear(8, 3, bias=True)
    )
```

The second way to implement our model is by specializing the `nn.Module` class. This is much more flexible in that arbitrary processing can be performed within the `forward` method. Here we simply compute the output of each layer in turn using as input the output from the previous layer. Layers are instantiated in the `__init__` method. Finally, on Line 18 we create an object of the multi-layer perceptron class.

```
    class IrisMLP(nn.Module):
        """Multi-layer perceptron for Iris Dataset."""

        def __init__(self):
            super().__init__()

            self.layer1 = nn.Linear(4, 8, bias=True)
            self.layer2 = nn.ReLU(True)
            self.layer3 = nn.Linear(8, 3, bias=True)

        def forward(self, x):
            z1 = self.layer1(x)
            z2 = self.layer2(z1)
            y_hat = self.layer3(z2)
```

```
15
16          return y_hat
17
18      model = IrisMLP()
```

In both ways of implementing the model, the trainable parameters are automatically created and initialized as part of the `nn.Linear` layers. But what values are the parameters initialized to? Remember that we wish to avoid vanishing and exploding gradients. Therefore it makes sense to initialize the parameters so that the activation functions are operating in their linear region. For the logistic function this is around zero.

One good approach is to randomly draw parameter values i.i.d. from a zero-centered Gaussian, $\mathcal{N}(0, \sigma^2)$, or symmetric uniform distribution, $\mathcal{U}(-\sigma, \sigma)$, where σ is small. The bias terms, b and d are typically set to zero. More sophisticated approaches consider the fan-in and fan-out of a node to try control the variance of the features as the propagate through the network. Two popular options are Golorot or Xavier initialization [29], which sets the Gaussian standard deviation as

$$\sigma = \sqrt{\frac{2}{n_i + n_o}} \tag{113}$$

where n_i and n_o are the number of inputs and number of outputs, respectively, and Kaiming initialization [38], which sets

$$\sigma = \sqrt{\frac{2}{n_i}} \quad \text{or} \quad \sigma = \sqrt{\frac{2}{n_o}} \tag{114}$$

to work better for models with ReLU activation functions.

Some tasks require more specialized initialization techniques to guide training towards particular solutions or away from degenerate ones. The spherical initialization for 3D shape fitting [10] is a good example of such techniques.

3.4.1 Training

Once we start training our model we want to keep track of how well the model is doing. **Learning curves** are a good way of doing this by plotting the training loss (or any other performance metric) as a function of training iteration. An example for training our Iris classification model is shown in Figure 25. Notice that the training loss decreases steadily with number of iterations. The second plot shows how misclassification or error rate changes as the model is trained. Randomly guessing would result in an error rate of $\frac{2}{3}$, since we have a balanced three-class problem. This is where the model starts when it is initialized. As the model trains the number of errors tends towards zero, but not necessarily in a monotonic way.

It can be mesmerizing to watch the learning curves as your model trains. And while learning curves are very helpful in diagnosing problems early, many a researcher has wasted hours and hours enjoying the peace and quiet of just staring at learning curves. This is something to be avoided.

You will often hear the term **learning dynamics** or optimization dynamics to describe the patterns and behaviours exposed by learning curves. Understanding these dynamics can sometimes help improve the design of learning algorithms as we will see throughout the remainder of this lecture.

Our task for Iris classification in this case study is small and the full training dataset can be easily processed to calculate the gradient at each iteration. However, for big models (e.g., ImageNet), using the entire dataset to calculate the gradient is very expensive. Recall that many loss functions decompose over training data $\{(x^{(i)}, y^{(i)})\}_{i=1}^N$, and hence so does the gradient. Thus far we have not been concerned with scaling factors, but in practice the loss function is normalized by the size of the training dataset N, making it somewhat invariant to the amount of training data,

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(f(x^{(i)}; \theta), y^{(i)}) \tag{115}$$

giving the gradient as

$$\nabla_\theta L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \, \ell(f(x^{(i)}; \theta), y^{(i)}) \tag{116}$$

Stochastic gradient descent (SGD) is a method for approximating gradients by using only a subset of the data, called a **mini-batch**, $\mathcal{I} \subseteq \{1, \ldots, N\}$, to get

$$\widehat{\nabla_\theta L} = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \nabla_\theta \, \ell(f(x^{(i)}; \theta), y^{(i)}) \tag{117}$$

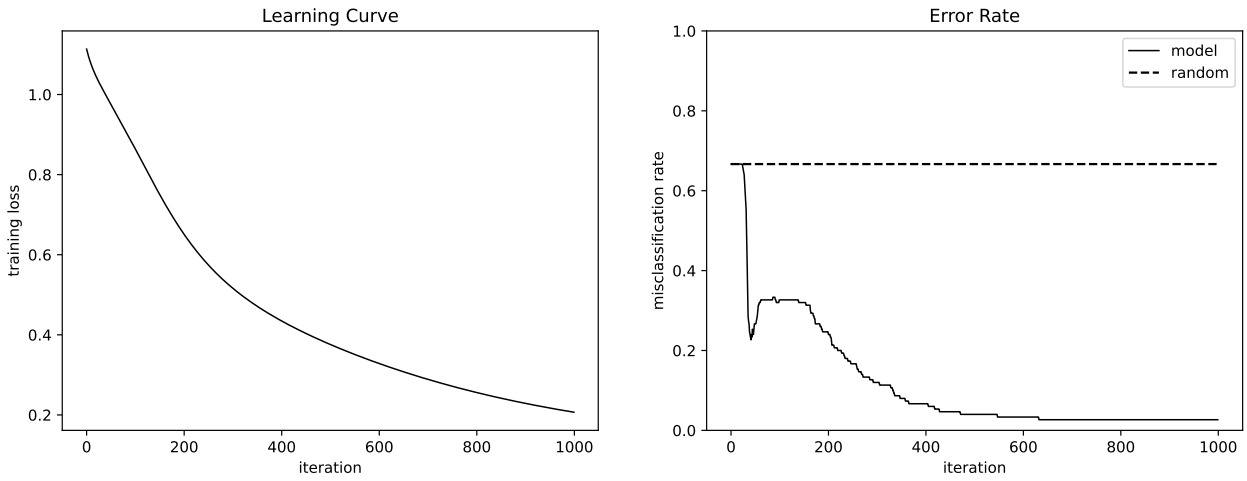


Figure 25: Learning curves plot performance metrics as a function of training iteration. They are very useful for diagnosing problems with your model or training algorithm.

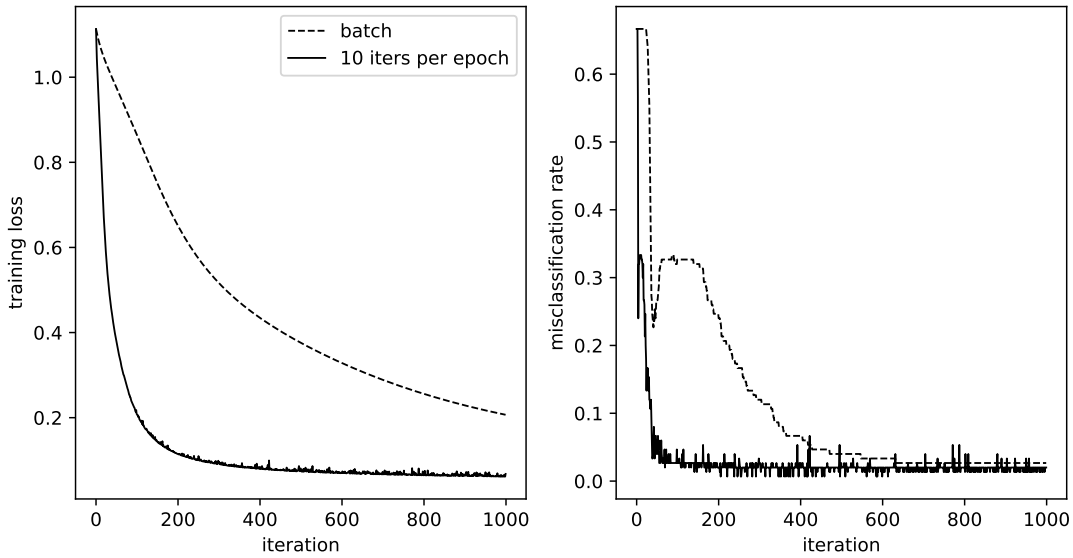


Figure 26: Learning curves comparing stochastic gradient descent with gradient descent.

The number of training examples in $\mathcal{I}$ is called the **batch size**.

Under mild assumptions the expected value of the approximate gradient from stochastic gradient descent will equal the true gradient, i.e., $E[\widehat{\nabla_\theta L}] = \nabla_\theta L$. This is good news because it means that over a large number of training iterations stochastic gradient descent will behave the same as gradient descent, and each iteration can be performed much, much faster. Gradients can also be accumulated over multiple mini-batches for a better estimate of the immediate gradient, which is sometimes done in a distributed fashion making use of parallel compute.

In practice we shuffle the indices of examples in the training dataset, i.e., we randomly permute $[N]$, and iterate through adjacent fixed-length intervals. Under this regime each gradient update step is called an **iteration**, and once through the dataset is called an **epoch**. The dataset is reshuffled at the start of each epoch to avoid biasing the model to perform well on just the last mini-batch in the shuffled dataset.

Figure 26 shows learning curves comparing stochastic gradient descent with gradient descent for our Iris classification problem. For the stochastic setting we choose a mini-batch size of 15, so it takes 10 iterations to complete an epoch. Notice that despite a noisy gradient estimate, here stochastic gradient descent converges much faster, although this is not always the case.

Deep learning models are generally non-convex functions so different parameter initializations and different mini-batch orderings may produce different results. Even different hardware and software libraries can cause differences in training, due to accumulation of small numerical differences or race conditions in the hardware. It is important when comparing methods that any perceived improvement is not a result of a lucky random seed favouring one method over

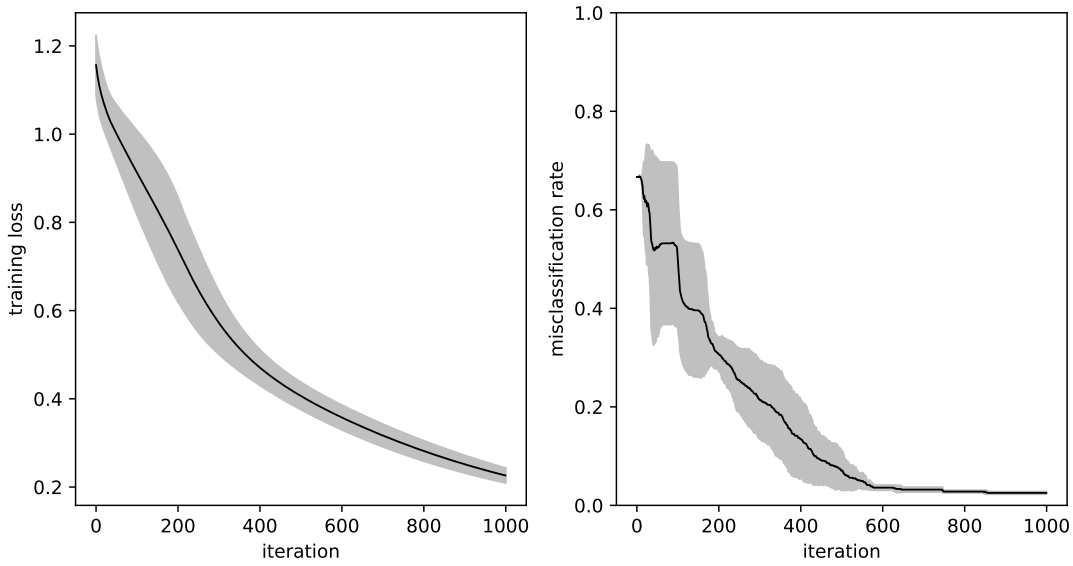


Figure 27: Different random parameter initializations can result in different learning curves. Shown is the mean and one standard deviation over five training runs, each with a different random seed.

another. Figure 27 shows the average learning curve over five training runs initialized with different random seeds. The shaded area shows one standard deviation of performance (measured at each iteration). In this simple example the five runs converge to the same final error rate.

One of the most important questions in deep learning is which model to deploy? Taking the model defined by the parameters from the last training iteration is not always the best since deep learning is prone to overfitting. Standard practice it to break the dataset into three disjoint subsets, each playing a different role. First, the **training set** is used for estimating gradients and updating the model parameters. Second, the **validation set** is used for selecting the model (i.e., set of parameters) to deploy. Every epoch or so during training the model parameters can be saved so that the model can be evaluated or have further diagnostics run later. This is known as **check-pointing**. The third subset, known as the **test set**, is used to evaluate generalization performance on the model chosen by the validation set. Ideally, the test set is never seen by model developers to avoid inadvertently using it to tweak model performance and therefore provide an invalid estimate of how it will perform on new data when deployed.

Figure 28 shows learning curves for the Iris classifier on a training subset and validation subset. Here we randomly shuffle the data and use the first half for training and second half for validation. That is, only 75 examples are used to estimate the gradient of the loss and update model parameters. The other 75 examples are used to track performance. Notice that as training proceeds the model does better on the training subset than the validation subset. This is a classic case of overfitting. In this example we would deploy the model parameters obtained at around 1000 training iterations rather than the model which achieves lowest training loss (at 5000 iterations).

There are several mechanisms that can be used to improve generalization performance. These include adding an explicit **regularization** term to the loss function, which is very standard in machine learning as we have already seen. We can also implicitly regularize the parameters through **data augmentation**. This is a technique that adds random noise/perturbations to the input data in an attempt to force the model to deal with slight variations of the training data that is may see at test time. Data augmentation is very standard in deep learning and built into the training pipelines of deep learning software frameworks. We'll be looking at some specific data augmentation approaches designed for computer vision applications in later lectures.

Another option is to collect more real training data. This is evident in the massive influence the large-scale ImageNet dataset had in solving the image classification problem. Unfortunately, this can also be very expensive and in some applications may not be possible (e.g., medical imaging or data from the surface of Mars).

The last mechanism, which has had surprising success in deep learning, is to use large-scale generic **pre-training**, on a dataset such as ImageNet, followed by **finetuning** on a smaller dataset curated for the specific task. Here finetuning refers to continued training of the model on the task-specific data, and can be thought of as a type of transfer learning.

The scale and distribution of features can have a big impact on training and model performance. We saw this in the example gradient descent optimization example in Figure 19 from the last lecture. Normalizing, also known as **whitening**, the data to have zero mean and unit covariance often improves convergence and generalizability as shown

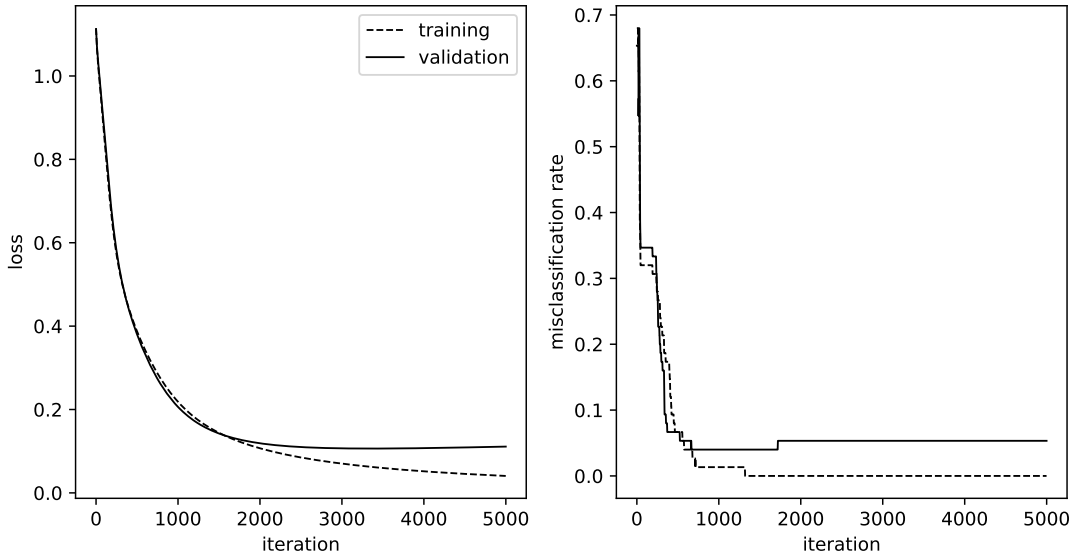


Figure 28: Training set versus validation set performance. Here we randomly shuffled the Iris dataset, and used the first half (75 examples) for training and second half (75 examples) for validation.

in Figure 29. Mathematically we have,

$$x^{(i)} \leftarrow \Sigma^{-1}(x^{(i)} - \mu), \qquad \text{for } i = 1, \ldots, N \tag{118}$$

where μ and Σ are the feature mean and covariance over the training dataset.

On large-scale data it is impractical to compute the full covariance matrix Σ so we typically normalize each dimension of the data independently,

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j}, \qquad \text{for } j = 1, \ldots, n \tag{119}$$

where μ_j is the mean of the j-th feature over the training data, and σ_j is its standard deviation. When applied to features within a deep learning model from online estimates of the mean and standard deviation, this is called **batch normalization** (or BatchNorm) and will be discussed in later lectures.

The learning rate η is an example of a **hyperparameter** and its value can also have a big influence on performance and convergence rate. Figure 30 shows learning curves for training our Iris classification model with different learning rates. As evident in the figure, too small a learning rate results in very slow convergence, whereas too large a learning rate can lead to unstable training and can even get the model stuck.

We do not need to use the same learning rate each epoch, and experience has shown that varying the learning rate can have a significant effect on training. A rule for determining the learning rate at each epoch is called a **learning rate schedule**. So far we have only considered a constant learning rate, $\eta^{(t)} = \eta_0$. Other popular learning rate schedules include the linear schedule,

$$\eta^{(t)} = \eta_{\text{init}} + \frac{t}{t_{\max}} \left(\eta_{\text{final}} - \eta_{\text{init}}\right), \tag{120}$$

the step schedule,

$$\eta^{(t)} = \begin{cases} \gamma \eta^{(t-1)}, & \text{if } t \,\%\, t_{\text{step}} = 0 \\ \eta^{(t-1)}, & \text{otherwise} \end{cases} \tag{121}$$

and the cosine schedule,

$$\eta^{(t)} = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left(1 + \cos\left(\frac{\pi t}{t_{\text{period}}}\right)\right) \tag{122}$$

Combinations of learning rate schedules can also be used, e.g., linear warm-up with cosine decay. Figure 31 shows this combination as well as the step learning rate schedule. A jump (rapid decrease) in training loss often accompanies a step change in the learning rate.

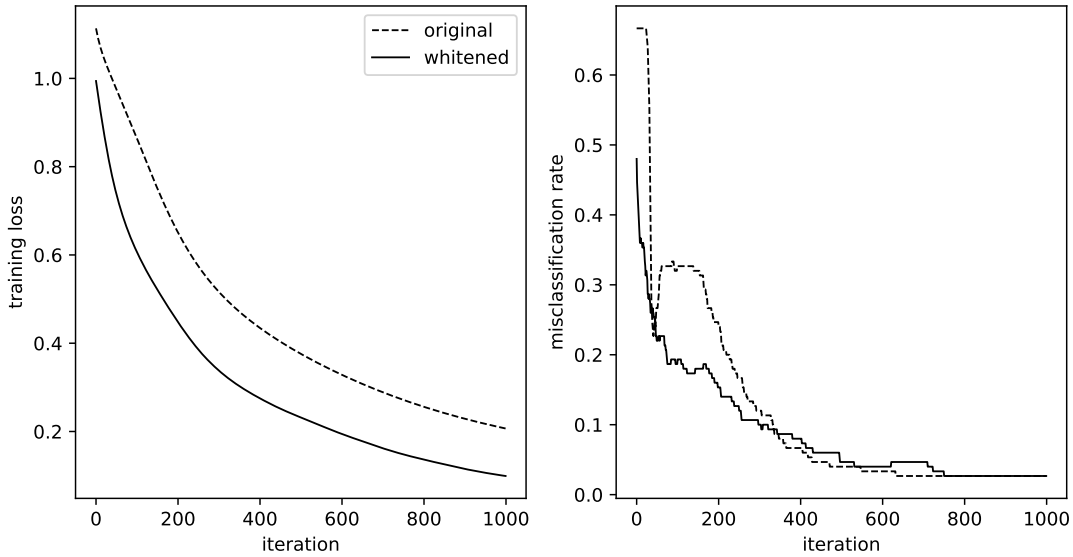


Figure 29: Normalizing data, also called whitening, often improves training convergence and stability. The learning rate here has been rescaled based on the average feature magnitude.

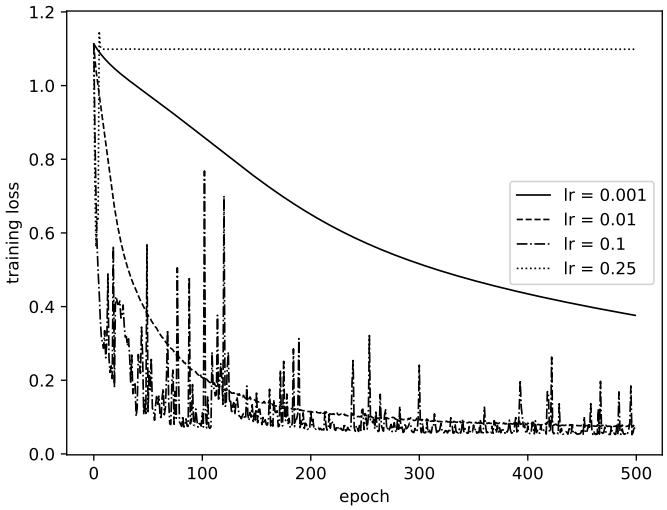


Figure 30: Learning rate, η, can have a significant effect on training convergence. Too small a learning rate and convergence can be very slow, too high a learning rate results in unstable training, which can sometimes get the model stuck.

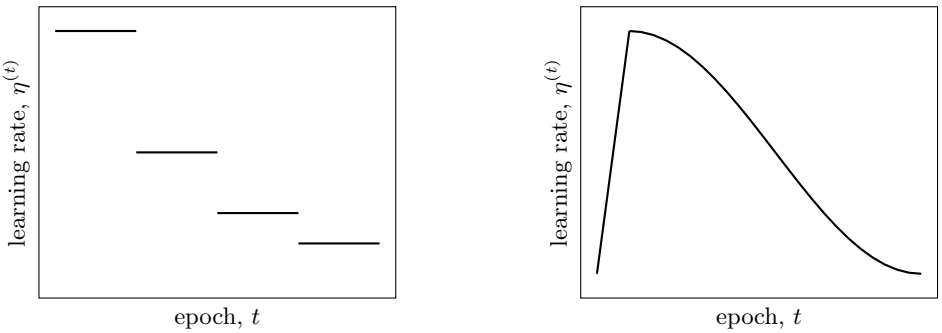


Figure 31: Learning rate schedules, $\eta^{(t)}$: (left) step learning rate, (right) linear warm-up followed by cosine decay.

Stochastic gradient descent can still be slow even with a good learning rate schedule and whitening of the data. This prompted researchers to explore alternative pseudo-second-order schemes to improve convergence rate. One popular method is to add **momentum** (also called the heavy ball method),

$$g^{(t)} = \nabla L(\theta^{(t)}) + \mu g^{(t-1)} \tag{123}$$

$$\theta^{(t+1)} = \theta^{(t)} - \eta g^{(t)} \tag{124}$$

which adds a multiple of the previous gradient to the current gradient estimate. This tends to smooth out any noise in the stochastic gradient estimate and keep the parameters moving in the same direction much like a heavy ball rolling down a bumpy road.

Other popular methods include AdaGrad [23], Adam [48] and AdamW [65], all of which we will discuss shortly. Each method comes with a set of hyper-parameters and you need to play around to see what works best for your problem, using the learning curves as a guide. A good strategy is to start with a method and schedule that had been demonstrated to work well for similar tasks or network architectures, and then adapt the various hyper-parameters in a process known as **hyper-parameter tuning**. An active area of research is exploring how to set the hyper-parameters automatically.

AdaGrad [23] is a method inspired by classical optimization algorithms. It views gradient descent as updating the parameters according to a first-order proximal method that solves the optimization problem,

$$\theta^{(t+1)} = \arg\min_{\theta} \left\{ \langle \nabla L(\theta^{(t)}), \theta \rangle + \frac{1}{2\eta_t} \|\theta - \theta^{(t)}\|_2^2 \right\} \tag{125}$$

whose solution, obtained by differentiating the objective and setting to zero, is the standard gradient update,

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla L(\theta^{(t)}) \tag{126}$$

As we've seen, however, this can lead to slow convergence because of the geometry of the loss landscape requiring different features to be scaled differently. The trick with AdaGrad is to adapt the step size (geometry) for different features to increase influence of rare but informative features by changing the norm of the proximal term in the above formulation,

$$\theta^{(t+1)} = \arg\min_{\theta} \left\{ \langle \nabla L(\theta^{(t)}), \theta \rangle + \frac{1}{2\eta_t} \|\theta - \theta^{(t)}\|_B^2 \right\} \tag{127}$$

giving

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t B^{-1} \nabla L(\theta^{(t)}) \tag{128}$$

The matrix B encodes the curvature of the loss landscape. In the classical Newton's method it would be the Hessian matrix, $\nabla^2 L(\theta^{(t)})$. However, this is too expensive to compute for large-scale problems being quadratic in the number of parameters. AdaGrad estimates B as a diagonal matrix using previous gradients,

$$G^{(t)} = \left(\sum_{k=1}^{t} \mathbf{diag}\big(\nabla L(\theta^{(k)})\big)^2 + \epsilon I \right)^{1/2} \tag{129}$$

which is only linear in the number of parameters and easily invertible. The main weakness of AdaGrad is that elements of the matrix $G^{(t)}$ keep growing so the learning rate becomes infinitesimally small over time.

Adam [48] fixes the above problem with AdaGrad by maintaining exponentially decaying averages of past gradients and gradients-squared,

$$m^{(t)} = \beta_1 m^{(t-1)} + (1 - \beta_1) \nabla L(\theta^{(t)}) \tag{130}$$

$$v^{(t)} = \beta_2 v^{(t-1)} + (1 - \beta_2) \nabla L(\theta^{(t)})^2 \tag{131}$$

To ensure an unbiased estimate (c.f. unbiased variance calculations) it modifies these decaying averages as,

$$\hat{m}^{(t)} = \frac{1}{1 - \beta_1^t} m^{(t)} \text{ and } \hat{v}^{(t)} = \frac{1}{1 - \beta_2^t} v^{(t)}$$

The update rule according to the Adam method is then,

$$\theta^{(t+1)} = \theta^{(t)} - \eta \, \mathbf{diag}\Big(\hat{v}^{(t)} + \epsilon\Big)^{-1/2} \hat{m}^{(t)} \tag{132}$$

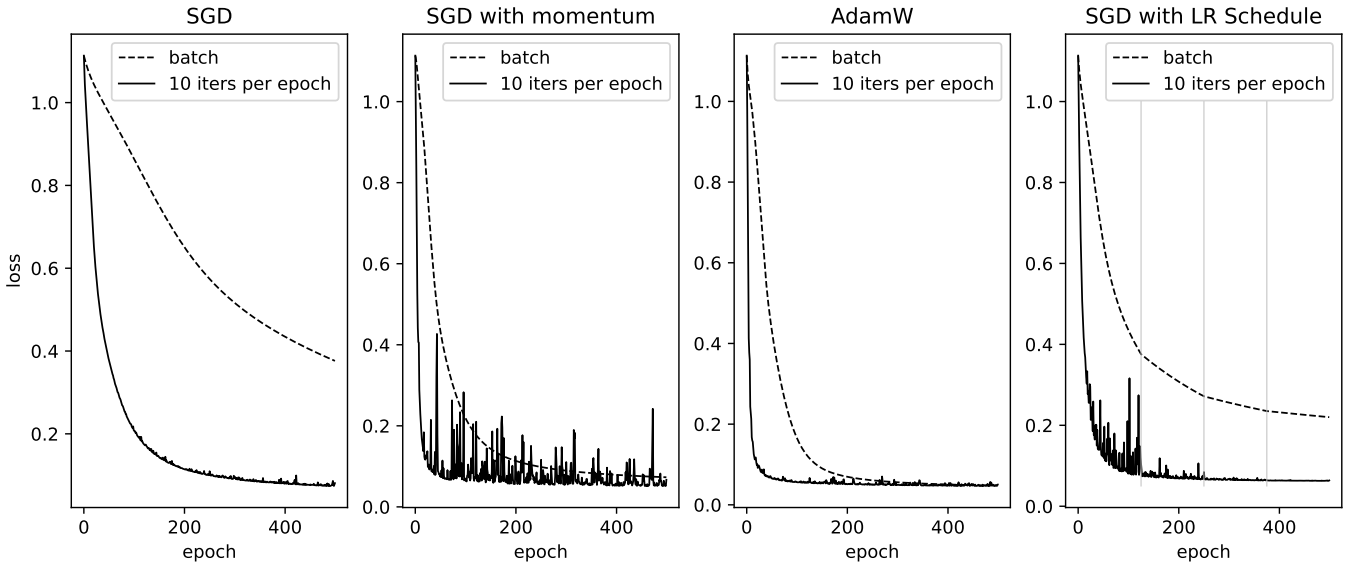


Figure 32: Comparison of learning curves for different variants of gradient descent and stochastic gradient descent.

AdamW [65] goes one step further and includes weight decay within the Adam update,

$$\theta^{(t+1)} = \theta^{(t)} - \eta \, \mathbf{diag}\Big(\hat{v}^{(t)} + \epsilon\Big)^{-1/2} \Big(\hat{m}^{(t)} + w\theta^{(t)}\Big) \tag{133}$$

This has been shown empirically to work better than applying ℓ_2 regularization on $m^{(t)}$ separately. AdamW is the de facto standard update rule (for now), although there are many other variants and combinations.

A comparison of learning curves for vanilla stochastic gradient descent (SGD), momentum, AdamW and the step learning rate schedule are show in Figure 32. SGD, momentum and AdamW use the same fixed step size whereas the step learning rate schedule starts higher and drops to lower during training. Observe that momentum accelerates convergence but can also be unstable. AdamW is overall the best for this problem.

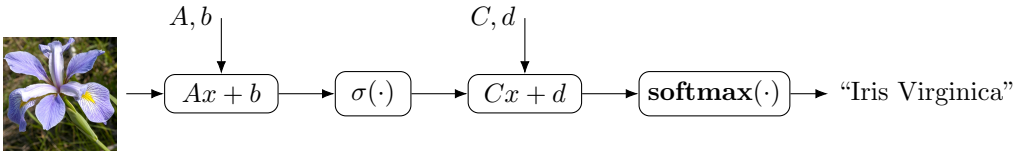


Figure 33: Hypothetical image classifier using a two-layer perceptron architecture.

4 Convolutional Neural Networks

Going beyond multi-layer perceptrons, this lecture introduces convolutional neural networks (CNNs) for image classification. We present the core mathematical building blocks and their justification. The second half of the lecture is devoted to the discussion of evaluating models, training variations, and popular CNN architectures.

Before introducing convolutional neural networks, let us consider building a hypothetical image classifier, i.e., a model that takes an image as input and produces a label for the image as output. We could, for example, try to classify images of iris flowers into the three categories, Setosa, Versicolour, and Virginica, like we did in the last lecture. However, unlike the classifier in that lecture, which used four manually measured features, here we want to use the raw image pixels as features directly. We can think about constructing our classifier from a multi-layer perceptron acting on a vectorized version of the image. An schematic is shown in Figure 33.

Let us assume that the image is 224-by-224 pixels and has three colours. The vectorized input size is therefore,

$$n = 224 \times 224 \times 3 \tag{134}$$
$$= 150,528 \tag{135}$$

For the sake of argument, let's have a 1000-dimensional hidden state and ten output categories (rather than just three iris types). The parameters of the multi-layer perceptron parameters depicted in Figure 33 then will have sizes

$$A \in \mathbb{R}^{150528 \times 1000}, \quad b \in \mathbb{R}^{1000}, \quad C \in \mathbb{R}^{1000 \times 10}, \quad d \in \mathbb{R}^{10} \tag{136}$$

giving a total of 150,539,010 parameters. Assuming that we use a standard 32-bit floating-point representation (i.e., 4 bytes per parameter), that amounts to 602,156,040 bytes (around 0.5GB) just to store the parameters!

Not only is that an enormous number of parameters to store, but training a model with so many parameters is computationally difficult and would require a huge amount of (labeled) data. Convolutional neural networks provide a much more efficient approach, and also address other issues to do with robustness as we will see later in the lecture.

4.1 Convolution

A **convolution** is a mathematical operation applied to two functions (or signals) that produces a third function, and is typically used to describe filtering of a signal. It can be applied to functions of any shape. We will start with the one-dimensional case. Let $x \in \mathbb{R}^n$ be an input signal and $a \in \mathbb{R}^p$ be a filter (or **kernel**). We write the convolution operation as[11]

$$y = a * x \qquad (\in \mathbb{R}^{n-p+1}) \tag{137}$$
$$y_i = \sum_{j=1}^{p} a_j x_{i+j-1} \qquad \text{for } i = 1, \ldots, n-p+1 \tag{138}$$

A worked example for a one-dimensional convolution operation is shown in Figure 34. Notice that the output signal is shorter than the input signal. Specifically, for input signal of length n and kernel of length p, the output signal will have length $n - p + 1$. We will address a strategy of how this can be avoided later in the lecture.

Instead of writing the convolution operation as an explicit summation, we can write it as a matrix-vector multiplication,

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-p+1} \end{bmatrix} = \begin{bmatrix} a_1 & \ldots & a_p & 0 & \ldots & 0 \\ 0 & a_1 & \ldots & a_p & \ldots & 0 \\ \vdots & & \ddots & & \ddots & \vdots \\ 0 & \ldots & 0 & a_1 & \ldots & a_p \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{139}$$

[11] In the signal processing literature a is reversed before summing. But since in deep learning we learn the coefficients of the kernel it is more standard to use the definition provided here.

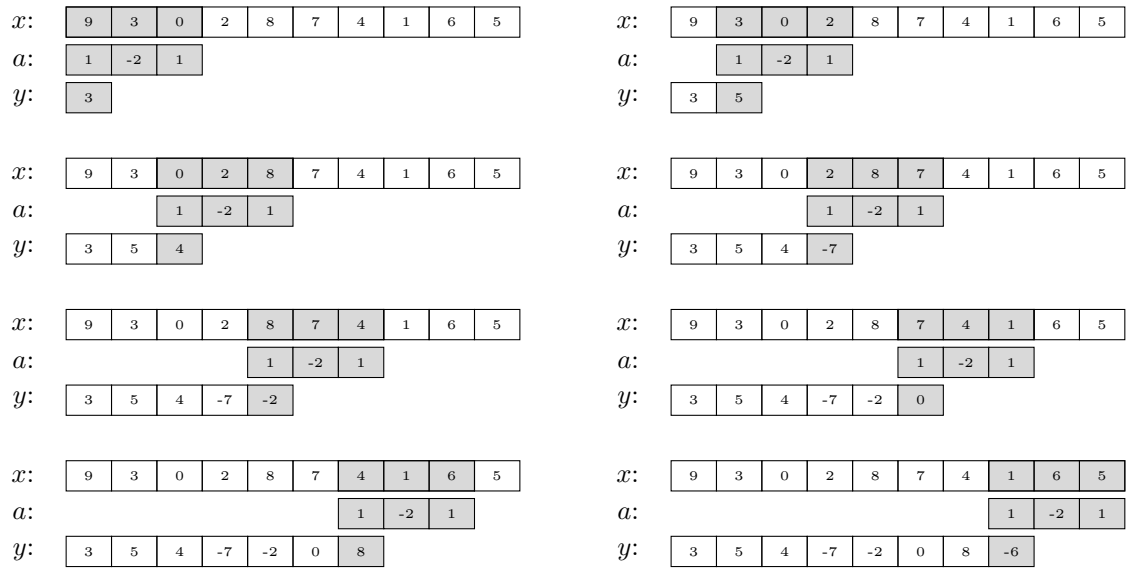


Figure 34: Worked example of a one-dimensional convolution operation, $y = a * x$.

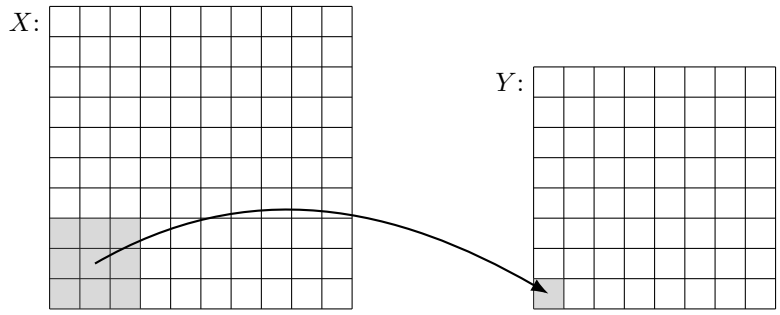


Figure 35: Two-dimensional convolution, $Y = A * X$, shifts the kernel over the plane.

or in a more compact form,

$$y = Ax \tag{140}$$

Here the matrix A, formed by rows containing shifted version of a^T, is called a Toeplitz matrix. Viewing convolution in this way we see that convolution is a type of linear operator with weight sharing (i.e., sharing of parameters).

Convolutions on two-dimensional signals (or feature maps) follow much the same pattern as one-dimensional convolutions. Let $X \in \mathbb{R}^{n \times m}$ be an input signal and $A \in \mathbb{R}^{p \times q}$ be a filter. Then

$$Y_{ij} = \sum_{k=1}^{p} \sum_{\ell=1}^{q} A_{k,\ell} X_{i+k-1, j+\ell-1} \tag{141}$$

for $i = 1, \ldots, n - p + 1$ and $j = 1, \ldots, m - q + 1$. Note that here the kernel is also two-dimensional and not only slides left-to-right over the signal, but also top-to-bottom, i.e., the kernel shifts over the plane. An illustration of two-dimensional convolution is shown in Figure 35. Once again, the output signal is smaller than the input signal.

Now, recall from the first lecture that a feature map can have multiple channels. For example, an n-by-m colour image has three channels, $X_R, X_G, X_B \in \mathbb{R}^{n \times m}$, for red, green and blue colour information, respectively.

We can produce a feature map $Y \in \mathbb{R}^{(n-p+1) \times (m-q+1)}$ by convolving each channel with a filter A_R, A_G and A_B, and summing the output,

$$Y = A_R * X_R + A_G * X_G + A_B * X_B \tag{142}$$

as depicted in Figure 36.

This is equivalent to stacking the channels into a 3-tensor and convolving with a 3D kernel. Convolutions on images and feature maps are defined as such. Let $X \in \mathbb{R}^{n \times m \times d}$ be an input signal and $A \in \mathbb{R}^{p \times q \times d}$ be a filter. Then

$$Y_{ij} = \sum_{p'=1}^{p} \sum_{q'=1}^{q} \sum_{d'=1}^{d} A_{p',q',d'} X_{i+p'-1, j+q'-1, d'} \tag{143}$$

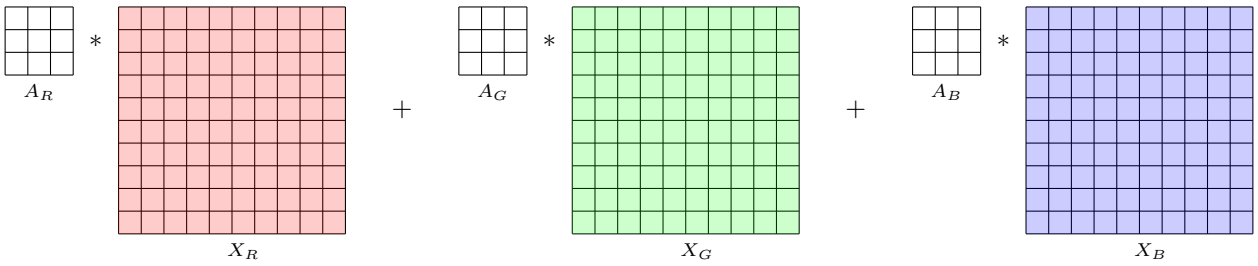


Figure 36: Two-dimensional convolution over multiple channels, $Y = A_R * X_R + A_G * X_G + A_B * X_B$.

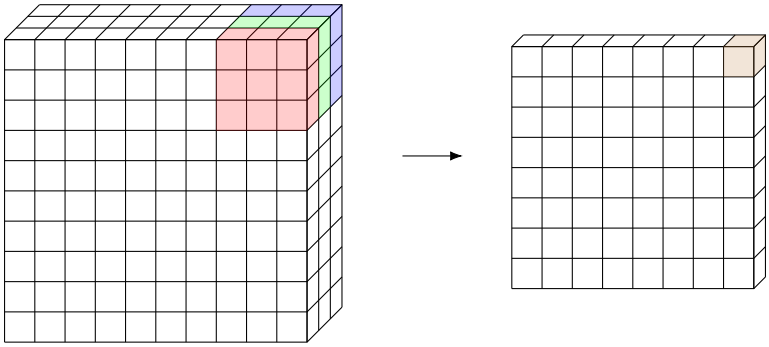


Figure 37: Convolutions over images and feature maps are really three-dimensional convolutions with the input and kernel have the same number of channels.

for $i = 1, \ldots, n - p + 1$ and $j = 1, \ldots, m - q + 1$. This is illustrated in Figure 37.

Observe that the input signal and kernel are both 3-dimensional tensors, but with same size third dimension (i.e., same number of channels d). This means that the filter kernel does not shift along the third dimension and the output signal only has one channel. Confusingly, this is called `Conv2d` in PyTorch.

So far we have considered a single convolutional filter applied to the input signal or feature map. Convolutional layers in deep learning apply multiple filters and stack each of their single-channel outputs into a multi-channel tensor as illustrated in Figure 38. So, in general, we have as input a $(C_{\text{in}} \times H \times W)$-tensor and apply K convolutional filter kernels, $\{A^{(1)}, \ldots, A^{(K)}\}$, which are learned parameters of the layer. Here each $A^{(k)}$ has shape $C_{\text{in}} \times P \times Q$. Then, with appropriate padding (to be discussed shortly in Section 4.1.1), the layer will produce as output a $(C_{\text{out}} \times H \times W)$-tensor with $C_{\text{out}} = K$.

4.1.1 Padding, Stride, Dilation, and Bias

It is a little annoying that the convolution operation results in an output feature map that is smaller than the input feature map, i.e., reduces the signal size. This can be addressed by **padding** the input signal, typically done by appending zeros to the beginning and end of the signal, although other values are possible (such as cyclic padding or extending the end values). Given a filter kernel of length p, appending $p - 1$ values will result in an output signal the same size as the input signal as shown in Figure 39.

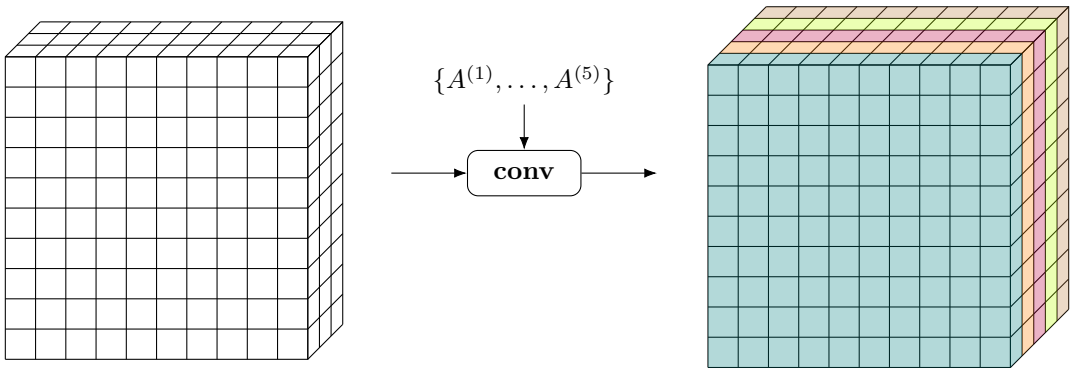


Figure 38: Applying multiple convolutions to the input to generate a multi-channel output. Here we show five convolutions being applied to a three-channel input, producing a five-channel output, one from each convolutional kernel.

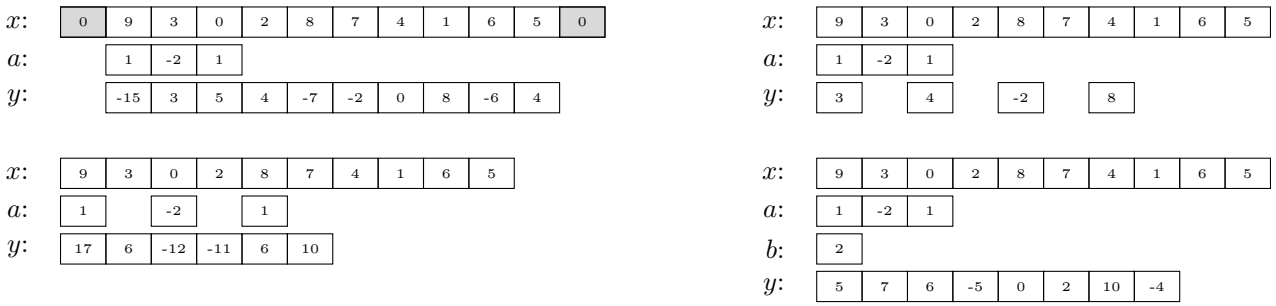


Figure 39: Results from (zero) padding, stride, dilation and bias shown going left-to-right then top-to-bottom for a one-dimensional convolution example.

Other variations of convolution are also possible. For example, we can change how far we move the convolutional filter for each successive output element. This is known as **stride**, and can be expressed mathematically for the one-dimensional case as,

$$y_i = \sum_{j=1}^{p} a_j x_{s(i-1)+j} \qquad \text{for } i = 1, \ldots, \left\lfloor \frac{n-p+1}{s} \right\rfloor \tag{144}$$

for a stride of s. This reduces the size of the output signal by roughly a factor of s and, similarly, reduces computation. Figure 39 shows an example for $s = 2$.

Instead of skipping inputs as we move to the next output, we can also skip inputs when multiplying by filter coefficients. This is called **dilation** or (**atrous**) convolutions, and is equivalent to having a larger filter kernel with zeros inserted, but computationally more efficient. Mathematically, we have

$$y_i = \sum_{j=1}^{p} a_j x_{i+d(j-1)} \qquad \text{for } i = 1, \ldots, n - dp + d \tag{145}$$

for a dilation factor of $d \geq 1$. Here $d = 1$ indicates no dilation.

Last, it is standard to add a constant term, called a **bias**, to the result of the convolution, which can be written as,

$$y = a * x + b \tag{146}$$

for scalar parameter b, which is learned during training along with the kernel parameters a. The striking similarity to the affine transform, $y = Ax + b$, within a multi-layer perceptron should not go unnoticed.

4.1.2 Back-propagation through Convolutions

Consider again the one-dimensional convolution,

$$y_i = \sum_{j=1}^{p} a_j x_{i+j-1} \qquad \text{for } i = 1, \ldots, n - p + 1 \tag{147}$$

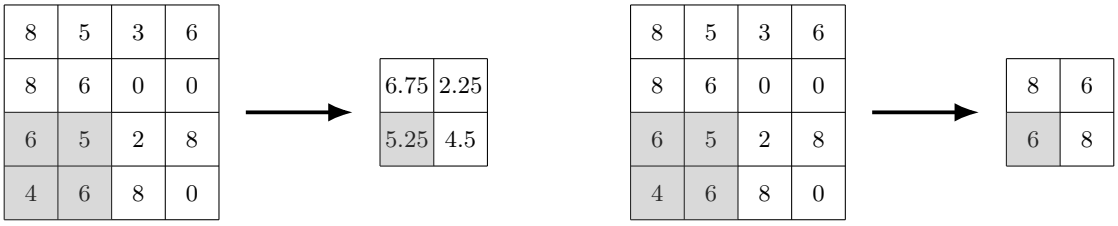
and observe that each a_j affects every y_i. As such, when back-propagating to compute the gradient of the loss with respect to a we need to sum the contributions for all the y_i's. We have,

$$\frac{\mathrm{d}L}{\mathrm{d}a_j} = \sum_{i=1}^{n-p+1} \frac{\mathrm{d}L}{\mathrm{d}y_i} \frac{\mathrm{d}y_i}{\mathrm{d}a_j} \tag{148}$$

$$= \sum_{i=1}^{n-p+1} \frac{\mathrm{d}L}{\mathrm{d}y_i} \cdot x_{i+j-1} \tag{149}$$

$$\therefore \ \frac{\mathrm{d}L}{\mathrm{d}a} = \frac{\mathrm{d}L}{\mathrm{d}y} * x \tag{150}$$

So it turns out that back-propagating through a convolutional layer in a deep learning network is itself a convolution operation!



(a) average pooling **(b)** max pooling

Figure 40: Example average and max pooling operations for a simple 4×4 feature map.

Likewise, each x_k affects many y_i. Here, however, the resulting expression has the convolution filter reversed

$$\frac{\mathrm{d}L}{\mathrm{d}x_k} = \sum_{i=1}^{n-p+1} \frac{\mathrm{d}L}{\mathrm{d}y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_k} \tag{151}$$

$$= \sum_{i=1}^{n-p+1} \frac{\mathrm{d}L}{\mathrm{d}y_i}\frac{\mathrm{d}}{\mathrm{d}x_k}\overbrace{\left(\sum_{j=1}^{p} a_j x_{i+j-1}\right)}^{y_i} \tag{152}$$

$$= \sum_{i=1}^{n-p+1} \frac{\mathrm{d}L}{\mathrm{d}y_i} \cdot \underbrace{a_{k-i+1}}_{\text{reversed}} \qquad \text{(setting } k = i + j - 1\text{)} \tag{153}$$

$$\therefore \ \frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y} * \mathbf{rev}(a) \tag{154}$$

Similar expressions can be derived for higher-order convolutions.

4.1.3 Pooling

In addition to convolution layers, convolutional neural networks include pooling layers that reduce the size of features maps by aggregating feature values over a small local window, which is stepped across the feature map. This not only saves computation in downstream convolution layers but also acts to compress information and make the network more robust to data perturbations and small shifts within the scene. There are two common pooling operations. **Average pooling** computes the average of feature values in the window, where as **max pooling** takes the maximum value, and is hence a non-linear operation. These pooling methods are illustrated in Figure 40.

A 2×2 window is very typical, with a stride of 2 (i.e., stepping the full window size), so that windows do not overlap. This results in downsizing a 2D feature map by a factor of four. Note that average pooling is the same as filtering (i.e., convolution) with a uniform kernel. For multi-channel feature maps, the pooling operation is applied separately on each channel.[12]

4.1.4 Convolutional Neural Networks

We now have all the ingredients necessary for constructing a convolutional neural network (CNN), which takes an image, i.e., $(3 \times H \times W)$-tensor, and performs successive layers of convolutions, elementwise non-linear transforms (such as ReLU), and pooling. Early CNNs also performed features normalization, i.e., $z \leftarrow z/\|z\|_2$, but that is less common today. The final layers of a CNN take the last feature map, flatten it, and then process it through a multi-layer perceptron, also called **fully-connected** or **dense** layers in the context of CNNs. An architecture diagram for the famous AlexNet CNN [56] is shown in Figure 41. There are many other variants appearing in the literature and that we'll see in future lectures.

As can be seen in the diagram for AlexNex, the feature maps have smaller spatial dimension in later layers of the network compared to earlier layers. If we map back the size of the convolutional filter kernels to the original image then we see that a small kernel at a later layer in the network is processing data from quite a large area in the image. This is known as the **receptive field** of the filter. Figure 43 illustrates the idea using three successive layers of 2-by-2 pooling on a 12-by-12 image. By the third layer, each element of the feature map corresponds to data from a 4-by-4 region of the image, and so a 3-by-3 kernel applied on the third-layer feature map would corresponds to a receptive field of 12-by-12 in the original image. Thus, deeper models allow for consideration of larger receptive fields, which are necessary for capturing context, but without the computational demand of larger filters.

[12] A early novel pooling method called **max-out** [30] operates across channels, but is less popular these days.

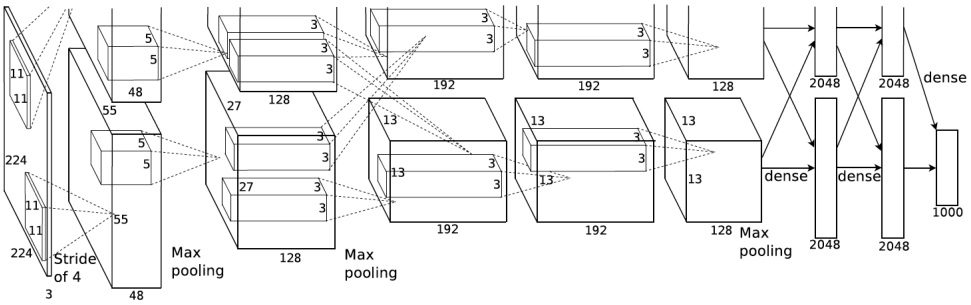
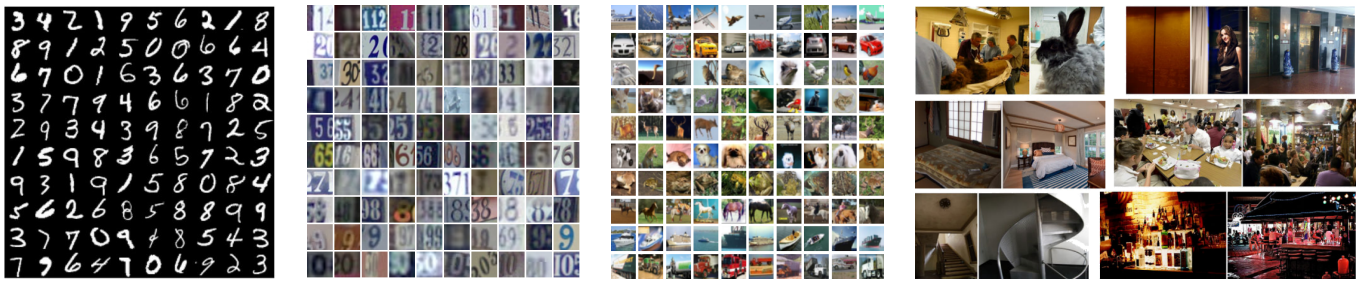


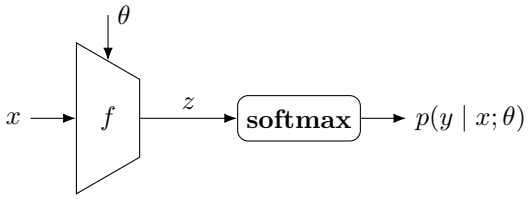
Figure 41: Architecture diagram of the famous AlexNet convolutional neural network for image classification (from [56]). The diagram shows the data shapes (i.e., tensors) at each stage of the network, together with convolutional filter size, and annotated with the pooling operation being performed. For example, the leftmost part of the diagram shows that the input to the network is a 224-by-224 pixel colour image, which padded to 227-by-227 (not indicated in the diagram) and then filtered by 48 convolutional kernels of size 11-by-11 with stride of 4 to produce a 48-channel, 55-by-55 element feature map. Processing through the convolutional layers is performed in two streams (for practical memory capacity reasons) and combined in the final fully-connected layers.



(a) MNIST **(b)** SVHN **(c)** CIFAR10/100 **(d)** Places365

Figure 42: Common datasets used for the image classification task.

The first and still most common use of convolutional neural networks is for image classification—that is, given an image, predict the class that it belongs to (from some pre-defined set of categories). Instead of directly producing a class label, CNNs produce a probability distribution over classes, using a softmax function on the final layer logits, from which the most likely class can be trivially obtained. So for a K-class problem, the CNN **backbone** maps from a $(3 \times H \times W)$-tensor representing the image to a K-vector as shown diagrammatically below:



The size of the input to the CNN is fixed, and therefore images must be resized before processing. We will see later architectures that deal with inputs of varying sizes.

Common datasets used in image classification (see Figure 42) include MNIST [57], SVHN [68], CIFAR10/100 [55], ImageNet [20], Places365 [64], and many more. The first three are very small by modern standards and only used to try out new ideas, not in any real application. Training is done using stochastic gradient descent on the cross-entropy loss, and evaluation performed by reporting top-1 and top-5 accuracy on a held out test set. Here top-1 accuracy is the same as one minus the misclassification rate. Top-5 accuracy measures the number of times the true positive class appears within the set of five most probable classes (ranked by softmax probability, or equivalently, logit value). More nuanced metrics such as confusion matrices and precision-vs-recall will be discussed shortly.

In addition to being a useful task in and of itself, image classification is often used as a test bed for developing new learning algorithms, and as a building block for more sophisticated image processing pipelines that we will see in later lectures.

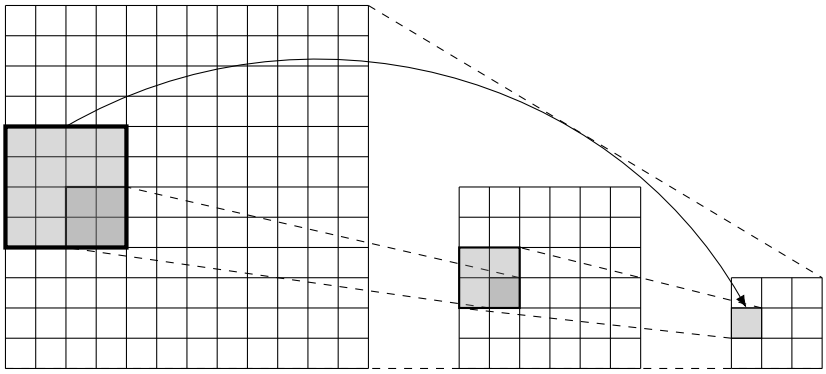
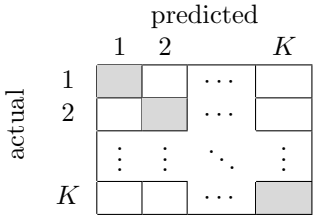


Figure 43: Receptive field of convolutional layers in the image increases as feature maps are downsized.

4.2 Evaluation Metrics

Accuracy, which is the ratio of the number of correct predictions to the total number of examples, only tells part of the story of how well a model is performing. To get the full story we need to evaluate the model using multiple different metrics. Moreover, for different applications, we may need to adjust the metrics to capture the important characteristics of the task at hand. For example, for unbalanced data accuracy is a poor metric. Imagine having 100 times more negative examples than positive ones. A so-called black stick classifier that labels everything as negative will be correct $100/101 = 99\%$ of the time, even though it never detects a single positive example.

To get a more complete picture of how a model is performing we need to compute a **confusion matrix**. This is a two-dimensional array with actual labels down the rows and predicted labels across the columns. The (i, j)-th entry in the table indicates the number of examples with actual class label i that get predicted as being of class j,



The sum of the rows gives the number of ground-truth examples for class i, whereas the sum down the columns gives the number of ground-truth examples that were predicted as class j. The diagonal sum is the number of correctly classified examples, and the sum over all entries in the array and the total number of examples. From this we can derive several statistics. For example, accuracy is the diagonal sum divided by the total sum. Indeed, the confusion matrix is cumbersome to display for large label spaces so we often just use it conceptually for deriving various summary statistics.

We can be creative in constructing our confusion matrices, ignoring certain categories and adding others. The number of rows and columns need not be the same. This allows for fine-grained or coarse-grained analysis. For example, we could group different animal categories into a single column and then analyse how many times a specific animal, say a *cat*, is classified as any one of the *animal* categories.

Getting back to accuracy. What we've been calling accuracy until now is an example of **micro-averaging**,

$$\text{accuracy}_{\text{micro}} = \frac{\text{number of correct predictions}}{\text{total number of examples}} \tag{155}$$

However, for unbalanced data we might wish to treat each class equally, and instead perform **macro-averaging**,

$$\text{accuracy}_{\text{macro}} = \frac{1}{K}\sum_{k=1}^{K} \frac{\text{number of correct predictions for class } k}{\text{total number of examples for class } k} \tag{156}$$

More generally, we can compute weighted accuracy where weights can be assigned per class or per example,

$$\text{accuracy}_{\text{weighted}} = \frac{1}{W}\sum_{i=1}^{N} w_i [\![\text{example } i \text{ correct}]\!] \tag{157}$$

where $W = \sum_{i=1}^{N} w_i$ and $[\![\cdot]\!]$ is the indicator function or Iverson bracket, returning one if it's argument is true and zero otherwise.

4.2.1 Metric for Binary Classification

For binary classification problems the confusion matrix is a 2-by-2 array,

		predicted 0	predicted 1
actual	0	TN	FP
	1	FN	TP

and the entries have special names. A **true positive** (TP), sometimes called a hit or detection, is a sample that has a ground-truth positive label and also predicted to be positive by the model. Likewise, a **true negative** (TN) or correct rejection is a sample that has a ground-truth negative label that is also predicted to be negative by the model. A **false positive** (FP), false alarm, or Type I error, on the other hand is a ground-truth negative sample that is predicted to be positive by the model, and a **false negative** (FN), miss, or Type II error, is a ground-truth positive sample that is predicted to be negative by the model.

It is important to note that for a sample x to be predicted as either positive or negative, a **detection threshold** t needs to be applied to the probability that is calculated by the model, i.e., $p(y = 1 \mid x; \theta) \geq t$ means that x will be labeled as positive and otherwise negative. The entries in the confusion matrix are a function of this threshold. Setting a high threshold will have the model predict more negatives, whereas setting a low threshold and the model will predict more samples as positive.

We can derive several statistics from the four basic elements in the confusion matrix. **Recall**, also known as true positive rate, sensitivity or hit rate, is defined as the number of true positives divided by the total number of ground-truth positives,

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{158}$$

Precision or positive predictive power is the number of true positives divided by the total number of samples predicted as positive by the model,

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{159}$$

A less common statistic in computer vision, the true negative rate or specificity, is the number of true negatives divided by the total number of ground-truth negatives,

$$\text{specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{160}$$

Accuracy, as we have already seen, is the sum over the diagonal entries divided by the sum over all entries,

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TN} + \text{FP} + \text{FN} + \text{TP}} \tag{161}$$

A statistic that is commonly used to summarize precision and recall into a single number is the F_1-score, which is the harmonic mean of precision and recall

$$F_1\text{-score} = \left(\frac{1}{2}\left(\frac{1}{\text{precision}} + \frac{1}{\text{recall}}\right)\right)^{-1} = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{162}$$

There are many other statistics used to report performance of binary classification methods, such as false alarm rate, false positive rate, fall-out, false discovery rate; F_β-score, but the ones defined above are the most common for tasks in deep learning and computer vision.

We already mentioned that the statistics above depend on a threshold applied to the probability output by the model. This results in the **classification rule** (for returning a positive prediction),

$$p(y = 1 \mid x; \theta) \geq t \tag{163}$$

For multi-class problems the classification rule is sometimes applied in one-vs-all fashion,

$$p(y = k \mid x; \theta) \geq t \tag{164}$$

for class k. Note that this is different to the rule of taking the class with maximum probability.

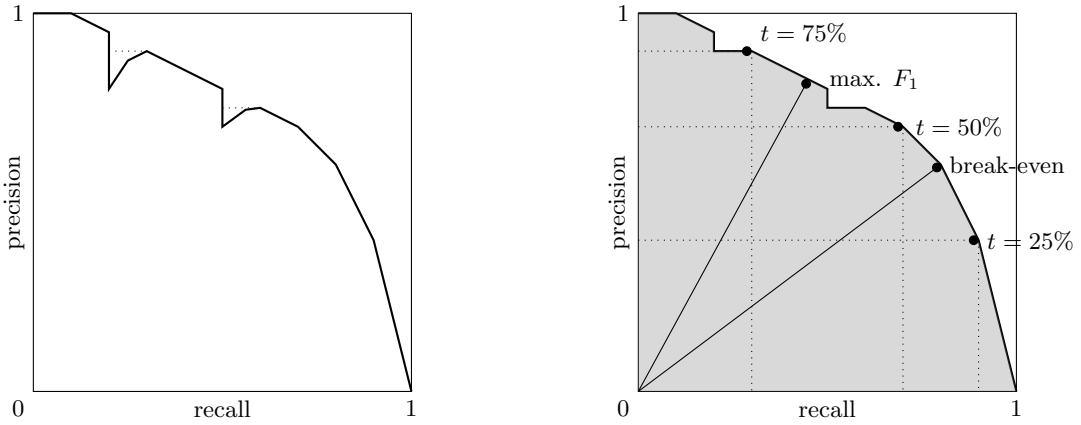


Figure 44: A precision-recall (PR) curve plots precision versus recall for a binary classifier. The curve is traced out by decreasing the threshold for detection, typically raising the recall and lowering the precision. The relationship is not, however, monotonic (left). Typically an interpolated curve (dotted) is plotted instead. Each point on the curve corresponds to a different operating threshold (right). The area under the PR curve, called the average precision (AP), is a very common metric used for comparing classifiers.

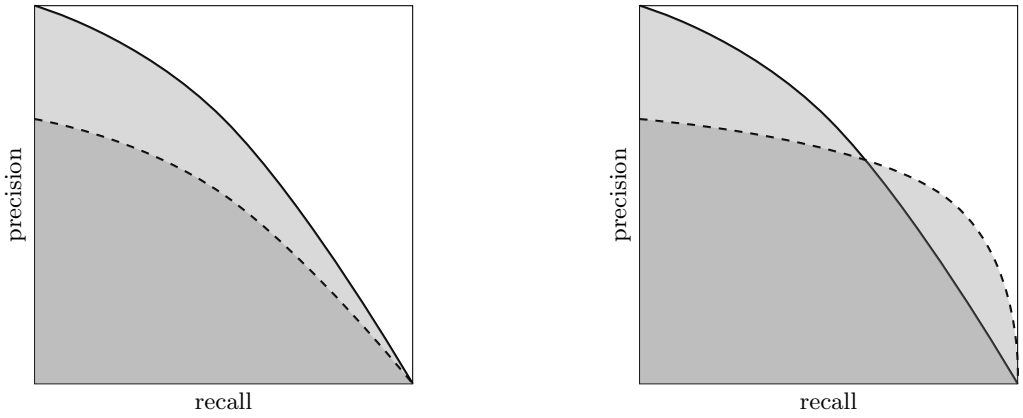


Figure 45: Comparing models using their PR-curves. It is sometimes clear that one model dominates the other (left). However, in many cases which model to choose depends on the operating point (right)—do you want high recall or high precision?

By sweeping over threshold t we can trace out a **precision-recall (PR) curve** as shown in Figure 44. Recall, plotted on the horizontal axis increases from left-to-right corresponding to a decrease in threshold t. The relationship is not necessarily monotonic and so an **interpolated precision** is often plotted instead,

$$\text{prec}_{\text{interp}}(r) = \max_{r' \geq r} \{\text{prec}(r')\} \tag{165}$$

The justification here is that in cases where precision dips, we could always chose another operating point (with lower threshold) where we get to improve recall while maintaining precision.

The area under the precision-recall curve is called **average precision** (AP). It is traditionally computed on an 11-point interpolated curve (i.e., recall ranging from 0 to 1 in increments of 0.1), which was introduced by the PASCAL VOC Challenge [25]. For multi-class tasks we aggregate AP on each category to get the **mean average precision** (mAP), which is a very standard metric used to compare and rank object classification models.

A single summary statistic is never sufficient for comparing models. Figure 45 shows example scenarios of precision-recall curves for two different models. In the plot on the left it is clear that one model dominates the other for all values of precision and recall. In the plot on the right, both models have approximately the same average precision (area under the curve) and one model isn't strictly better than the other at all operating points. Here we would select one model (solid line) if we cared more about precision than recall, and the other model (dashed line) if our application demands higher recall at the cost of precision.

Other metrics that you will often see reported when researchers evaluate models are the number of model parameters and the speed of inference. Sometimes training time is also reported, but this is of less concern for models that are trained once and deployed to millions of users.

Two other factors should be considered when reporting performance and comparing models. First is to run with

multiple different random seeds and report mean and standard deviation across the runs. Second is to report trends against one or more meta-parameters rather than a single operating point. These both give a sense of the method's robustness and mitigates against favoring one model due to luck. However, they are not always feasible with large models/datasets. In such situations—reporting a single operating point for a single run—at least be aware that you are violating good experimental practice (and be skeptical of results reported in the literature this way).

Finally, the most important thing when developing a new model and assessing its performance is to **always visualize your data and your results** to make sure that your metrics align with your intuition. Every researcher has a horror story (some have several) of corrupted data, buggy code, or some other error or misunderstanding that could easily have been avoided using data visualization. We will drive this message home further in a later lecture on debugging algorithms and diagnosing problems.

4.2.2 Why Convolutions?

Let us get back to the question of why we need convolutions. At the start of the lecture we calculated the number of parameters needed for an image classifier based on a two-layer perceptron architecture that could distinguish between 10 different classes. We arrived at 150,639,010 parameters. Now let's analyze the AlexNet convolutional neural network architecture, which can achieve much better performance than an MLP and handles 1000 different classes. With reference to Figure 41 we can calculate the number of parameters as

$$\begin{aligned}
\#\text{ parameters} = {} & 2 \times (11 \times 11 \times 3 + 1) \times 48 + && \text{(conv1)} \\
& 2 \times (5 \times 5 \times 48 + 1) \times 128 + && \text{(conv2)} \\
& 2 \times (3 \times 3 \times 128 + 1) \times 192 + && \text{(conv3)} \\
& 2 \times (3 \times 3 \times 192 + 1) \times 192 + && \text{(conv4)} \\
& 2 \times (3 \times 3 \times 192 + 1) \times 128 + && \text{(conv5)} \\
& (6 \times 6 \times (128 + 128) + 1) \times (2048 + 2048) + && \text{(fc1)} \\
& (2048 + 2048 + 1) \times (2048 + 2048) + && \text{(fc2)} \\
& (2048 + 2048 + 1) \times 1000 && \text{(fc3)} \\
= {} & 60,522,856
\end{aligned}$$

where, for example, the first convolutional layer applies an 11-by-11 filter kernel over the 3-channel input. There are 48 filters in the first layer in each of the two data streams, and each filter has a bias term, giving $(11 \times 11 \times 3 + 1) \times 48$ parameters for the first layer.

This represents a massive reduction in parameter count for much deeper network. The convolutional neural network architecture is also invariant to small pixel shifts thanks to max pooling, so will generalization better. Furthermore, the parameter sharing from the fact that the same kernel is applied to all locations in the feature map, also helps the model to generalize. All up, CNNs are a win-win for image classification.

4.3 Other Architectural and Training Ingredients

4.3.1 Data Augmentation

Data augmentation is a technique used to improve the robustness and generalization ability of neural networks by generating new training data from existing data. Since the model is now being asked—through training—to perform well on a larger set of data, overfitting is lessened. Transformations are applied to the existing training samples to create new samples with the same labels, i.e.,

$$(x^{(i)}, y^{(i)}) \rightarrow (T(x^{(i)}), y^{(i)}) \tag{166}$$

where T is the data augmentation transformation function and $(x^{(i)}, y^{(i)})$ is a sample from the training batch. These transformations are typically applied on-the-fly as the data is loaded for a training iteration, and may be specific for a task, unlike general techniques such as regularization. They include a random element so that each time the transform is applied a slightly different data sample is generated. Example data augmentations for computer vision tasks are shown in Figure 46.

4.3.2 Batch Normalization

Recall from the last lecture that normalizing data helps convergence during training. Batch normalization (BN) [45] is a technique for on-the-fly centering and scaling of feature maps within the network. Consider a batch $\mathcal{B} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$

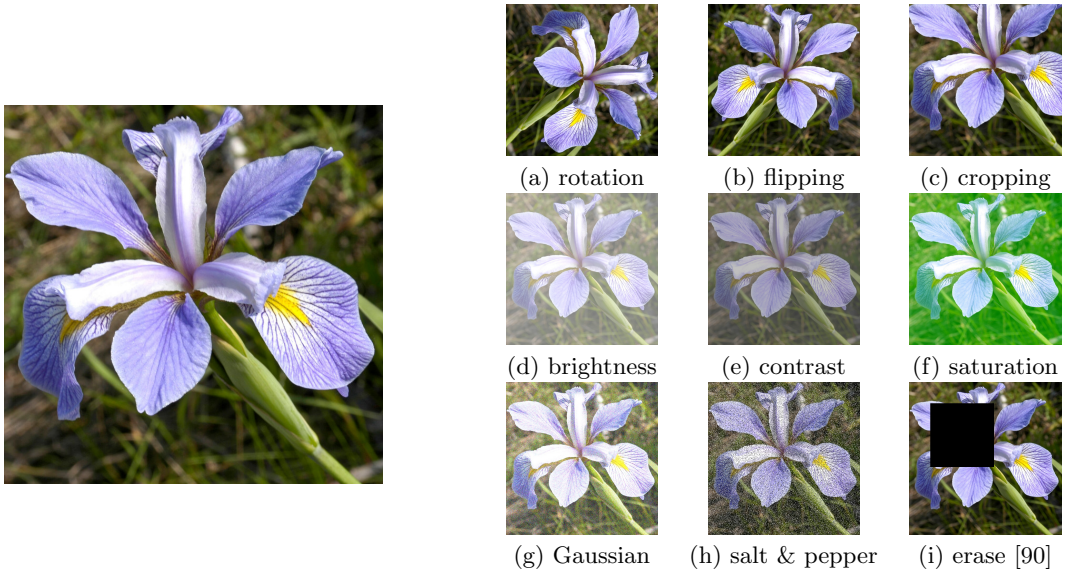


Figure 46: Example data augmentation techniques for computer vision tasks. The original image is shown on the left. Data augmentations include (a)–(c) geometric transformations, (d)–(f) colour space changes, and (g)–(i) adding random noise.

with intermediate features $z^{(i)} \in \mathbb{R}$, then BN calculates as its output

$$\hat{z}^{(i)} = \gamma \left(\frac{z^{(i)} - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}} \right) + \beta \tag{167}$$

where

$$\mu_{\mathcal{B}} = \frac{1}{N} \sum_{i=1}^{N} z^{(i)} \quad \text{and} \quad \sigma_{\mathcal{B}} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (z^{(i)} - \mu_{\mathcal{B}})^2} \tag{168}$$

are the mean and standard deviation over the batch, respectively, and γ and β are learnable scale and offset parameters. For vector- or tensor-valued features, BN is applied to each index separately.

Running averages of μ and σ are maintained for use during test time, since computing batch statistics at test time would violate the identically and independently distributed (i.i.d.) assumption of the test data.

4.3.3 Residual Connections and ResNet

One way to address the issue of diminishing gradients is via short-cut (or bypass or residual) connections that provide an identity transformation path in addition to some feature processing path. The ResNet block [39] shown in Figure 47 is a classic example. Let x be the input signal and let f be any neural network processing function, e.g., $f(x) = C\sigma(Ax + b) + d$. Then, after adding a residual signal we have,

$$z = f(x) + x \tag{169}$$

Note that in the original paper [39] ResNet was applied to the task of image classification. As such, the linear transformation was implemented as a 3D convolutional layer. The block also included batch normalization within the function f.

If we now consider the gradient through the ResNet block we have,

$$\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}z} \left(\frac{\mathrm{d}z}{\mathrm{d}x} + I \right) \tag{170}$$

As such, even if $\mathrm{d}z/\mathrm{d}x$ vanishes, we still have a gradient signal $\mathrm{d}L/\mathrm{d}z$ propagating through to the input. He et al. [39] showed that it is possible to train networks with a thousand layers using this trick by stacking successive ReNet blocks (see Figure 48). Today ResNet models are one of standard backbone architectures for all sorts of deep learning applications, going under the monikers of ResNet18, ResNet34, ResNet50 and ResNet101, which denote the number of layers.

It is quite easy to implement a ResNet block in PyTorch. Here we present code for a very basic convolutional version with batch normalization.

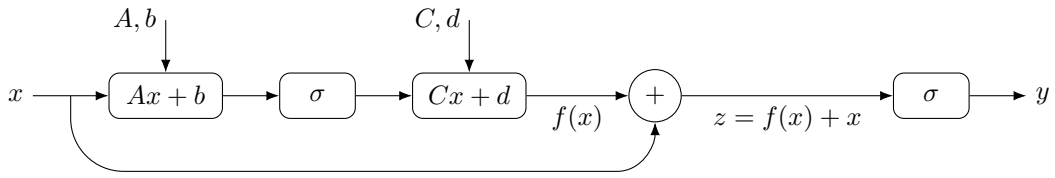


Figure 47: General form of a ResNet [39] block, which include a short-cut connection from the input. The linear layers can be either convolutional layers or fully-connected layers. The activation functions are typically rectified linear units (ReLU).

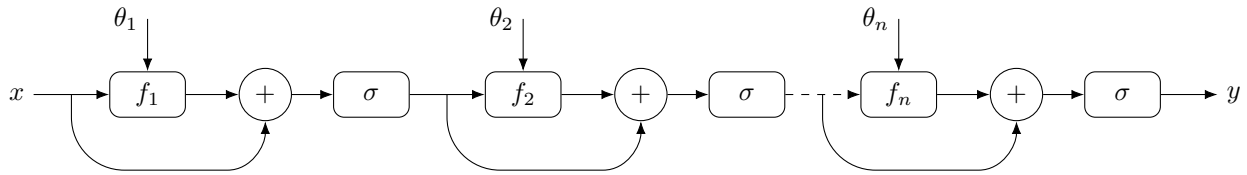
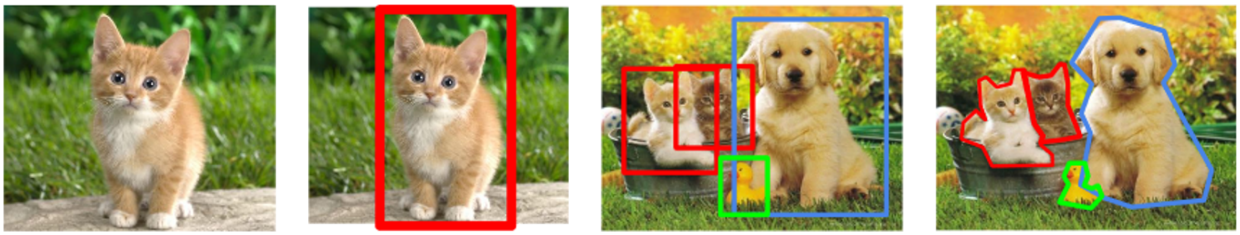


Figure 48: Stacking ResNet blocks to produce a very deep network that does not suffer from vanishing gradients. Parametrized functions f_i are typically two-layer perceptrons or two-layer convolutional neural networks, $f_i(z) = C_i\sigma_i(A_i z + b_i) + d_i$ where A_i and C_i are either arbitrary weight matrices or Toeplitz matrices (for convolutions), and σ_i is an elementwise activation function.

```python
    class BasicResNetBlock(nn.Module):
        """Example convolutional ResNet block."""

    def __init__(self, in_planes, out_planes):
        super().__init__()

        self.conv1 = nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_planes)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_planes, out_planes, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_planes)

    def forward(self, x):

        f = self.bn2(self.conv2(self.relu(self.bn1(self.conv1(x)))))
        z = f + x
        y = self.relu(z)

        return y
```



(a) image classification (b) object localization (c) object detection (d) instance segmentation

Figure 49: Classification, localization, detection and segmentation. (Image courtesy: Karpathy, Johnson, et al.)

5 Object Detection

This lecture moves from image classification to object detection. We start by discussing the detection task and how it is evaluated. We then present some simple approaches for building an object detector from an image classification model. These approaches are progressively enhanced to arrive at state-of-the-art object detection methods. We end the lecture with a brief discussion on evaluating and comparing methods.

The task of **image classification** studied in the previous lecture works great for images of single objects (or tagging scenes of a particular type like *city* or *rural* or *forest*). We could also imagine a task where we cropped the image around the single object within it, called **localization**, to make the subject of classification more precisely cropped. But most scenes contain more than one object. **Object detection** is the task of finding multiple objects in an image by placing a bounding box around each object found and assigning a label to that bounding box. An even more detailed detection and localization task, called **instance segmentation**, will be discussed in a future lecture. Figure 49 illustrates the difference between these tasks.

More formally, we can define image classification as the task that takes as input a $(3, H, W)$-tensor representing a colour image and produces a class label from a set of K possible classes. The task is evaluated based on an accuracy metric, i.e., how many images were correctly classified from the hold-out test set. The localization task also takes an image as input but produces box coordinates (x, y, w, h) denoting the top-left coordinate of the object $(x, y) \in [0, W-1] \times [0, H-1]$, its width $w \in [1, W-x]$ and its height $h \in [1, H-y]$.[13] To avoid having to deal with images of different sizes, the box coordinates are usually normalized to between zero and one.

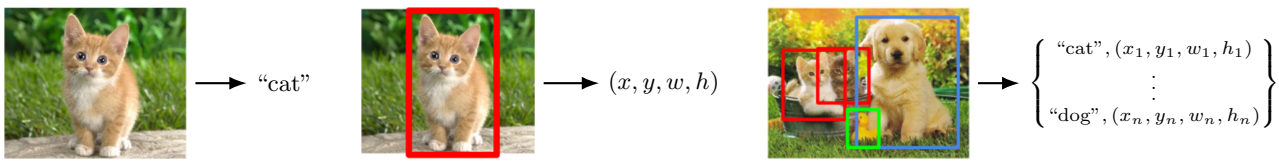
Since localization is regressing to a 4-dimensional vector (denoting the bounding box coordinates) we cannot simply compare to some exact ground-truth annotation, where a single pixel difference would result in an incorrect prediction. Instead we use a more forgiving intersection-over-union metric that will be discussed shortly. The task of object detection, then, can be summarized as performing both classification and localization, and is therefore scored based on both accuracy (correct class) and localization (correct bounding box) metrics. Moreover, instead of outputting a single category and bounding box, object detection must return a list of category-bounding box pairs, one for each object in the image. An illustration of the difference between the three tasks is shown in Figure 50.

Intersection-over-union (IoU), also called Jaccard index, measures the similarity of two sets. By thinking of bounding boxes as sets of pixels or areas in two-dimensional space, we can use IoU in object detection to determine whether two bounding boxes are close. Given two sets A and B, we define the intersection-over-union of the sets as,

$$\text{IoU}(A, B) = \frac{A \cap B}{A \cup B} \tag{171}$$

Note that this is symmetric in A and B, and is one if and only if A and B coincide exactly. IoU is zero if the two sets are disjoint. A graphical depiction of intersection-over-union for two-dimensional bounding boxes is shown in

[13] Note the zero-based indexing here. Note also that bounding boxes are sometimes represented by the top-left and bottom-right coordinates, i.e., (x_1, y_1, x_2, y_2) instead of top-left, width and height, (x, y, w, h).



(a) classification (b) localization (c) detection

Figure 50: Illustration of image classification, object localization and object detection tasks.

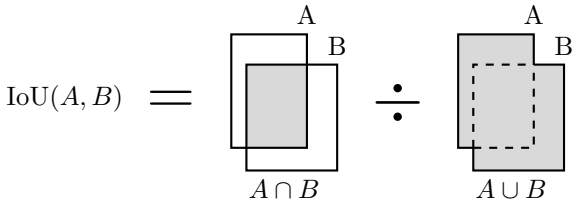
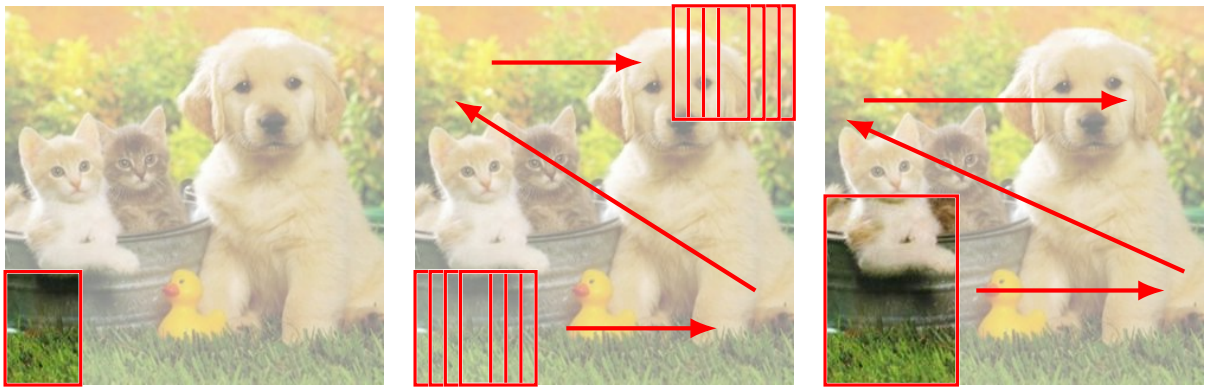


Figure 51: Graphical depiction of calculating the intersection-over-union metric.



(a) evaluate classifier on contents of bounding box

(b) slide window over all locations in image

(c) repeat at all different scales

Figure 52: Sliding-window approach to object detection.

Figure 51 and can be computed in terms of bounding box coordinates as,

$$\text{IoU}(A, B) = \frac{w_{A\cap B} h_{A\cap B}}{w_A h_A + w_B h_B - w_{A\cap B} h_{A\cap B}} \tag{172}$$

where

$$w_{A\cap B} = \max\{\, 0,\, \underbrace{\min\{x_A + w_A, x_B + w_B\}}_{\substack{\text{leftmost} \\ \text{right edge}}} - \underbrace{\max\{x_A, x_B\}}_{\substack{\text{rightmost} \\ \text{left edge}}} \,\} \tag{173}$$

$$h_{A\cap B} = \max\{\, 0,\, \underbrace{\min\{y_A + h_A, y_B + h_B\}}_{\substack{\text{bottommost} \\ \text{top edge}}} - \underbrace{\max\{y_A, y_B\}}_{\substack{\text{topmost} \\ \text{bottom edge}}} \,\} \tag{174}$$

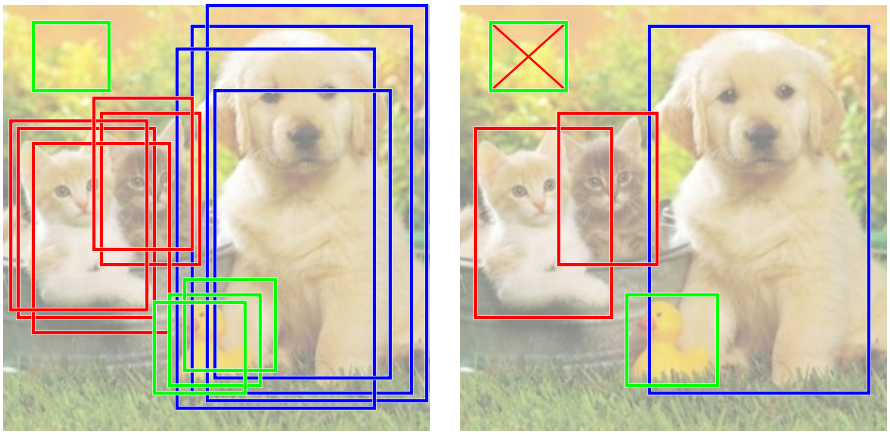
are the width and height of the intersection between boxes A and B, respectively.

For a detection to be considered positive it must both predict the correct class label, and have IoU greater than a given threshold, denoted IoU_t, compared to some ground-truth annotation. Typical thresholds for object detection are 0.3, 0.5 and 0.7 (denoted $\text{IoU}_{0.3}$, $\text{IoU}_{0.5}$ and $\text{IoU}_{0.7}$), where 0.3 is considered quite loose and 0.7 considered quite tight. To prevent multiple detections of the same object we usually only count the highest probability detection matching with a ground-truth bounding box as correct and all other associated detections as incorrect (even if they have the correct label and IoU above the designated threshold).

We know from the previous lecture how to design an image classifier, but the question now arises for object detection: *where do the bounding boxes come from?* One older strategy is known as the **sliding window** approach. Let us assume that we have an already-trained image classifier. In the sliding window approach, we start by picking some nominal bounding box size and place that bounding box at the corner of the image. We then run our image classifier on the cropped contents of the bounding box (see Figure 52). The classifier is augmented with a *background* category, which allows the model to decide that there is no object (or too many objects) within in the window. This amounts to $(K + 1)$-way classification for a K-class object detector.

We then slide the bounding box to the next location in the image and repeat until we have evaluated the classifier at every bounding box location (similar to sliding the filter during 2D convolutions). But this only detects objects at a single scale, the scale defined by the nominal bounding box size. So we increase the bounding box dimensions and repeat sliding the newly sized bounding box over the image and running the classifier on the contents at each location. We keep going until we are satisfied that we have evaluated bounding boxes at all scales of interest.

At the end of the sliding window process we are left with a set of bounding boxes of different scales that cover the image. Associated with each bounding box is a probability distribution over object classes (and background). The



(a) filtering (b) non-maximal suppression

Figure 53: Filtering of low scoring detections followed by non-maximal suppression to get rid of overlapping detections of the same object. High scoring false positive detections may remain as can be seen at the top-left.

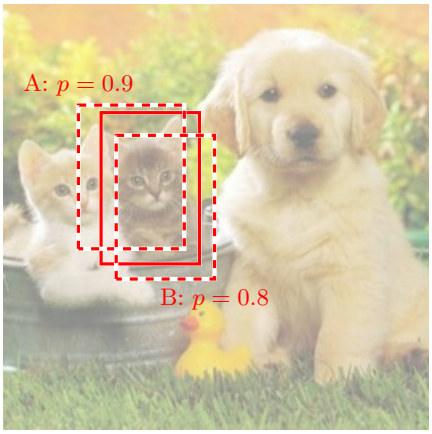


Figure 54: Example of how duplicate detections of the same object may still occur even after non-maximal suppression. Here we show a ground-truth bounding box (solid) obtained from a human annotator. Two bounding boxes, A and B, output by the object detector have low IoU so are not filtered by non-maximal suppression. However, both bounding boxes have IoU above the threshold with the ground-truth. Detection A will be counted as positive since it has higher probability. Detection B will be counted as negative since a higher probability bounding box is already matched with the ground-truth.

class with highest probability is considered to be the class label for the bounding box. Many boxes overlapping the same object will have high probability (for the same class label) and these need to be handled in some way. The simplest approach is to first filter out low probability bounding boxes and then apply **non-maximal suppression** (NMS) to get rid of multiple overlapping detections of the same object. See Figure 53. Here bounding boxes are sorted from highest probability to lowest probability. If a bounding box overlaps significantly with a bounding box of higher probability (i.e., preceding it in the list) and having the same label, then the box is removed from the list. This acts to remove overlapping detections of the same object, but in cases of very crowded scenes may also suppress correct detections of nearby objects.

Non-maximal suppression does not get rid of false-positives. Missed detections also cannot be recovered from the filtering and non-maximal suppression algorithm. Mean average precision (mAP) based on some IoU threshold with ground-truth annotations is then calculated on the list of remaining bounding boxes to determined the performance of the detector. Repeated detections of the same object are counted as false-positives, meaning that the highest probability detection (that meets the IoU requirement) is determined as correct and the rest as incorrect. See Figure 54. This prevents a detector from gaming the metric by labeling every bounding box as a detection.

While the sliding-window approach is simple, it is very costly due to the large number of times we need to evaluate the image classifier, i.e., once per bounding box. It is a fairly straightforward calculation to count the number of unique bounding boxes of any possible size and placed at any possible location within an $H \times W$ image. Specifically, the top-left of the box can be anywhere from $x = 0$ to $W - 1$ and $y = 0$ to $H - 1$. And the width and height of the box

can be anywhere from $w = 1$ to $W - x$ and $h = 1$ to $H - y$. Therefore we have

$$\# \text{ boxes} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} (W - x)(H - y) \tag{175}$$

$$= \frac{W(W+1)}{2} \frac{H(H+1)}{2} \tag{176}$$

$$\approx \tfrac{1}{4} (\# \text{ pixels})^2 \tag{177}$$

So, as an example, for a 1280×720 image there are 212×10^9 unique boxes! Clearly we would like to do better.

The **selective search** algorithm proposed by Uijlings et al. [83] finds a small set of candidate boxes that are likely to contain objects. The algorithm performs a bottom up over-segmentation and then uses agglomerative clustering to merge similar segments. We won't go into the details, but the result is 2,000 class-agnostic **bounding box proposals** per image independent of the image size. This is far fewer than enumerating all possible bounding boxes. However, because the bounding box proposals were found in a class-agnostic way using low-level features they may not align the best for each detected object. We can fix this with a trained bounding box alignment model.

To recap, when performing object detection on an image we take bounding boxes within the image, crop out the region enclosed by the bounding box, resize it, and feed the resized cropped image into a convolutional neural network classifier as if we were performing image classification. At the same time we can perform regression on the bounding box coordinates to improve (i.e., tighten) its localization of the object. Figure 55 shows a typical architecture. A common convolutional neural network backbone computes a feature representation of the cropped image. This representation is then fed into two separate branches. The first branch completes the standard image classification pipeline by passing the features through a multi-layer perceptron followed by a softmax classifier. The second branch passes the features through a different multi-layer perceptron that predicts the refined (4-dimensional) bounding box coordinates.

The model is trained end-to-end through the multi-layer perceptrons and shared backbone using a combination of two loss functions. To train the classifier branch we use a cross-entropy (CE) loss on true label, $y^{(i)}$, whereas to train the bounding box regressor we use a mean-square-error (MSE) loss on true bounding box, $b^{(i)}$. Here the bounding boxes are adjusted to be relative to the cropped region (which is what the network sees). The second loss is only applied to non-background boxes (determined by IoU with ground-truth). The total training loss over a batch of crops $\mathcal{D} = \{(x^{(i)}, y^{(i)}, b^{(i)})\}_{i=1}^N$ is then a weighted combination of the CE and MSE losses,

$$L(\theta; \mathcal{D}) = \sum_{i=1}^{N} -\log p\left(y^{(i)} \mid x^{(i)}; \theta\right) + \frac{\lambda}{2} \begin{cases} \left\| \text{bbox}(x^{(i)}; \theta) - b^{(i)} \right\|^2, & \text{if } y^{(i)} \neq \text{“background”} \\ 0, & \text{otherwise,} \end{cases} \tag{178}$$

where p represents the probability distribution estimated by the classifier branch, and θ represent the combined parameters of the model, including the parameters of the two multi-layer perceptrons and the backbone convolutional neural network. Here λ is a hyperparameter that controls the trade-off between lower cross-entropy loss for classification and lower mean-square-error loss for better localization, though the two need not be in competition in that tuning the backbone network parameters to reduce one may also help to reduce the other. Note that the parameters of the multi-layer perceptrons in the classifier and bounding box branches are only affected by the losses applied to their branch, cross-entropy and mean-square-error, respectively. The backbone convolutional neural network parameters are affected by both losses.

Training data for the detector comes from manually annotated ground-truth, where a human has gone through and drawn a bounding box around every object in the set of training images and provided a label for each box.[14] Crops of these bounding boxes together with crops of proposed bounding boxes obtained from selective search. Any proposed bounding box with sufficiently high IoU with a ground-truth bounding box takes that ground-truth label. Multiple bounding boxes for the same object (i.e., duplicate detections) are allowed during training as these form a type of data augmentation, making the learned model more robust.

[14] Sometimes very small objects and highly occluded objects are ignored.

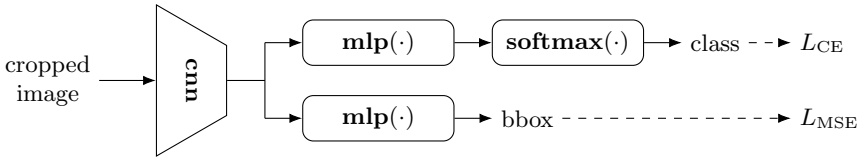
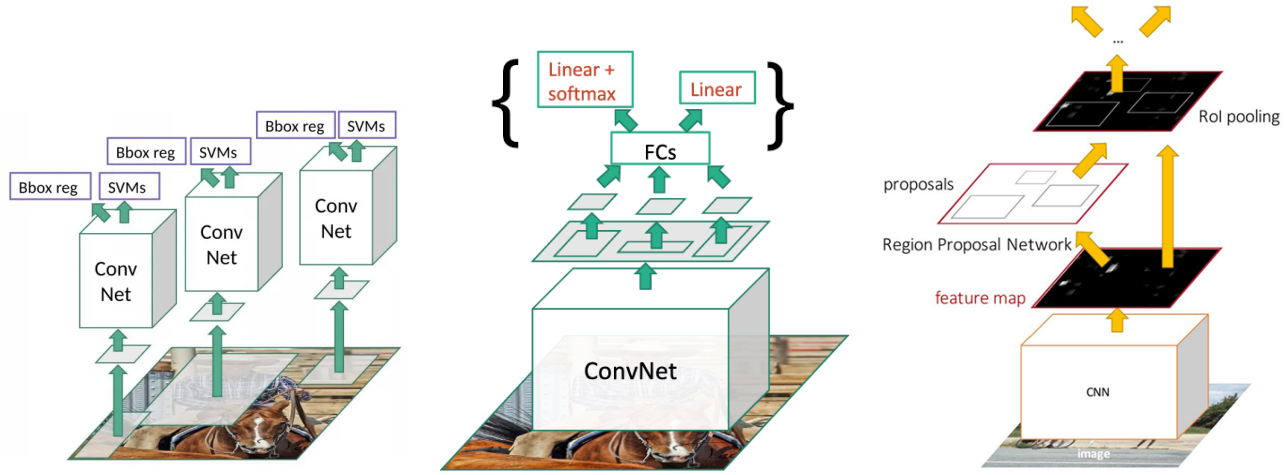


Figure 55: Classifying and refining proposal bounding boxes.



(a) R-CNN [28] (b) Fast R-CNN [27] (c) Faster R-CNN [76]

Figure 56: R-CNN, Fast R-CNN and Faster R-CNN networks for object detection. Images courtesy [28, 27, 76]

.

5.1 Region-CNN Family of Detectors

The first successful neural network object detector model was the R-CNN model proposed by Girshick et al. [28], although it was only partially end-to-end trainable. The basic idea is shown in Figure 56(a) and follows the recipe that we have been discussing thus far. Namely, the model is comprised of a CNN backbone that is pre-trained on the ImageNet dataset for image classification. Selective search [83] is used to propose rectangular regions and image crops are then resized to 227-by-227 pixels independent of the bounding box aspect ratio, and processed by the CNN backbone to obtain a 4096-dimensional feature vector per crop. Linear classification (called a support vector machine in the original paper) and bounding box regression branches operate on this feature representation of the image crop.

To obtain good results the bounding box branch predicts a transformation $(\Delta x, \Delta y, \Delta w, \Delta h)$ of the proposed region $b = (x, y, w, h)$ as

$$\hat{b} = (x + w\Delta x,\ y + h\Delta y,\ w \exp \Delta w,\ h \exp \Delta h) \tag{179}$$

where the translation $(\Delta x, \Delta y)$ is proportional to the box size (w, h), and the scale transformation $(\Delta w, \Delta h)$ is in log-space. This makes the transformation invariant to re-scaling of the image.

One of the drawbacks of the R-CNN model is that the convolutional neural network backbone is repeatedly applied to every single proposed bounding box. This is computationally wasteful, especially for bounding boxes that are overlapping. A successor to R-CNN called Fast R-CNN [27] solves this problem by introducing **region-of-interest (RoI) pooling** (see Figure 56(b)). The idea is to first process the full image through the CNN backbone to obtain a 7-by-7 feature map. Bounding box proposals are projected onto the feature map and pooling performed per projected box proposal to obtain the region's feature vector representation, which is then passed to the classification and bounding box regression branches as before.

Just one problem remains: the model requires pre-processing of the image to obtain bounding box proposals, which as we will see is the main computational bottleneck. The Faster R-CNN model [76] shown in Figure 56 incorporates a **region proposal network (RPN)** that operates on the convolutional neural network feature map to do away with selective search pre-processing. This network predicts and scores bounding box proposals as it processes the image. Importantly, it uses the same CNN backbone allowing computation to be shared between region proposals and region classification.

5.1.1 RoIPool and RoIAlign Details

RoIPool and RoIAlign are operations that allow feature maps computed by the backbone CNN over the whole image to be pooled over an arbitrary rectangular region within it. The proposal regions in image coordinates are first projected onto the feature map. For **RoIPool** the projected region is snapped onto the feature map grid, whereas for **RoIAlign** the coordinates of the projected bounding box remain fractional and bilinear interpolation is used to align the feature map to the region. Max pooling is then applied over projected subregions (with all channels pooled independently). See Figure 57. This produces a fixed size output feature map that is independent of the size of the original proposed bounding box, and hence can be used by a downstream multi-layer perceptron for classification or bounding box regression. The feature map used in Fast R-CNN is 512-by-7-by-7.

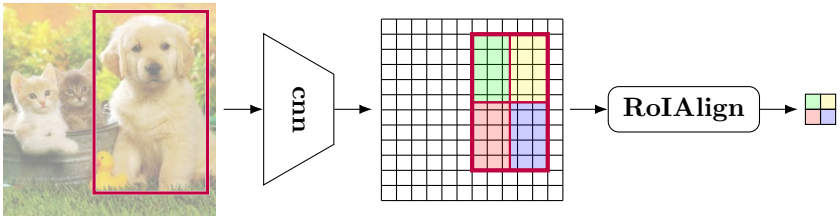
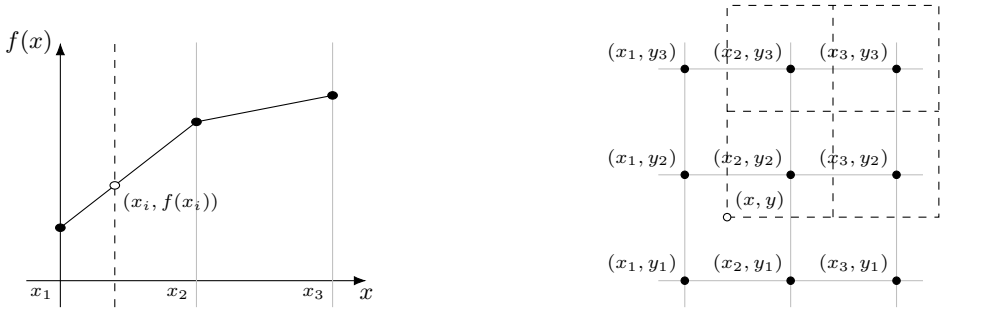


Figure 57: RoIAlign allows features computed on the entire to be pooled over an arbitrary rectangular region.



(a) linear interpolation of a 1D function

(b) bilinear interpolation of a 2D function

Figure 58: Bilinear interpolation allows estimation of features at fractional coordinates.

Bilinear interpolation is a general technique for estimating a function value between discretely sampled points. Let us start by considering the simpler problem of linearly interpolating between any two points on a one-dimensional function $f : \mathbb{R} \to \mathbb{R}$ as shown in Figure 58(a). Picking a point x between two points x_1 and x_2 we have the interpolated value of the function at x as the convex combination of the function values $f(x_1)$ and $f(x_2)$,

$$f^{\text{interp}}(x) = \alpha f(x_1) + (1 - \alpha) f(x_2) \tag{180}$$

where $\alpha = \frac{x_2 - x}{x_2 - x_1}$ is the relative weight assigned to the first point (and $1 - \alpha = \frac{x_1 - x}{x_2 - x_1}$ is the weight assigned to the second point). Note the inverse relationship between the weight and distance to the points.

We can extend this idea to the case of a two-dimensional function $f : \mathbb{R}^2 \to \mathbb{R}$ defined on integer coordinates as illustrated in Figure 58(b). Bilinear interpolation allows us to compute function values at fractional coordinates by taking the weighted average of the four neighbouring points where nearer points are given proportionally higher weight. Here we first linearly interpolating along the x-direction to estimate the function at points (x, y_1) and (x, y_2), and then linearly interpolate between these two points to get the function value at (x, y).[15] The first interpolation gives,

$$f^{\text{linear}}(x, y_1) = \alpha_2 f(x_1, y_1) + \alpha_1 f(x_2, y_1) \tag{181}$$

$$f^{\text{linear}}(x, y_2) = \alpha_2 f(x_1, y_2) + \alpha_1 f(x_2, y_2) \tag{182}$$

where $\alpha_1 = \frac{x - x_1}{x_2 - x_1}$ and $\alpha_2 = \frac{x_2 - x}{x_2 - x_1}$. The second interpolation gives,

$$f^{\text{bilinear}}(x, y) = \alpha_{22} f_{11} + \alpha_{12} f_{21} + \alpha_{21} f_{12} + \alpha_{11} f_{22} \tag{183}$$

where $\alpha_{ij} = \frac{|(x_i - x)(y_j - y)|}{(x_2 - x_1)(y_2 - y_1)}$ and $f_{ij} = f(x_i, y_j)$. By studying this expression we can see that bilinear interpolation sums the function values at each corner point weighted by the relative area to the opposite point, e.g., $(x_2 - x)(y_2 - y)/(x_2 - x_1)(y_2 - y_1)$ is the weight used for $f(x_1, y_1)$.

The expression above can be rewritten into a succinct matrix form,

$$f^{\text{bilinear}}(x, y) = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x \\ x - x_1 \end{bmatrix}^T \begin{bmatrix} f(x_1, y_1) & f(x_1, y_2) \\ f(x_2, y_1) & f(x_2, y_2) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix} \tag{184}$$

for any point $(x, y) \in \mathbb{R}^2$ lying within a box defined by integer coordinates (x_1, y_1) and (x_2, y_2) for the bottom-left and top-right corners, respectively.

[15] Alternatively we could first interpolate in the y-direction at points (x_1, y) and (x_2, y) and then interpolate between these points to get the same result.

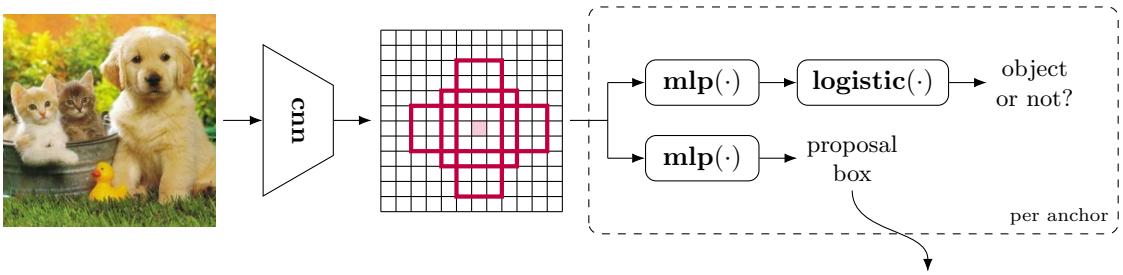


Figure 59: Region proposal network (RPN) architecture.

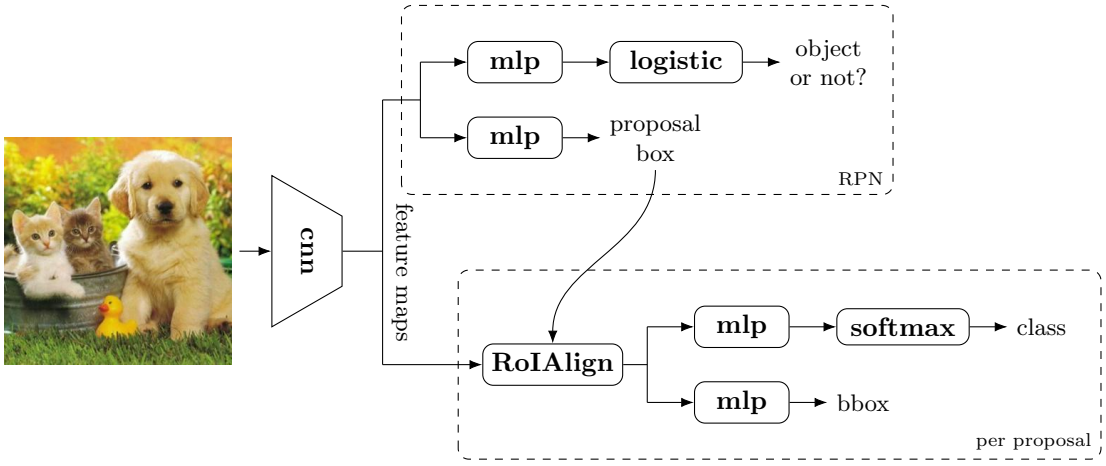


Figure 60: Overview of the Faster R-CNN model architecture.

5.1.2 Region Proposal Network Details

The region proposal network (RPN) in Faster R-CNN associates **anchor boxes** with each location in the feature map obtained from the backbone networks. These anchor boxes are of various sizes and aspect ratios, centered on the feature map location. See Figure 59. A multi-layer perceptron is used to classify each anchor box as either containing an object or not. For positive boxes, a second multi-layer perceptron regress bounding box transformation as described above for the R-CNN model.

The top-k scoring proposal boxes are fed to an RoIAlign module (along with feature map) to produce a feature representation for the corresponding bounding box. The model is trained end-to-end using a weighted combination of four loss components that include a binary cross-entropy loss and a transformation regression loss on anchor boxes, and multi-class cross-entropy and bounding box regression on proposals.

Putting all of these components together we arrive at the Faster R-CNN architecture shown in Figure 60. The CNN backbone processes the entire image to produce a feature map; the region proposal network generates bounding boxes; each proposed bounding box is fed to RoIAlign, which pools over the associated region in the feature map for classification and bounding box regression.

Including region proposals into the same deep learning model as the bounding box classifier (and regressor) makes a significant difference to inference speed. Shown in Figure 61 is the time taken by different models in the R-CNN family to process a single image. Pre-processing the image to obtain selective search bounding box proposals takes approximately two seconds per image. This is a bottleneck in the Fast R-CNN model, which is alleviated in the Faster R-CNN model. As we will see later, the accuracy of the model (Faster R-CNN) is also slightly improved, although this could be attributed to multiple factors.

5.2 Single Stage Detectors

The object detectors described thus far are called **two-stage detectors** because they first involve proposing a set of bounding boxes and then each bounding box is individually classified. There is an alternative paradigm called **single-stage detection**, which does everything all at once. This approach to object detection was proposed at about the same time by Redmon et al. [75] and Liu et al. [62], although variants of the YOLO model of Redmon et al. [75] are by far the more popular model these days.

The single-stage detection approach is very similar to Faster R-CNN. Instead of first proposing bounding boxes that are then processed by pooling features over their corresponding regions, the single stage approach directly predicts

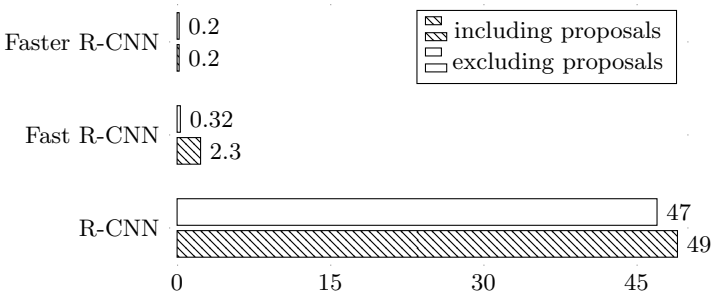


Figure 61: Inference time per image (in seconds) for different variants of R-CNN. Pre-processing associates with bounding box proposals takes about two seconds per image. Note that Faster R-CNN incorporates proposal generation into the network.

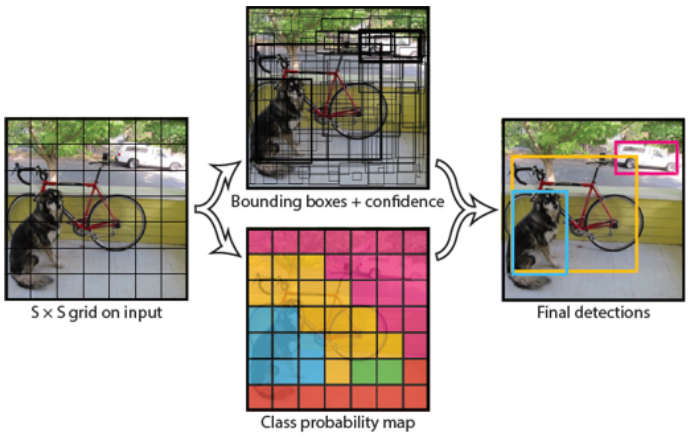


Figure 62: YOLO [75] network architecture for single-stage object detection.

bounding box coordinates and class probabilities from the same anchor point. Specifically, the input image is divided into a grid of S-by-S cells as illustrated in Figure 62. For each cell location we regress and classify m bounding boxes, resulting in an $(S \times S \times (5m + K))$-tensor output, where the $5m$ corresponds to the bounding box locations and scores and the K corresponds to the class probabilities. Flattening out the first dimensions we can think of the single-stage detector as taking in a fixed-size tensor, a $(3 \times H \times W)$-image, and producing a fixed-size mS^2-length list of scored bounding boxes and class probability distributions,

$$\underbrace{\begin{bmatrix} x_1 & y_1 & w_1 & h_1 & s_1 & p_1^{(1)} & \dots & p_1^{(K)} \\ x_2 & y_2 & w_2 & h_2 & s_2 & p_2^{(1)} & \dots & p_2^{(K)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ x_{mS^2} & y_{mS^2} & w_{mS^2} & h_{mS^2} & s_{mS^2} & p_{mS^2}^{(1)} & \dots & p_{mS^2}^{(K)} \end{bmatrix}}_{\text{boxes}} \quad \underbrace{\qquad\qquad}_{\text{probabilities}} \tag{185}$$

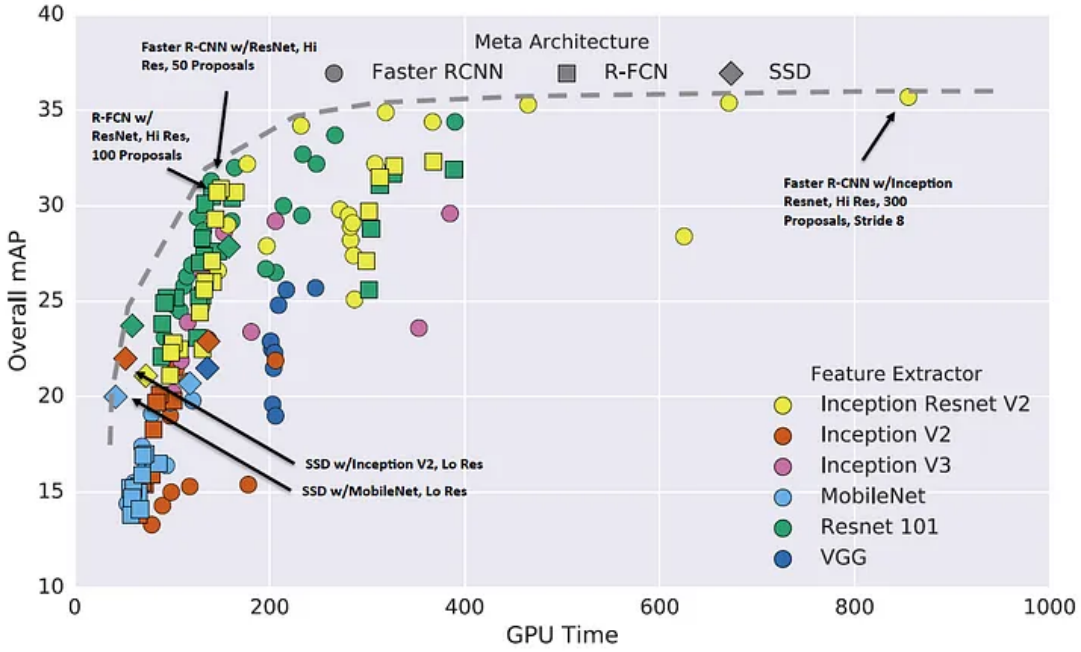
The S-by-S grid has replaced the anchor box locations proposals from the two-stage network. As before, thresholding is applied to remove low probability detections followed by non-maximal suppression.

YOLO is about ten times faster than Faster R-CNN with the same backbone CNN.

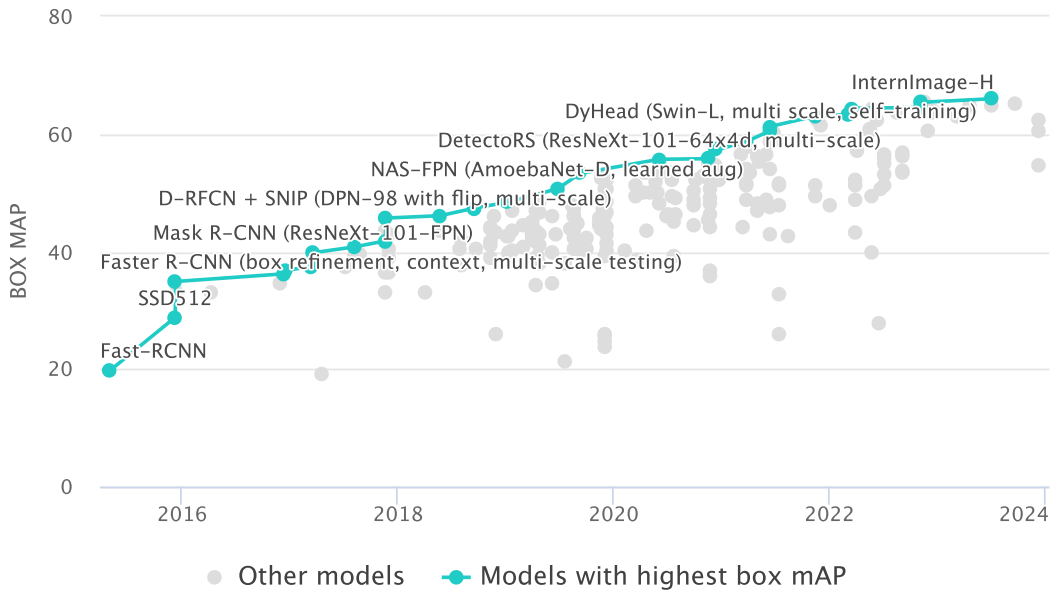
5.3 Evaluation and Progress

Of course reporting speed is not enough, we also need to report the accuracy (i.e., mAP) of a detection model and there is often a trade-off between the two. Figure 63 shows some performance graphs reported in the literature plotting the trade-off between speed and accuracy for various model architectures (left) and increasing accuracy as models have been improved over the past several years.

Note that the performance of a detector depends on many factors, including the method, the backbone convolutional neural network architecture, image resolution, training regime, datasets used for training and evaluation, thresholds used for filtering, non-maximal suppression and IoU metrics, etc. Determining the best settings and combinations is a subject of ongoing research. Always be conscious of these factors when comparing models.

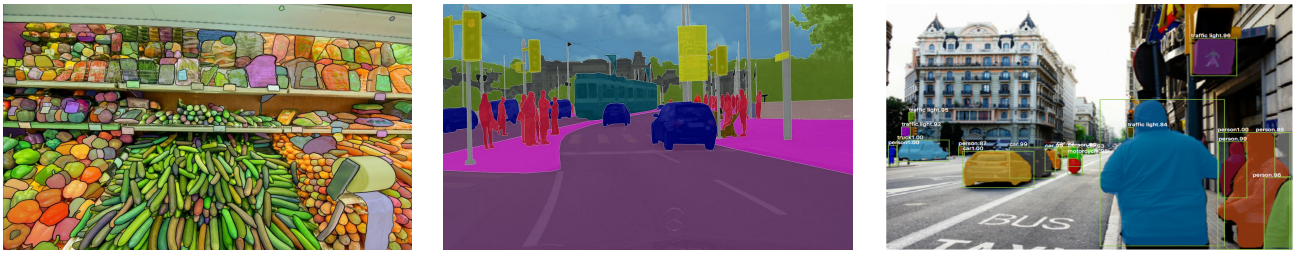


(a) (Huang et al., CVPR 2017)



(b) (`paperswithcode.com/sota/object-detection-on-coco`)

Figure 63: Performance of object detection models have steadily improved over the past several years.



(a) Detailed segmentation produced by Segment Anything [52]

(b) Semantic segmentation of pixels into classes

(c) Instance segmentation of pixels into distinct objects

Figure 64: Example of three different types of segmentation tasks.

(a) Graph based [26]

(b) SLIC [2]

Figure 65: Superpixel algorithms break an image up into many small coherent regions.

6 Image Segmentation

In this lecture we present the three main segmentation tasks in computer vision: unsupervised over-segmentation, semantic segmentation, and instance segmentation. For the latter two tasks we introduce fully convolutional networks and upsampling of feature maps through transposed convolutions. We then present the popular U-net architecture, which is the predominant architectural component for these tasks. Finally, we discuss the mask R-CNN model that builds on the object detection methods that we saw in the last lecture, and the segment anything foundation model (SAM).

Image segmentation falls under the umbrella of **pixel labelling**, where every pixel in an image is assigned a label.[16] Contrast this to image classification (in Week 4) where the entire image was assigned a single label. Pixel labels can be anything from (continuous) depth to abstract identifiers to (discrete) semantic category and combinations thereof. In image segmentation tasks include **unsupervised segmentation**, which breaks an image up into meaningful regions or superpixels and tends to be driven by low-level visual cues, **semantic segmentation**, which label each pixel with as belonging to a given class, and **instance segmentation** where each pixel is assigned an object/instance identifier. Examples of these different tasks are shown in Figure 64.[17]

6.1 Unsupervised Over-segmentation

Unsupervised over-segmentation is a method for breaking an image up into small regions of uniform colour or texture, also known as **superpixels**. There have been many different algorithms proposed in the literature and we will only present two here, neither of which are data driven and, hence, do not involve any learning (hence are unsupervised).

Felzenszwalb and Huttenlocher [26] proposed an efficient graph-based image segmentation algorithm that is the method used by Selective Search [83] for bounding box proposals in R-CNN discussed in the last lecture. In their method, the image is treated as graph $G = \langle V, E \rangle$, where pixels (and incrementally built segments) are nodes in the graph, $v_i \in V$, connected to their neighbouring pixels (or segments). The edges in the graph, E, are weighted by similarity

[16] Sometimes you will also hear the term **dense prediction**, which is essentially the same thing.

[17] While the Segment Anything model (SAM) [52] is a foundation model, which can be prompted to produce different segmentation types, the example shown in Figure 64(a) is typical of output that you would see from an unsupervised segmentation algorithm for that scene.

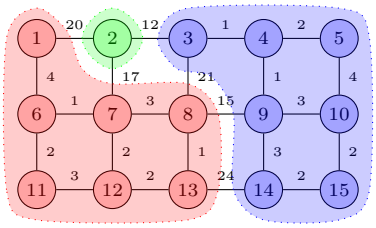


Figure 66: Toy example of graph-based segmentation [26]. Nodes are annotated with their index (i.e., identifier). Edges are annotated with their weights, w_{ij}. The graph is partitioned into segments satisfying the boundary condition, $\mathbf{diff}(C_a, C_b) > \min\{\mathbf{int}(C_a) + k/|C_a|, \mathbf{int}(C_b) + k/|C_b|\}$ for $k = 10$.

w_{ij} (where we set $w_{ij} = \infty$ if v_i and v_j are not adjacent). The graph is then partitioned into segments based on within- and between-segment similarity as shown in Figure 66.

Let $C \subseteq V$ be a segment (or component) in the graph. We define two measures,

$$\mathbf{int}(C) = \begin{cases} 0, & \text{if } |C| = 1 \\ \max_{(v_i,v_j)\in\mathbf{mst}(C)} w_{ij}, & \text{otherwise} \end{cases} \tag{186}$$

$$\mathbf{diff}(C_a, C_b) = \min_{v_i\in C_a, v_j \in C_b} w_{ij} \tag{187}$$

where $|C|$ denotes the number of nodes in C, and $\mathbf{mst}(C)$ is a minimum spanning tree of C. Note that $\mathbf{int}(C)$ is infinite if C is disconnected.

We want to find a partitioning of the graph into segments such that the following boundary condition is satisfied between any two distinct components C_a and C_b in the segmentation,

$$\mathbf{diff}(C_a, C_b) > \min\{\mathbf{int}(C_a) + \tau(C_a), \mathbf{int}(C_b) + \tau(C_b)\} \tag{188}$$

where $\tau(C) \triangleq k/|C|$ prevents degenerate solutions (e.g., singleton segments). An example partition that satisfies this condition is shown for a toy graph in Figure 66.

It turns out that a partitioning can be found that satisfies the boundary condition using a greedy algorithm that iteratively merges clusters of pixels, starting from each pixel being placed in its own cluster. The following pseudo-code outlines the algorithm, which can be implemented very efficiently using a **disjoint set** data structure, $O(|V| \log |V|)$.

function GraphBasedSegment(graph, $G = \langle V, E\rangle$, and threshold, k)
  / initialize */*
  for all nodes $i \in V$ **do**
    set pixel to its own segment, $C_i = \{i\}$ with $\mathbf{int}(C_i) = 0$
  end for
  for all edges $(i, j) \in E$ **do**
    set edge weights w_{ij} to be square distance between pixels i and j in colour space
  end for
  sort edges (i, j) by weight in non-decreasing order
  / merge clusters */*
  for all edges (i, j) **do**
    let $C_i \ni v_i$ and $C_j \ni v_j$
    if $C_i \neq C_j$ and $w_{ij} \leq \min\left\{\mathbf{int}(C_i) + \frac{k}{|C_i|}, \mathbf{int}(C_j) + \frac{k}{|C_j|}\right\}$ **then**
      merge C_i and C_j
      set $\mathbf{int}(C_i \cup C_j) = w_{ij}$
    end if
  end for
  return segmentation $S = (C_1, C_2, \ldots)$
end function

A worked example showing each step of the algorithm for the previous toy example is shown in Figure 67.

The graph-based algorithm of Felzenszwalb and Huttenlocher [26] produces irregularly shaped regions as can be seen in the example in Figure 65. An alternative algorithm that produces more regular shaped superpixels, and is also very efficient, is the **simple linear iterative clustering** (SLIC) algorithm of Achanta et al. [2]. In their approach k-means clustering is repeatedly applied, limited to a local neighbourhood in a $2S \times 2S$ patch around each centroid,

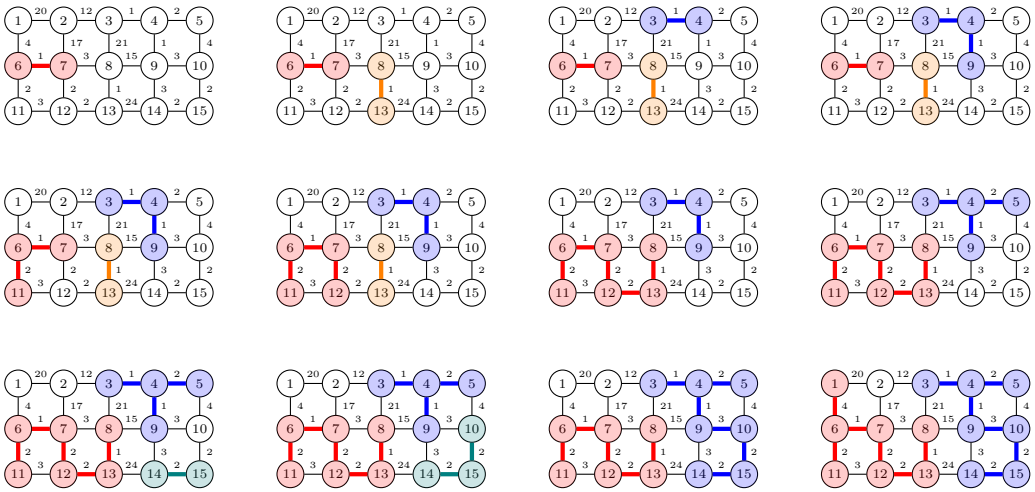


Figure 67: Worked example graph-based segmentation algorithm of Felzenszwalb and Huttenlocher [26] for a 5-by-3 image with 4-connected neighbourhood and $k = 10$. The algorithm proceeds from left-to-right then top-to-bottom choosing the lowest cost edge to merge segments. Where there are multiple edges all with the same lowest cost, one is chosen arbitrarily. Edges are annotated with similarity score. Segment membership is indicated by colour and the minimum spanning tree indicated using bold edges.

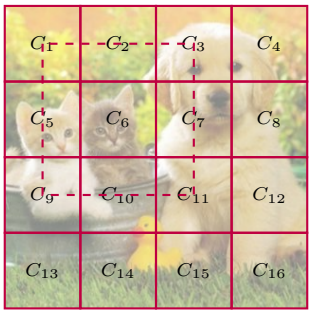


Figure 68: Illustration of local neighbourhood clustering within the SLIC [2] algorithm. Any pixel within the dashed region can be assigned to cluster C_6, allowing pixels to move from the eight neighbouring clusters C_1, C_2, etc.

with $S = \sqrt{WH/K}$ where K is the number of desired superpixels. Clustering is done using a distance metric defined on 5-dimensional colour and spatial features, (l, a, b, x, y) as

$$d(i,j) = \sqrt{(l_i - l_j)^2 + (a_i - a_j)^2 + (b_i - b_j)^2} + \frac{m}{S}\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{189}$$

where m controls the compactness of the superpixels, trading off spatial versus colour distances. The algorithm is summarized by the pseudo-code below and illustrated in Figure 68.

```
function SLIC(grid size, S)
    /* initialize */
    set cluster centroids C_k by sampling pixels on a regular S × S grid      ▷ (avoids putting centroid on
    move centroids to lowest image gradient in 3 × 3 neighbourhood            ▷ an edge or noisy pixel)
    /* iterate */
    repeat
        assign best matching pixel from 2S × 2S neighbourhood for each cluster k
        recompute centroid C_k for each cluster
    until no change (or maximum iterations reached)
    /* finalise */
    tidy up orphaned and small clusters
    return superpixels (C_1, C_2, ...)
end function
```

6.2 Semantic Segmentation

Think back to the task of object detection from the last lecture. The definition of the task requires that we place a bounding box around objects found in the scene. But this presents a problem for detailed analysis because it still

(a) problem (b) solution

Figure 69: One problem with object detection is that it does not fully specify which pixels belong to each object found in the scene (left). A proposed solution is semantic segmentation—annotate every pixel in the image with a category label (right).

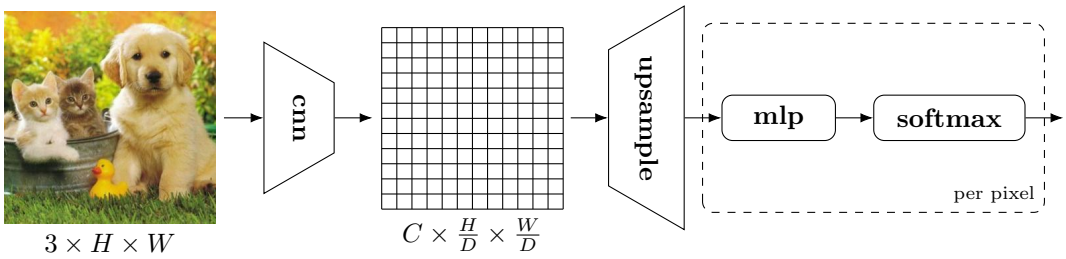


Figure 70: A fully convolutional network (FCN) upsamples feature maps from a convolutional neural subnetwork so that it can label every pixel in an image. The per-pixel multi-layer perceptron is implemented using 1-by-1 convolutions allowing the image to be of arbitrary size.

leaves ambiguous which pixels within the bounding box belong to the object of interest as shown in Figure 69. Some pixels will belong to the background (or unknown objects), and in cases of overlapping bounding boxes, some pixels will belong to other objects. Semantic segmentation aims to solve this problem by assigning a category label (from a predefined set of known categories) to every pixel in the image. Note that semantic segmentation does not distinguish between different instances of the same object category so only partially addresses the issue of ambiguity caused by overlapping bounding boxes (i.e., when the categories are different).

Common datasets used for semantic segmentation research include MSRC [17], COCO [61], and Cityscapes [16], all of which have had their pixels painstakingly annotated by human labelers.

We saw in the last two lectures that convolutional neural networks are very good for processing images. However, the pooling layers reduce the feature map size so that we do not have a one-to-one correspondence between features and image pixels. For example, we may start with an $H \times W$ colour image and end up with a C-channel $\frac{H}{D} \times \frac{W}{D}$ feature map, where $D > 1$ is the downsampling factor. This is a problem if we want to classify pixels individually. One solution, illustrated in Figure 70, is to **upsample** the feature maps back to the original resolution of the image. The resulting C-dimensional features at each feature map location can then be passed through a multi-layer perceptron (MLP) to obtain per-pixel classifications. Long et al. [63] showed that these per-pixel MLPs can be implemented efficiently using 1-by-1 convolutions allowing for arbitrary sized images to be processed.

There are several options for upsampling the feature maps as shown in Figure 71. The simplest approach is nearest neighbour (also known as piecewise constant) upsampling. Here each element in the upsampled feature map takes its value from the closest location in the downsampled map. Treating the indices of elements in the downsampled map as integers and those in the upsampled map as fractional, we can write nearest neighbour upsampling as

$$f^{\text{nearest}}(x, y) = f_{\lfloor x \rceil, \lfloor y \rceil} \tag{190}$$

where the operator $\lfloor \cdot \rceil$ rounds its argument to the nearest integer. So, for example, the $(1.5, 1)$-th element lies halfway between the $(1, 1)$-th and $(2, 1)$-th elements, and takes the value of the $(2, 1)$-th element.[18] If, in the case of max pooling, we happened to remember the index of the maximum element when downsampling then we can simply restore the value to that location when performing upsampling, and setting the remaining elements to zero. This is known as unpooling and requires some housework in storing the maximizing indices in addition to the feature maps throughout the convolutional neural network layers. Yet another option is to perform bilinear interpolation,

$$f^{\text{bilinear}}(x, y) = \begin{bmatrix} \lceil x \rceil - x \\ x - \lfloor x \rfloor \end{bmatrix}^T \begin{bmatrix} f_{\lfloor x \rfloor \lfloor y \rfloor} & f_{\lfloor x \rfloor \lceil y \rceil} \\ f_{\lceil x \rceil \lfloor y \rfloor} & f_{\lceil x \rceil \lceil y \rceil} \end{bmatrix} \begin{bmatrix} \lceil y \rceil - y \\ y - \lfloor y \rfloor \end{bmatrix} \tag{191}$$

over a 2-by-2 grid as discussed in the previous lecture, or bicubic interpolation, which is popular in photo editing tools when resizing images, but requires a 4-by-4 grid.

[18] Here we have chosen to round 0.5 up to the nearest integer.

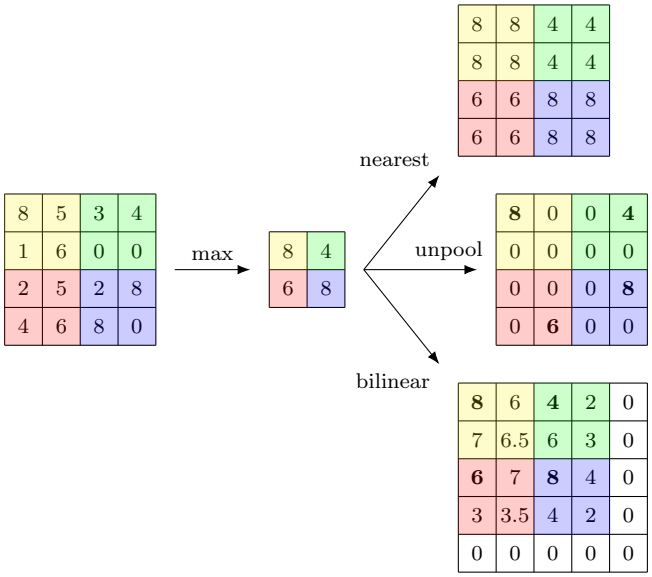


Figure 71: Upsampling can be done using fixed functions such as nearest neighbour interpolation, unpooling, or bilinear interpolation. If we think of the elements in the downsampled feature map as indexed by $\{(i,j) \mid i,j \in \{1,2\}\}$ then the upsampled feature maps are indexed by $\{(i,j) \mid i,j \in \{0.5,1,1.5,2\}\}$.

A more flexible approach is to learn the upsampling function. In deep learning this is done via **transposed convolutions**, sometimes called deconvolution. Note that this is not the same as a mathematical inverse. Let $x \in \mathbb{R}^n$ be an input signal, $a \in \mathbb{R}^p$ be a filter kernel, and $s \geq 1$ be a stride. Mathematically, we can write the transposed convolution operator as

$$y_i = \sum_{j=1}^{p} a_j x_{\frac{i-j+s}{s}} \qquad \text{for } i = 1, \ldots, s(n-1)+p \tag{192}$$

where $x_{\frac{i-j+s}{s}}$ is defined to be zero if the index is fractional or greater than n.

The operation is perhaps easier to understand through a worked example. There are two ways the computation for transposed convolution defined above can be implemented. Figure 72 shows a worked example for filter $a = (1, -2, 3)$ with stride of 2 using the reverse strided convolution method. Figure 73 shows the same operation but implemented using the scale, shift and sum (or accumulate) method. Both methods produce the same result—as must be the case. Observe also that the output signal is larger than the input signal.

Just like standard convolutions, we can view transposed convolution as a linear operator. Recall that for a one-dimensional convolution (with stride = 1) we have as a matrix equation,

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-p+1} \end{bmatrix} = \begin{bmatrix} a_1 & \ldots & a_p & 0 & \ldots & 0 \\ 0 & a_1 & \ldots & a_p & \ldots & 0 \\ \vdots & & \ddots & & \ddots & \vdots \\ 0 & \ldots & 0 & a_1 & \ldots & a_p \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{193}$$

We can similarly write the one-dimensional transposed convolution (with stride = 1) as,

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+p-1} \end{bmatrix} = \begin{bmatrix} a_1 & 0 & \ldots & 0 \\ \vdots & a_1 & & \vdots \\ a_p & \vdots & \ddots & 0 \\ 0 & a_p & & a_1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_p \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{194}$$

In compact form we have, $y^{\text{conv}} = Ax$ and $y^{\text{trans}} = A^T x$, and hence the name transposed convolution.

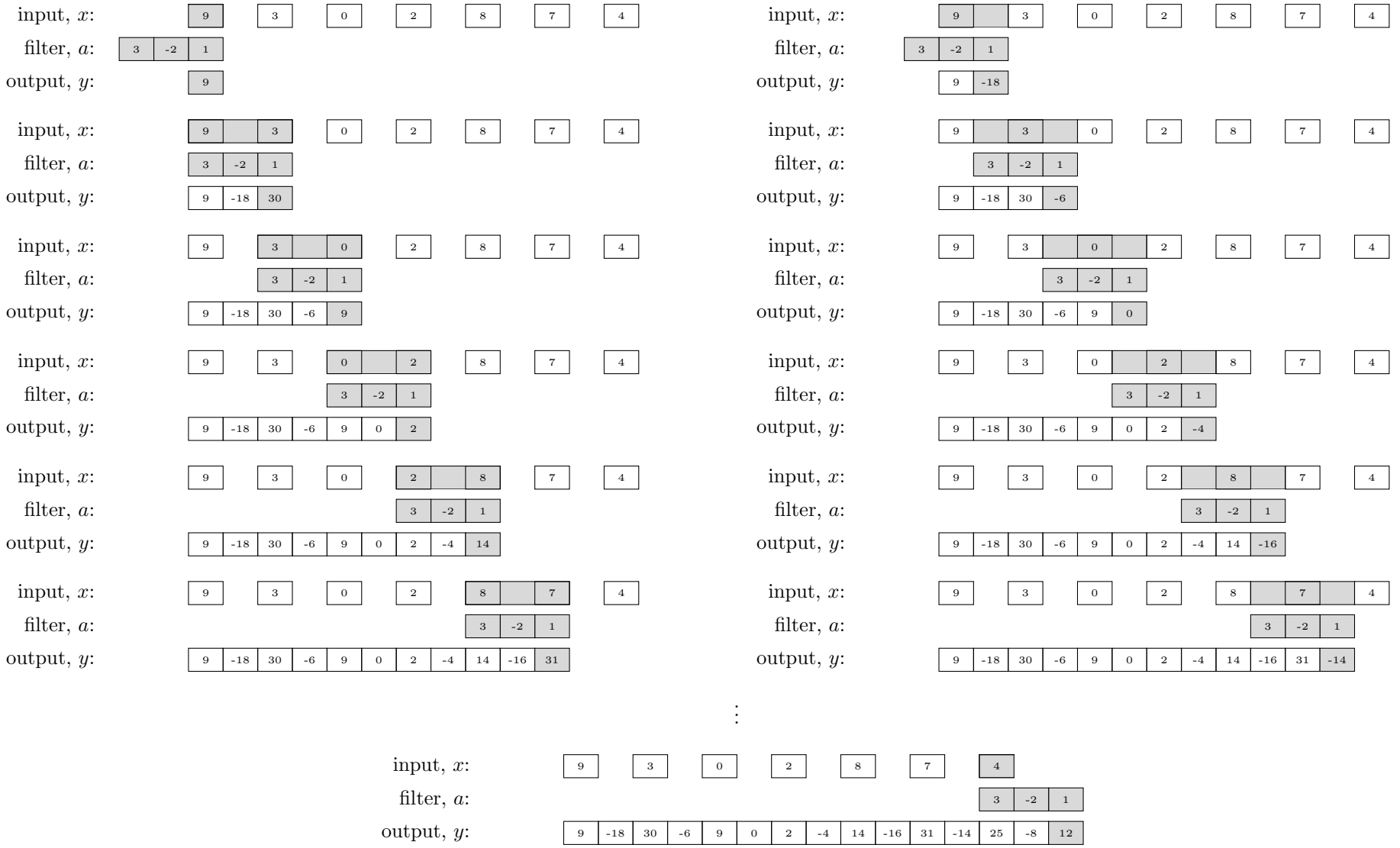


Figure 72: Worked example of a one-dimensional transposed convolution operation as reverse strided convolution.

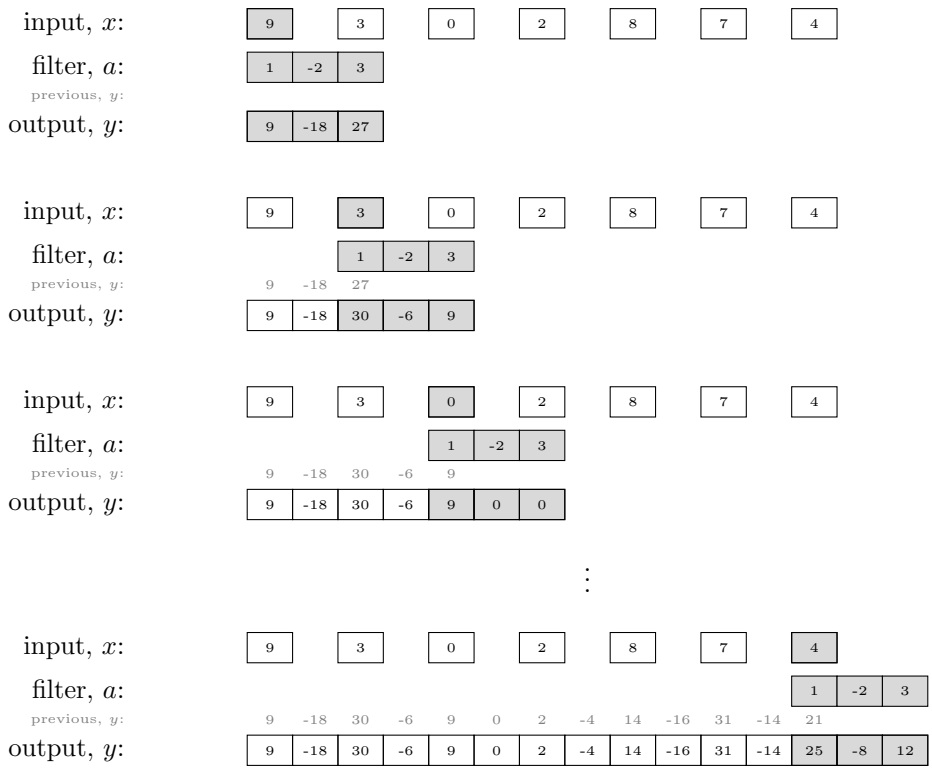


Figure 73: Worked example of a one-dimensional transposed convolution operation by scaling, shifting and summing.

Long et al. [63] proposed three variants of the fully convolutional network (FCN) architecture as shown in Figure 74. All variants apply learned upsampling (i.e., transposed convolutions) to different pooling layers for pixelwise prediction. The first variant, FCN-32s, applies upsampling to the fifth pooling layer, requiring 32 times upsampling. The second variant, FCN-16s, applies two times upsampling to the seventh convolutional layer, which is also then added to the forth pooling layer. The resulting sum is then upsampled 16 times to get back to the original image size. The last variant, FCN-8s, adds the (upsampled) seventh convolution layer and forth pooling layer to the third pooling layer, which is then upsampled eight time to reach the original image size. In summary, the FCN architecture predicts pixel labels on upsampled feature maps from different pooling layers. These predictions are then fused via pixelwise summation as a way of averaging coarse and fine level predictions.

A similar yet more flexible approach combines feature maps of the same size from the downsampling branch with the upsampling branch, and only makes predictions on the combined feature maps of the final layer. Consider a sequence of downsampled and upsampled feature maps

$$F_1^{\mathrm{E}} \xrightarrow{\text{down}} F_2^{\mathrm{E}} \xrightarrow{\text{down}} \cdots \xrightarrow{\text{up}} F_2^{\mathrm{D}} \xrightarrow{\text{up}} F_1^{\mathrm{D}} \tag{195}$$

with $\mathbf{shape}(F_\ell^{\mathrm{E}}) = \mathbf{shape}(F_\ell^{\mathrm{D}})$. The so-called **skip connections** compute $F_{\ell-1}^{\mathrm{D}}$ using $F_\ell^{\mathrm{D}} \oplus F_\ell^{\mathrm{E}}$ instead of just F_ℓ^{D}, where $\oplus$ denotes concatenation along the feature dimension. This propagates contextual information from the "encoder" to the "decoder" layers of the network as illustrated schematically in Figure 75.

The very popular U-Net [78] architecture, shown in Figure 76, uses this idea but does not require the feature maps to be the same size. Indeed, the feature maps on the encoder side are larger and than on the decoder side and the skip connection crops the larger feature map before concatenating with the smaller one. The network was originally proposed for medical image analysis but is now the backbone in many pixel-to-pixel computer vision tasks including semantic segmentation, image denoising, and image generation.

6.3 Instance Segmentation

We now return to the problem highlighted earlier that semantic segmentation does not distinguish between different instances of the same object type. See, for example, Figure 77. Various solutions of similar flavour have been proposed, but the main idea is to jointly perform object detection and semantic segmentation. Gould et al. [33] proposed an early version of this predating deep learning object detection and semantic segmentation models. A more modern approach builds on the R-CNN architecture with a model called Mask R-CNN [40], which for each bounding box proposal adds a binary pixel segmentation branch to mask out pixels not belonging to the object. This is in addition to the class prediction and bounding box refinement multi-layer perceptrons. The complete architecture is shown in Figure 78.

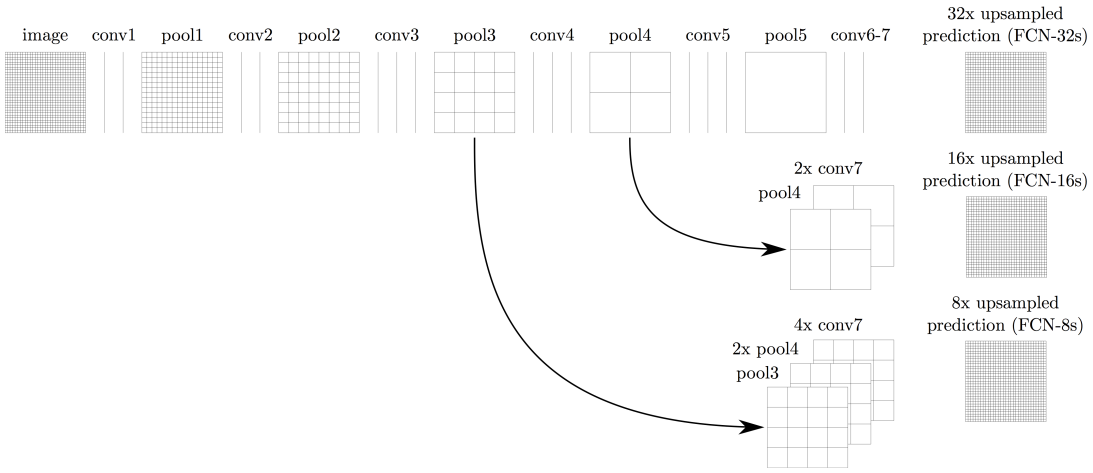


Figure 74: The fully convolutional network (FCN) architecture of Long et al. [63].

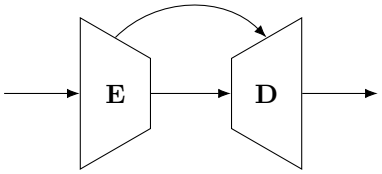


Figure 75: Skip connections combine feature maps of equal size from the downsampled and upsampled network paths.

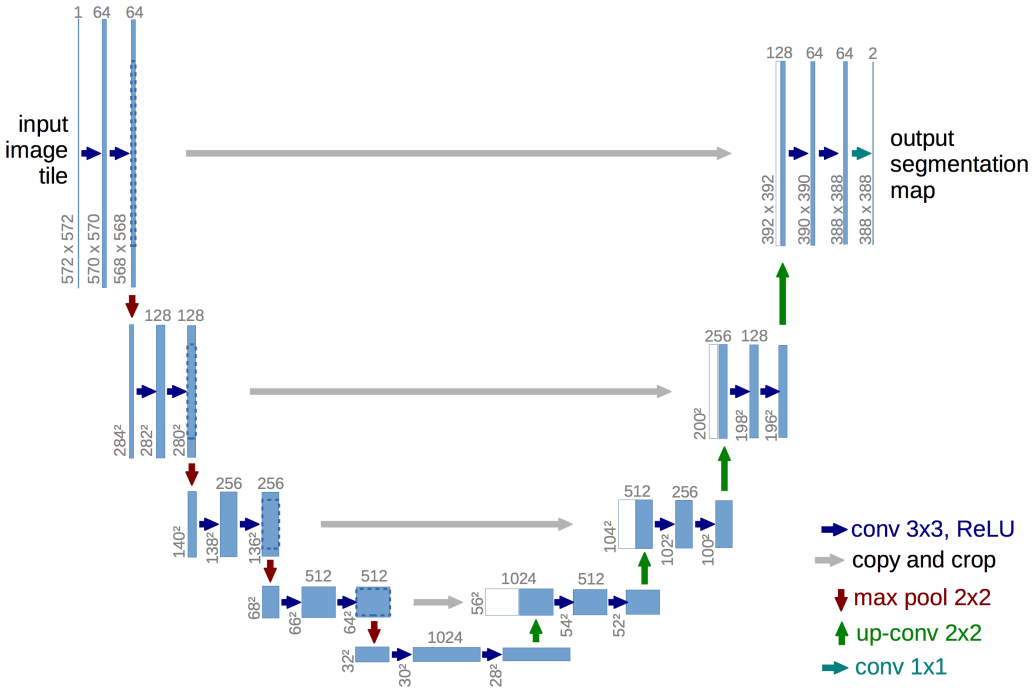
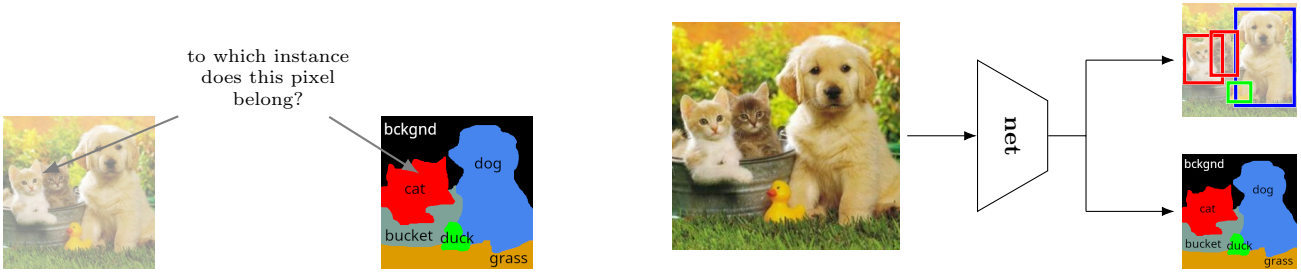


Figure 76: The U-Net architecture proposed by Ronneberger et al. [78] uses skip connections to propagate information from the encoder to the decoder layers.



(a) problem (b) solution

Figure 77: One problem with semantic segmentation is that it does not distinguish between instances of the same class (left). A proposed solution is instance segmentation—perform object detection and semantic segmentation jointly (right).

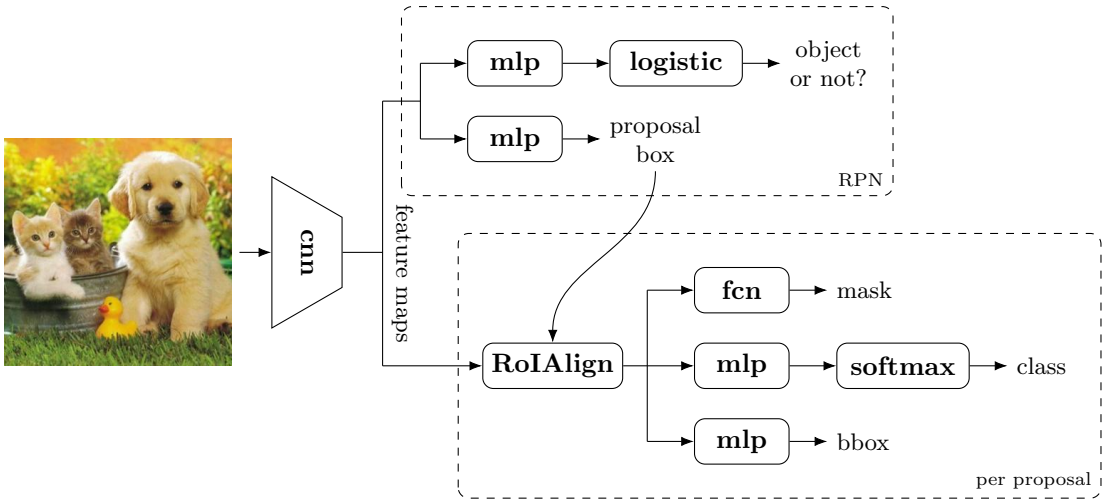


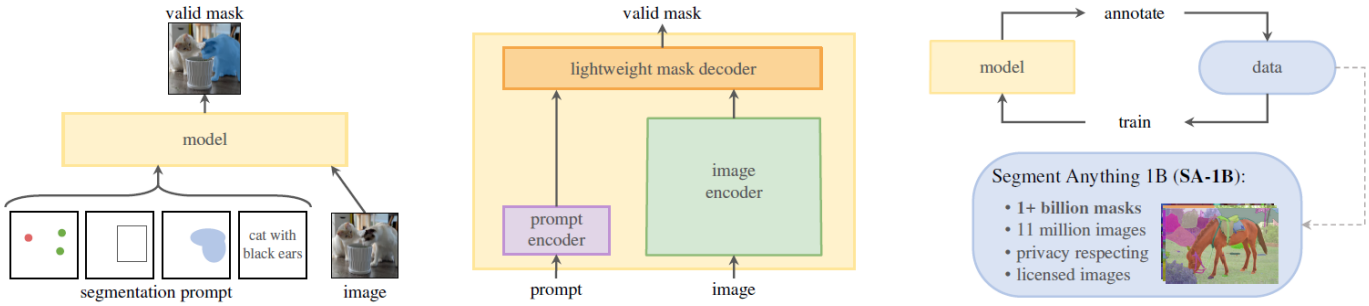
Figure 78: Mask R-CNN [40] architecture.

Some researchers like to distinguish between instance segmentation—as the task of labeling unique instances—from panoptic segmentation [51], which labels both instances and class types. The main difference from Mask R-CNN is that the background regions (such as sky, trees, grass, road, buildings, and water) are also labeled.

6.4 Segment Anything

Very recently Kirillov et al. [52] introduced a model for various image segmentation tasks called Segment Anything. The model can run off-the-shelf or be provided with different types of prompts, including foreground/background hints and text queries (see Figure 79). It was trained on a very large set of images and based on the ViT architecture, which will be discussed in later lectures. Training took approximately three full days on a cluster of 256 A100 GPUs. The fact that it can address a host of segmentation tasks and amount of training data put in the class of **foundation models**—i.e., a model that is trained at scale and can be adapted to a wide range of tasks.

The training set used by Segment Anything includes 1B pseudo-ground-truth masks and 11M images. This set was bootstrapped from a smaller set of 4.3M human annotated masks and 120k images followed by a further automatically generated 5.9M masks on 180k images that were carefully filtered to eliminate poor pseudo-labels.



(a) **Task**: promptable segmentation (b) **Model**: Segment Anything Model (**SAM**) (c) **Data**: data engine (top) & dataset (bottom)

Figure 79: Segment Anything Model [52].

7 Developing, Debugging and Diagnosing

This lecture is unlike the others. We discuss various issues you may face when developing machine learning algorithms. We will highlight some common bugs, implementation concerns and design issues, and present strategies for diagnosing performance problems. The lecture may be delayed until later in the course.

7.1 Numerical Issues

Numerical calculations on a computer are always subject to errors and deep learning is no exception—deep learning algorithms are full of numerical calculations. One of the biggest sources of numerical error is due to limited precision representations and arithmetic. For example, what is the result of computing `255 + 1`? Surprising to people not familiar with how computers work, the answer can vary depending on how the numbers are represented. If we use an 8-bit unsigned integer representation then the answer is zero! You can verify this for yourself by running the following Python code, which adds one to numbers zero, one, forty-one, and two hundred and fifty-five.

```
import numpy as np

x = np.array([0, 1, 41, 255], dtype='uint8')
x += 1
print(x)
```

Of course moving to floating-point representations does not remove the problem of numerical precision, it simply shifts it elsewhere as the following example shows—what is `16,777,216 + 1`? The answer, if we're using 32-bit IEEE floating-point format, is `16,777,216`. We have not performed additional at all!

```
import numpy as np

x = np.array([16777216], dtype='float32')
x += 1
print(x)
```

You may think that such examples are pathological and never really occur in practice. You would be wrong. Consider the very common task in deep learning of applying a convolutional filter to an image. If the image is represented in 8-bit format (as is typical when loaded from storage), then the filtering calculations are susceptible to these types of numerical **overflow** and **underflow**.

Other numerical issues can arise from algorithmic limitations such as not being able to generate true random numbers, which to be honest is hardly an issue in practice for deep learning models.[19] How an algorithm or calculation is implemented can also cause unexpected issues. Consider, for example, computing the empirical standard deviation of a set of scalar samples $\{x_i \in \mathbb{R}\}_{i=1}^n$, defined as,

$$\hat{\sigma} = \sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{n-1}} \tag{196}$$

where $\mu = \frac{1}{n}\sum_{i=1}^n x_i$ is the empirical mean. This calculation requires two passes through the data—once to compute the mean, and then again to compute the standard deviation using the mean.

A seemingly better approach is to perform equivalent calculation,

$$\hat{\sigma} = \sqrt{\frac{n\sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i\right)^2}{n(n-1)}} \tag{197}$$

which only requires one pass through the data. However, the one-pass approach is fraught with danger. If the x_i are all large or the sum of the x_i is large, then we risk overflowing the numerator, since the square of a large number is a much larger number. The two-pass approach mitigates against this type of numerical overflow by centering the data about the mean before squaring, and is hence more numerically stable.

A naive implementation of **softmax** for a vector $z \in \mathbb{R}^K$,

$$\textbf{softmax}(z) = \left(\frac{\exp z_1}{Z}, \ldots, \frac{\exp z_K}{Z}\right) \tag{198}$$

[19] This is more of an issue in the field of cryptography.

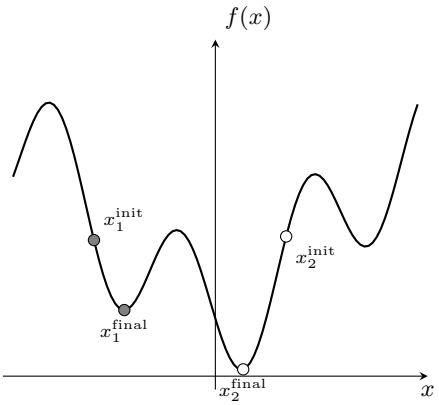


Figure 80: Different random initialization can result in different solutions. You always need to make sure that what you are claiming is due to a real statistical effect and not pure luck such as a favourable random seed.

where $Z = \sum_{k=1}^{K} \exp z_k$, has the same problem of being susceptible to numerical underflow and overflow. Fortunately a simple trick can make the calculation numerically stable. Let $z_{\max} = \max\{z_1, \ldots, z_K\}$. Then implementing softmax as

$$\textbf{softmax}(z) = \left(\frac{\exp(z_1 - z_{\max})}{Z}, \ldots, \frac{\exp(z_K - z_{\max})}{Z} \right) \tag{199}$$

where $Z = \sum_{k=1}^{K} \exp(z_k - z_{\max})$ gives the same result mathematically (i.e., under infinite precision) but in a much more robust way. It should be easy to convince yourself that the calculations are mathematically equivalent since the quantity $\exp(-z_{\max})$ cancels in the numerator and denominator, i.e.,

$$\frac{\exp(z_i - z_{\max})}{Z} = \frac{\exp(z_i)\exp(-z_{\max})}{\sum_{k=1}^{K} \exp(z_k)\exp(-z_{\max})} = \frac{\exp(z_i)\exp(-z_{\max})}{\exp(-z_{\max})\sum_{k=1}^{K} \exp(z_k)} = \frac{\exp(z_i)}{\sum_{k=1}^{K} \exp(z_k)} \tag{200}$$

Stability comes from the fact that terms $\exp(z_i - z_{\max})$ are always less than or equal to one, with equality holding for at least one or them (i.e., for all i such that z_i is a maximizer, $z_i = z_{\max}$).

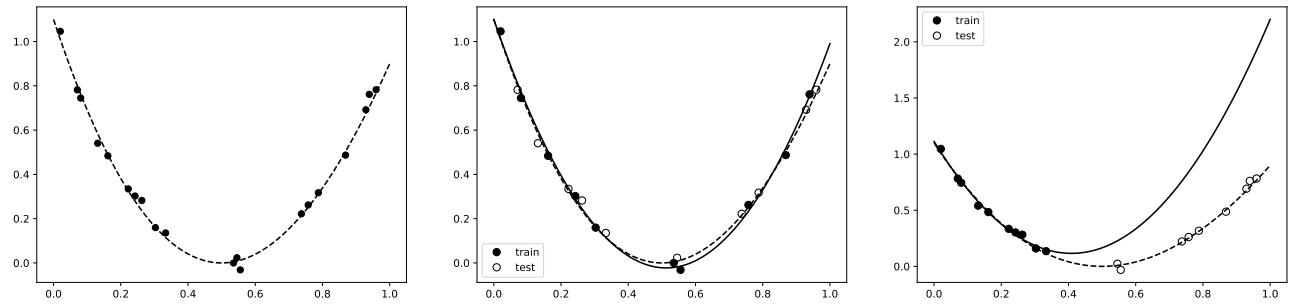
The above examples have demonstrated that mathematical equivalent calculations does not necessarily mean computational equivalence. Calculating the same thing in different ways can result in very different results due to numerical issues. But numerical issues can also simply be caused by bugs in the code, i.e., erroneous implementations. If you're lucky this will result in divide-by-zero exceptions or not-a-number values, which are easy to detect. Unfortunately, however, bugs in machine learning algorithms can be very difficult to detect as we will show later in the lecture.

7.2 Algorithmic Issues

Other issues that we should be aware of are algorithmic issues, including poor experiment design (e.g., choosing a favourable random seed), systematic error from mini-batch sampling or division into train-test split, and leaking information from test set during training. Let's consider these one at a time.

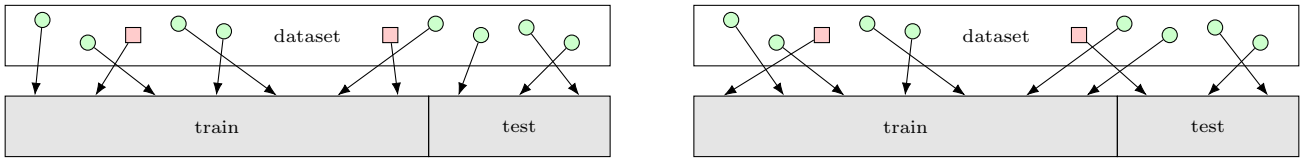
Objective functions in deep learning are, in general, non-convex so we cannot hope to optimize them fully. Indeed, we are often satisfied with a local optimum of our loss functions. And even then, a better value of the loss function on the training set does not always translate to better performance on the test set. Putting the problem of generalization to the side for the moment, consider the task of minimizing the function f shown in Figure 80. Providing that the function is sufficiently smooth and we take small enough steps, gradient descent will take us from the initial point to the nearest local minimum. As such different initializations will lead to different solutions, and it is not hard to see that a lucky random seed can favour or penalize one model over another. To mitigate against this it is important to always repeat experiments initializing with different random seeds and report statistical results (mean and standard deviation over performance metrics). You should also try to run controlled experiments when comparing different variants of a model (e.g., by changing one thing at a time and running the variants with the same random seeds, which has the added advantage of making your results reproducible).

Data sampling strategies can have a big influence on results. A key tenet in machine learning is that the training sample distribution should match test sample distribution (or population distribution). If this is not the case then the learned model will fail to generalize to unseen data. Figure 81 illustrates this idea. Given a set of noisy samples for curve fitting (a), if we split the samples so that they are distributed well over the input domain (b), then the model will generalize well as indicated by its performance on the hold-out test set. However, if we sample in a way that covers only a portion of the input space (c), then the model will perform poorly at test time.



(a) noisy samples (b) good train/test split (c) poor train/test split

Figure 81: Example of two different sampling strategies and their effect on regression.



(a) random sampling (b) stratified sampling

Figure 82: Sampling strategies for unbalanced data. Stratified sampling ensures that the empirical training class distribution and test class distribution are approximately equal.

It is important that sampling strategies are designed for the given application. For example, when curve fitting it may be appropriate to alternate every second data point between the training and test sets. But this would be a very poor strategy when sampling frames from a video, for example, since there is very high correlation between consecutive frames and a model trained on such high-dimensional data may end up simply over-fitting to the training data. The high correlation between train and test sets then gives an overly optimistic estimate of generalization performance. A better strategy in this case is to split data at the video level rather than the frame level.

A similar thing can happen during stochastic gradient descent if we don't shuffle data within the training set between epochs. The model, repeatedly seeing data in the same order, will bias performance to data seen at the end of the epoch rather than the start. This is why we always reshuffle data[20] and apply different data augmentations when training models over multiple epochs.

Another sampling problem occurs when data is not balanced across categories in classification problems. In such situations rare categories may have very different distributions between train and test splits, violating the previously mentioned tenet underlying machine learning. Here a common solution is to use **stratified sampling**, which is to randomly split data between training and test sets sampling within each class separately rather than the entire dataset globally. The idea is illustrated in Figure 82. Note that there is still imbalance between the classes, but at least the training and testing distributions match. The approach can also be applied when constructing mini-batches during stochastic gradient descent. To deal with class imbalance, reweighting of rare samples in the loss function or sampling with replacement (combined with data augmentation) can also be used.

Test data should never be seen during training or development of the model. This is very difficult to achieve in practice where standard benchmarks use fixed test sets and there is a natural tendency to build on models that work well based on performance reported on the test set. Nevertheless, we can ensure that the training algorithm does not have access to information about the test set. An area where this is often overlooked and hence a common source of **information leakage** is when pre-processing data. For example, when whitening input data, the mean and standard deviation should be computed using the training data only not the entire dataset. Normalization using these statistics are then applied to examples across the whole dataset.

Another common mistake is not putting models into **evaluation mode** at test time. For example, `BatchNorm`, should use learned parameters on test data, not parameters estimated from the test batch. This is done using the `.eval()` method on the model, and instructs the model to keep parameters fixed. For most tasks, the test data is assumed to be independently and identically distributed. Moreover, it should not matter in which order the data is processed, or whether each example is processed individually or in a batch. Figure 83 demonstrates why this may matter if we have not put the model into evaluation mode. The figure depicts a batch of positive and negative examples. With `BatchNorm` in training mode, the data will be normalized to have zero mean *with respect to the batch*. This may unfairly favour prediction of samples in the batch. Information from the test set has leaked into the evaluation process. On

[20]Note that this applies only within the training set. We do not need to shuffle the order of data in the test set, nor do we change the train/test split during an experimental run.

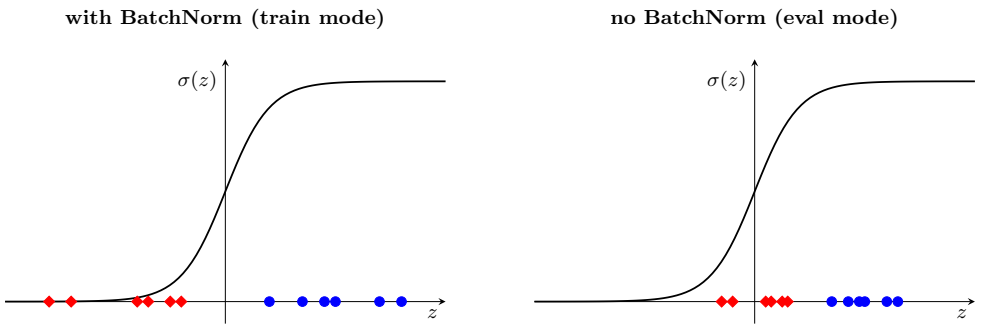


Figure 83: Information leakage. We must be careful to avoid parameters or statistics used by the model to use information from the test set. One common mistake is forgetting to put the model into evaluation mode when running on validation and test sets.

the right of the figure we see the scenario where we have set `BatchNorm` to evaluation mode. No information leakage occurs in this case.

7.2.1 Bugs in Machine Learning Algorithms

Bugs in machine learning algorithms can be notoriously difficult to find. This is because machine learning algorithms are inherently trying to optimize for good solutions, so if an algorithm is making progress towards a solution (i.e., minimizing a loss function) then it appears to be working as expected. Consider, for example, trying to minimize the following scalar-valued function

$$f(x) = 10x^4 + x^2 \tag{201}$$

We will use a damped Newton's method, which is a second-order optimization algorithm with updates

$$x \leftarrow x - \eta \frac{f'(x)}{f''(x)} \tag{202}$$

To illustrate the difficulty of debugging, let us introduce a small bug into our implementation of the first- and second-order derivatives. The true derivatives are

$$f'(x) = 40x^3 + 2x \tag{203}$$

$$f''(x) = 120x^2 + 2 \tag{204}$$

but we may accidentally implement our model with the incorrect sign on the second term,

$$\tilde{f}'(x) = 40x^3 - 2x$$

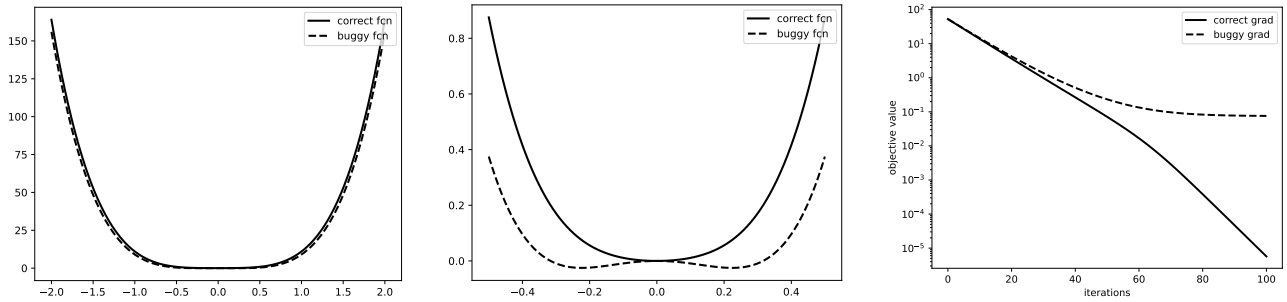
$$\tilde{f}''(x) = 120x^2 - 2$$

This sort of bug is not uncommon in numerical applications, and in this example would correspond to the function, $\tilde{f} = 10x^4 - x^2$, which on a macroscopic scale looks very similar to the true function, f, as shown in Figure 84(a). It is only when we zoom in around the minimum that we see a difference (see Figure 84(b)). The manifestation of this bug, is that when we try to optimize using the buggy implementation we fail to converge to the true optimal solution as illustrated when we compare learning curves of the correct implementation against the buggy one in Figure 84(c).

The take home message from this example is to use automatic differentiation wherever possible. More generally, however, produce lots of plots and perform lots of tests. We will discuss some useful diagnostics that will help discovering bugs later in the lecture.

Sometimes you may think you have a bug in the code, but the result of your algorithm is quite reasonable for what your are asking it to do. A very good example is the pathological case of fitting a mixture of Gaussians as illustrated in Figure 85. The aim of fitting a mixture of Gaussians is to represent the empirical distribution defined by a set of samples by the weighted combination of Gaussian probability distributions. This is often used to identify clusters in the data under the assumption that each cluster can be well represented by a single Gaussian. Fitting the mixture solves the following optimization problem for a set of data points $\{x^{(i)} \in \mathbb{R}^n\}_{i=1}^N$ and K mixture components,

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^N \log \underbrace{\sum_{k=1}^K \pi_k \mathcal{N}(x^{(i)}; \mu_k, \Sigma_k)}_{p(x^{(i)};\theta)} \\ \text{subject to} \quad & \sum_{k=1}^K \pi_k = 1,\ \pi_k \geq 0 \end{aligned} \tag{205}$$



(a) full function (b) zoomed in function (c) learning curve

Figure 84: Bugs in machine learning algorithms can be hard to find. Here a small error in a derivative calculation can prevent an optimization algorithms from converging to the true solution. However, without reference to the correct implementation, the algorithm looks like it is making progress.

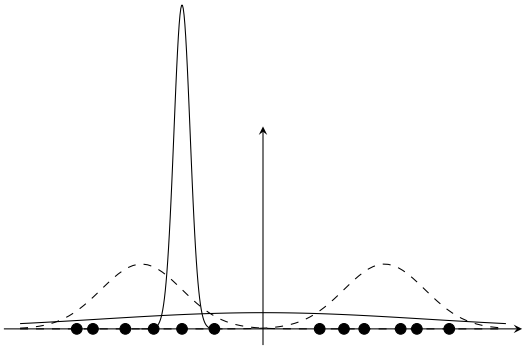


Figure 85: Pathological case in fitting a mixture of Gaussians model that maximizes the likelihood by placing a very narrow Gaussian on one data point and a wide Gaussian to capture the remainder (solid). But this is not be a desirable solution for representing the data or generalizing to unseen samples (dashed).

where π_k, μ_k, and Σ_k are parameters of the model. That is, we find the parameters that maximize the log-likelihood of the data when represented by a mixture of Gaussians.

It turns out that the true optimal solution to this problem is to place a very narrow Gaussian over one of the data points so that it's likelihood approaches infinity and spread the remaining Gaussian components over the rest of the data. While this maximizes the log-likelihood it is, unfortunately, not the solution that we really want. Most of the time, if we initialize randomly and use gradient descent (or an alternative optimization algorithms known as expectation-maximization [19]), we will converge to a local optimum, which is more desirable.

All this is to say that if your algorithm results weird output it may not be because of a bug. Ask yourself if what you are seeing is reasonable, or just possible, based on what you are asking the machine to do. The simple example above of fitting a mixture of Gaussian is the exact same thing that occurs in a phenomenon known as **mode collapse** in self-supervised deep learning.

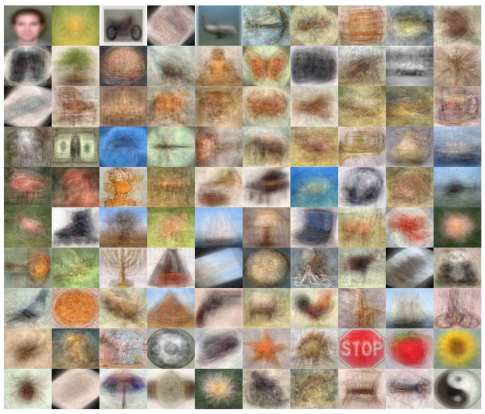
7.3 Implementation Tips

The following is a brief set of tips and standard practices that will help avoid algorithmic issues and bugs, or make it easier to debug when they inevitably occur. Perhaps the most important thing to remember when writing code of any sort is that **software is written for people, not machines**. That is, your software source code should be clearly structured, well commented, and easy to understand by other researchers or software developers (including a future version of yourself). Telling yourself that your code is being written for another person to understand, not for a machine to understand, will to along way to achieving this goal, and help enormously in avoiding or debugging problems when they do occur.

To catch bugs as early as possible it is always a good idea to check for `NaN` and `Inf`. These indicate that something has gone wrong in a numerical calculation (e.g., divide-by-zero), which render any subsequent calculations or model updates invalid. Similarly, always assert pre- and post-conditions of methods. For example, check that tensors match the size that you expect. When writing research code, it is better that your code crashes than produces invalid results.

When debugging code, try to write the simplest test case that will expose the bug. This not only rules out confounding factors and helps to rapidly narrow down the source of the error, but also gives you a free regression test for the future. It is also easier to share your test case with other developers and collaborators who may help you resolve the problem.

(a) Near duplicate images in MSRC [17]



(b) Per-class average images in Caltech-101 [59]

Figure 86: Every dataset is biased.

In a similar vain, use existing well-tested code where possible. The fact that other people have looked over and used the code means that it is more likely to be correct. We have already mentioned the many excellent deep learning frameworks—PyTorch, TensorFlow, and JAX. You would have to have a very good reason to write your own deep learning library. Often you'll be extending someone else's deep learning model, and hopefully they have shared code that implements their model. However, don't blindly trust third-party code. Make sure you can reproduces published results before extending the code. If you can't reproduce published results, the problem may be with the code or publication, not with you.

As with all software development, good software development practices should be applied. In particular, use software revision control (e.g., github), and commit your changes regularly. Do not store files that can be reproduced or downloaded from other sources. However, you should include scripts for performing downloads and other data pre-processing such as separating into training, validation and test sets. When running code, print out and save debugging information, but not within tight loops where is can slow down execution of your code. You should also save checkpoints of your models, which will allow you to resume training without having to start from scratch.[21]

There are many tools that can be used to monitor your experiments as they run. Linux's `top`, Window's Task Manager, and Nvidia's `nvidia-smi` are very simple tools to check for memory usage and potential leaks. And before embarking on a very large experiment that will consume many hours or days, make sure to run your code for a few iterations on a small example rather than your entire dataset.

7.4 Dataset Issues

Data is the fuel for deep learning algorithms—the cleaner the data, the more efficient the engine and the better the results will be; the more data we have, the bigger the engine you can drive (model you can train) and the better the results will be. Unfortunately, all datasets are biased and almost all datasets contain errors. Figure 86 shows examples of bias from two very popular, albeit aging, datasets. The left panel shows near duplicate images from a semantic segmentation dataset called MSRC [17]. The problem with such examples is that if one image from each pair appears in the training set and the other appears in the test set, then for a model to do well it simply needs to memorize the images, and we get a false sense of the ability for the model to generalize, which is the whole point of the test set.

The right panel in Figure 86 shows the average image from each class of the Caltech-101 dataset [59]. This dataset was a (small) precursor to the ImageNet dataset [20] for image classification. Here the average image for a class is obtained by simply taking the mean pixel values at each location for all images of that class. Clearly some classes, like face, airplane, stop sign, ying yang, etc., exhibit very little variation and would therefore be very easy to classify. This is despite the fact that if we looked at a broader collection of images (from outside the dataset) we would see significantly more variability, pointing to an unintentional bias in how the dataset was collected.

Datasets can also be labeled incorrectly. Figure 87 shows examples from several popular datasets where the given label in the dataset for a particular example is wrong. Such erroneous labels can be very difficult to find and correct without laboriously examining every image (and label) in the dataset. One may think it possible to train and an image classification model and then examine only the images that it predicts incorrectly to determined whether those images are really incorrect or whether they have been labeled wrong in the dataset. However, this only corrects false-positive labels not false-negatives (image where both the model and ground-truth are incorrect). Inevitably we have to develop

[21] However, when reporting results (to your boss or via a scientific publication) make sure to re-run from scratch. Changing code that led to a checkpoint but resuming training from that checkpoint will create results that cannot be reproduced.

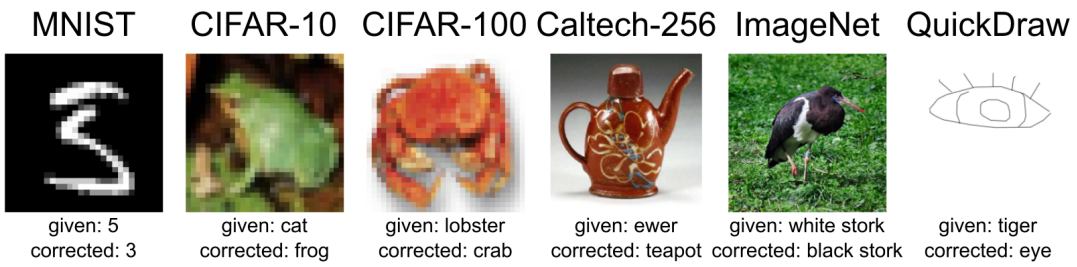


Figure 87: Almost every dataset is wrong. Source: `https://github.com/cleanlab/cleanlab`.

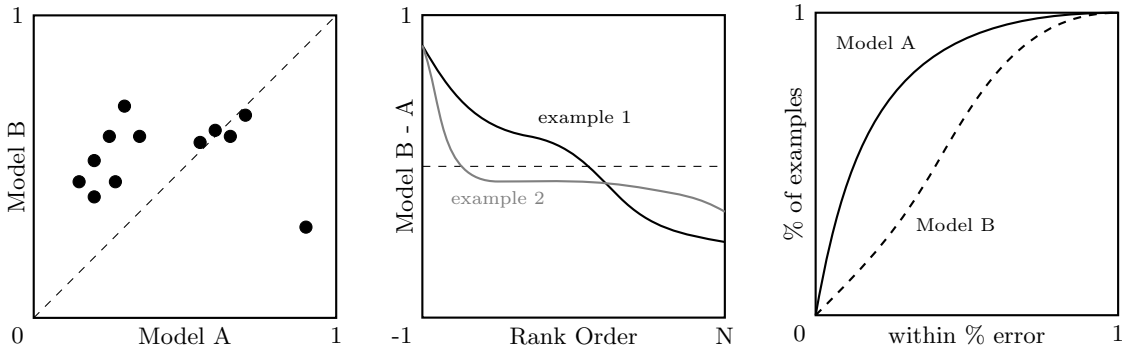


Figure 88: Use different types of plots to discover correlations in your data or compare models.

algorithms that can handle noisy labels and accept the fact that our performance estimates will be slightly tainted by incorrect ground-truth.

7.5 Diagnostics

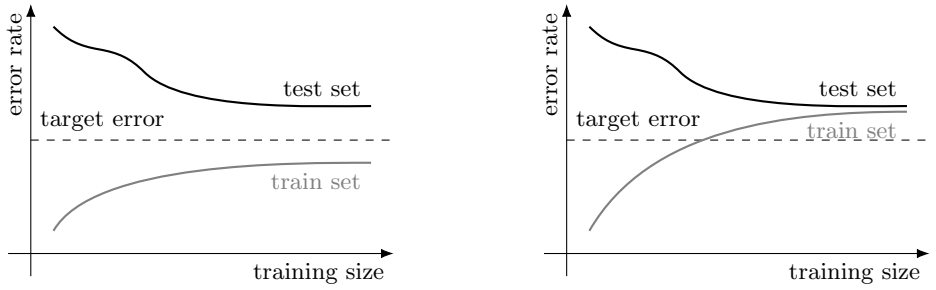
Given all the things that can go wrong, its important that you develop skills and tools for diagnosing your model. One useful tip is to measure everything and visualize everything. The former is particularly important for very long running experiments where you want to avoid having to run jobs a second time just to obtain some statistic you forgot to record during the first run. The latter allows us to quickly understand a bigger picture story that is hard to see when looking at raw statistics. It can also very quickly uncover catastrophic failures or sub-patterns within the data. For example, Figure 88 shows three different ways to compare models. In the first way, we plot the performance on individual test samples on one model versus another. The diagonal line separates samples that perform better in one model than the other. This sometimes can reveal clusters of samples that warrant further investigation. For example, there appears to be one outlier that performs much better on Model A than Model B in the figure. We may want to understand what is special about this data sample.

The second way to compare models is to plot their difference in performance on each sample in rank order. Figure 88 shows two examples. In the first example, both Model A and Model B appear to perform about the same on average, albeit performing differently on different sets of samples. In the second example, Model B performs much better than Model A on a few samples and slightly worse on many samples. In both example, the average performance of the models looks about the same, but just looking at the average hides the nuances of their behaviour over the samples.

The third way to compare models is by plotting a cumulative distributions over some performance measure (such as percentage error). In the example in Figure 88, Model A clearly dominates over Model B.

These examples show that viewing results in different ways gives different insights into the performance of the model. Indeed, the choice of metrics can influence interpretation of results. We saw this when we discussed metrics for image classification in Lecture 4. You should always be asking questions of your metrics. Not just what is being measured but what the measurement tells you about the model or data. In a similar vein, you should maintain a mental model of your code/algorithm. Exercise your mental model against your observations to ensure you understand what your seeing. As the famous physicist Enrico Fermi once said "experimental confirmation of a prediction is merely a measurement; experimental disproving of a prediction is a discovery." That is, try to break your mental model to discovery something new. You can develop diagnostic tests to help refine your mental model as we will show shortly.

When conducting experiments it is important to only change one thing at a time, i.e., run controlled experiments. This again should align with your mental understanding of your method and either confirm or disprove that the thing that you are changing is doing what you think it is doing. For example, loss functions are often constructed as the



(a) high variance (b) high bias

Figure 89: Diagnosing over-fitting versus poor features from learning curves using bias and variance.

weighted sum of many different terms,

$$L(\theta) = \lambda_1 L_1(\theta) + \lambda_2 L_2(\theta) + \dots \tag{206}$$

Investigate the contribution of each term by varying the weight of just that term. Setting the weight to zero should remove the behaviour encouraged by that term, whereas setting the weight very high should make that term, and hence its effect, dominate. When setting the weight of the loss terms remember that not all losses have the same scale or are bounded, so take this into account if you want to balance the effect of the different terms. You can do this with other hyper-parameters too. Remember the idea is to get a better understanding of how your model and algorithm is working, not just to get the best performance.

Last, always report trends not just single results. This reveals patterns of behavior that are less likely to be due to random chance, which is important not just for diagnoses but also scientific integrity.

7.5.1 Diagnostic Example: Over-fitting versus Poor Features

Suppose that we have trained and tested a model and determined that the test error is unacceptably high (meaning that we won't be able to ship a product, for example). We suspect that the problem is either that the model is **over-fitting** or the **features** are not good enough. The first hypothesis suggests that training error will be much lower than test error whereas the second hypothesis suggests that training error and test error will both be high. This is illustrated in Figure 89(a) and (b), respectively, which shows learning curves representative of the two different hypotheses. In the first case there is a big gap between the performance on the training set versus the test set. This is an example of **high variance** and indicates that the model has over-fitted to the training data. In the second case the training and test performances are close but they are both too high. This is an example of **high bias** and indicates that the features are not good enough or the model is not expressive enough.

In general there is a trade-off between bias and variance as we change the model complexity as depicted in Figure 90. Here we are plotting the error rate of a model *after* training as a function of the model complexity, which can be roughly measured as the number of learnable parameters. Sometimes the number of training iterations is also used as a surrogate for model complexity when the number of parameters is very high. The idea can be made more formal, but that is not necessary for our purposes.

As complexity increases models move from having high bias (i.e., both train and test error is high) through to high variance (i.e., train error is low but test error is high). We traditionally want to stop (or pick a model) somewhere between these to regimes where the error on the test set is the lowest. A phenomenon known as **double descent** has been observed in deep learning, where further increasing model complexity sometimes results in the test set error dropping again (with training error continuing to drop). This is not yet well understood theoretically.

Diagnosing bias and variance problems provides hints as to what to try next. For bias problems we can try a larger set of features or larger capacity model, such as higher dimensional hidden layers or deeper model. For variance problems we try getting more training examples, but this may be expensive or simply not possible. Alternatively, we could try reduce the set of features, try a smaller capacity model, or try training for longer (and hope for double descent).

7.5.2 Diagnostic Example: Objective versus Optimization Algorithm

Perhaps our poor performance is not due to over-fitting or poor features at all. Perhaps its a problem with our **optimisation algorithm** (e.g., not running for long enough) or maybe a problem with our **objective** (i.e., loss function). Unfortunately, it is often very difficult to determine whether training has converged as Figure 91 illustrates. But this does not mean that we can't diagnose optimization issues.

Suppose we care about maximizing some accuracy measure, $J(\theta)$, and our learning algorithm is trying to minimize

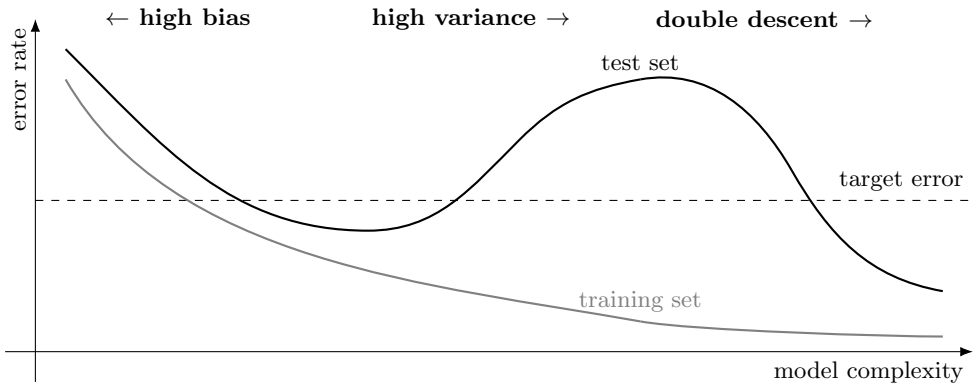


Figure 90: Bias-variance trade-off, and the double descent phenomenon.

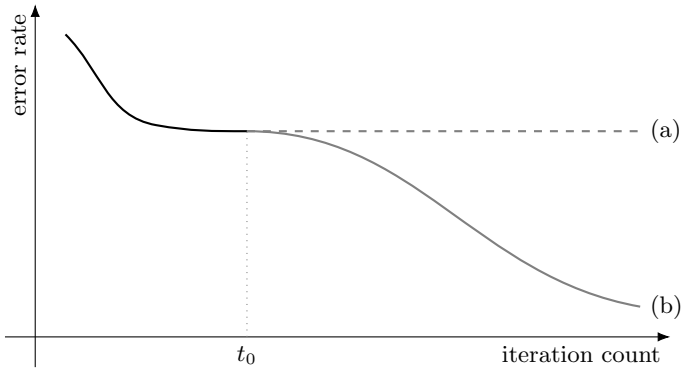


Figure 91: Diagnosing optimization problems. If we only train up to iteration t_0 it is hard to tell whether the optimization algorithm has converged (a) or whether it will continue to improve (b).

some surrogate loss, $L(\theta)$. Let $\theta^\star$ be the parameters returned by the learning algorithm, and let $\theta^\dagger$ be any other parameters (e.g., guessed or from a different learning algorithm). Then,

- if $L(\theta^\star) < L(\theta^\dagger)$ but $J(\theta^\dagger) > J(\theta^\star)$ then we are optimizing the wrong objective since a lower value of the loss function does not translate to better accuracy;
- if $L(\theta^\star) > L(\theta^\dagger)$ and $J(\theta^\dagger) > J(\theta^\star)$ then we have a poor optimization algorithm since there exists another set of parameter values with lower loss;
- if $L(\theta^\star) > L(\theta^\dagger)$ but $J(\theta^\dagger) < J(\theta^\star)$ then we still have a poor optimization algorithm but perhaps we got lucky in that the suboptimal parameters achieve better accuracy;
- if $L(\theta^\star) < L(\theta^\dagger)$ and $J(\theta^\dagger) < J(\theta^\star)$ then we probably don't have a problem at all (or $\theta^\dagger$ is not a good choice for this diagnostic).

Diagnosing optimization versus objective problems provides us with hints as to what to try next. For optimization problems try running for more iterations, try using a different algorithm (e.g., AdamW instead of SGD), try reinitializing parameters using different random seeds (known as random restarts), or try smoothing (e.g., momentum, exponential moving average). For objective problems we can try a different regularization method (such as different data augmentations), try weighting training examples to encourage poorly performing categories, try a different loss function, or try changing the model.

7.5.3 Diagnostic Example: Search versus Score

As another example suppose we are using an **approximate nearest neighbour** algorithm to find similar objects (e.g., images in a large corpus or matched keypoints across two different images). To do this we must define a similarity measure that our algorithm can use. Assume that we are getting poor matches. How can we tell if we have a problem with the nearest neighbour algorithm or our similarity measure?

Let $x^\dagger$ be a match found by the algorithm, and let $x^\star$ be a laboriously-obtained hand-selected match (i.e., ground-truth). If $\textbf{similarity}(x, x^\dagger) < \textbf{similarity}(x, x^\star)$, then the problem is with the similarity measure. Otherwise, initialize the nearest neighbour algorithm with the true solution and if the algorithm moves away from the true solution, then the problem is, again, with the similarity measure. Otherwise, the problem is with the nearest neighbour search algorithm.

7.5.4 Error Analysis and Ablation Analysis

Diagnostics are an important tool when developing your machine learning method. Their purpose is to improve your understanding of your method, not to simply measure its performance. We showed examples for bias/variance, optimization/objective, and search/score, but there are many others. When your model is not performing how you would like, diagnostics can save a lot of wasted effort by guiding your choice of what to try next. They also allow you to develop insights into your particular application and justify your design decisions.

Diagnostics often involve repeated experiments with different parameter settings while keeping everything else fixed. So it is important to set up an environment where you can run experiments quickly and be able to reproduce results.

Other important diagnostic tools are that of **error analysis**, i.e., understanding where your method is making mistakes, and **ablation analysis**, i.e., understanding which parts of your model help the most. Error analysis tries to explain the difference between **current** and **perfect** performance. That is, answering the question, how much error is due to different model components? One way you can do this is by plugging the ground-truth (if available) into each component and see how it affects accuracy. You can also add noise to each component and see how it (adversely) affects accuracy. You can also use error analysis to determine whether the algorithm fail on a particular subclass of examples. And, of course, always visualize the data and results.

Ablation analysis tries to explain the difference between some **baseline** and **current** performance. Often methods are built in an ad hoc fashion and it is not clear whether design decisions made during incremental development are still helping the final model. It is quite common, for example, for features/components are added in early development and kept even if not needed. Ablation analysis removes one feature/component from the model one at a time and sees which results in the biggest decrease in performance, thereby supporting the decision to include that feature/component. Note that the order of removal matters. If a feature or component is removed without changing the performance (or behaviour) of the model, then the model can be simplified (and then possibly further developed).

7.6 Advice for Getting Unstuck

We conclude the lecture with some advice on what to do when you get stuck. Some problems are just inherently difficult. If you can't solve a problem then try finding a simpler (related) problem and solve that first. This will give you experience and hints that may help to solve the original problem. You can also break a big problem down into subproblems, which by themselves are easier to solve.

It is a good idea to explore boundary conditions. This will give you a feel for the conditions under which your method works and when it fails. Extreme examples can give surprising results. What happens, for example, if you provide your classification algorithm with an all black image? What about all white?

Reading research papers (both old and new) and blog posts can be an inspiration for ideas. However, be skeptical about results reported in the literature that are not reproducible. If researchers have released code along with their papers, then try downloading and running their code. Similarly, talk to friends and colleagues. If there is nobody around you can try a technique from software development called **rubber duck debugging**. In this technique, you explain your problem to a rubber duck.[22] Just verbalizing the problem is often enough for a solution to present itself. The method works surprisingly well.

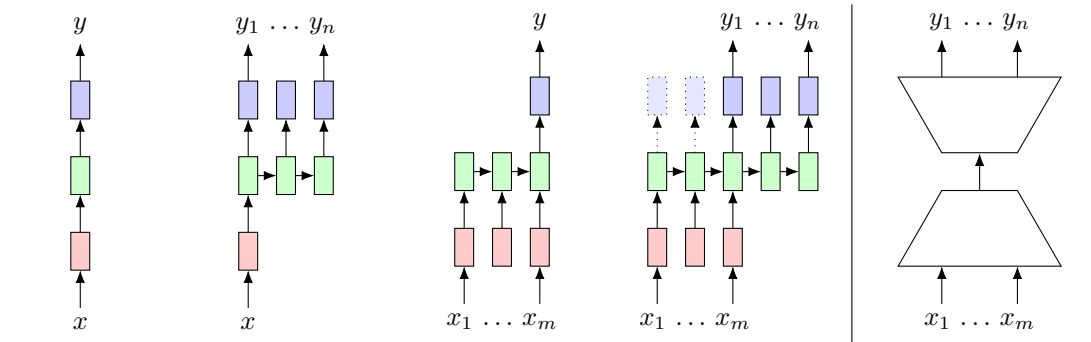
A more principled approach is to introduce more information and slowly remove. In the extreme case you can "cheat" by introducing ground-truth information (e.g., features that encode the answer) into your algorithm. Of course, this is only for debugging and the ground-truth needs to be removed once you've determined a direction to proceed. It helps to print big warning messages when you're doing this sort of experiment so as not to accidentally report tainted results. At the other extreme you can test on random features. If you're algorithm performs better than random guessing, then there is probably something wrong.

Last, you can generate synthetic training data or test cases. This allows you create very large datasets and control the type of samples on which you train/test your model. It can also provide some intermediate ground-truth signals to diagnose what information is important to your algorithm if available (e.g., depth information). There is a very large body of work that looks at how to translate from algorithms trained on synthetic data to perform on real data, a problem known as **sim-to-real**.

7.7 Tools for Support Experiments

A very popular tool for managing and comparing experimental runs is Weights & Biases (`https://wandb.ai`), which is free for small projects and personal use. Code is easily instrumented to allow reporting of statistics and results by Weights & Biases, and the tool comes with very good online documentation.

[22] The duck can be imaginary. No one will know.



(a) one-to-one (b) one-to-many (c) many-to-one (d) many-to-many (e) encoder-decoder view

Figure 92: Unlike an image or label, a sequence model allows inputs and outputs to have arbitrary length. Different combinations are suited for different applications, including (a) image classification, (b) image captioning, (c) language-prompted image generation, and (d) language translation. Again the encoder-decoder paradigm (e) is useful when thinking about these applications.

8 Sequence Models

In this lecture we go beyond models that process fixed-size vectors and tensors, and produce fixed-size outputs, to models that can handle input and output of arbitrary length. These are known as sequence models, and they come in a variety of flavours. We will introduce the idea of a recurrent neural network (RNN) for processing sequence data, and then show how these can be used to solve several tasks involving language. In the next lecture we will discuss the transformer model, which is another approach to processing sequences.

So far in this course we have been concerned with processing fixed-size objects, such as n-dimensional feature vectors and H-by-W pixel images, which has been useful for tasks such as image classification and object detection. Being able to process sequential data is also very important. There are many examples of tasks involving such data, including image captioning [46, 7], visual question answering [9], video classification [80, 15], natural language translation [82], and vision-and-language navigation [8, 43]. These can be categorized as being one-to-many, many-to-one or many-to-many based on the number of elements in the input and output sequences, respectively, as illustrated in Figure 92.

Unlike images and labels, a sequence has arbitrary length making it difficult to process with the models that we studied in previous lectures.[23] We need a network architecture that can handle data of any length.

8.1 Recurrent Neural Networks

A recurrent neural network (RNN) is a model that processes an input sequence one element at a time. It also takes part of the output and feeds it back into itself as an additional input. This allows it to keep state information based on previously processed inputs. A schematic diagram of an RNN is shown in Figure 93. Let x_t be the t-th element in the input sequence and h_{t-1} be the partial output from the previous time-step. Then the output of the RNN is computed as,

$$(y_t, h_t) = f(x_t, h_{t-1}; \theta) \tag{207}$$

where the same function f with same parameters θ is applied at each time-step. Variable h_t is called the **hidden** or **latent** state. It is initialized to some nominal value, h_0, at the start of the sequence. Note that as defined, the RNN produces one output for every input, and is therefore a many-to-many model. It can be made into a many-to-one model by waiting until the entire sequence is processed and discarding all but the last output.

A concrete example of an early recurrent neural network proposed by cognitive scientist Elman [24] is shown in Figure 94. Here the latent state and output are computed as,

$$h_t = \mathbf{tanh}(W_h h_{t-1} + U_h x_t + b_h) \tag{208}$$

$$y_t = \mathbf{sigmoid}(W_y h_t + b_y), \tag{209}$$

[23] We did seen examples of things like bounding box proposals, which are in principle sets or arbitrary size, but these were always padded or truncated to fixed-length lists stored in fixed-size tensors. We also saw fully convolutional networks for image segmentation, which can be viewed as a very special case of the types of models that we will see in this lecture.

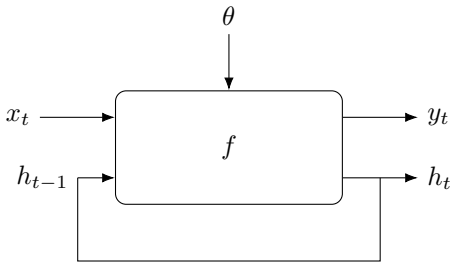


Figure 93: A recurrent neural network (RNN) is used to process sequence data.

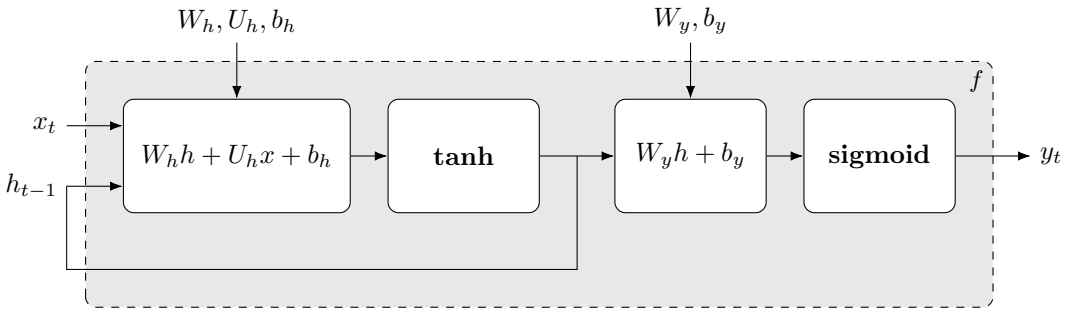


Figure 94: Concrete example of an RNN model.

respectively. This is just a variant of the multi-layer perceptron where the hidden state h_t and output y_t are elementwise non-linear transformations of linear functions of the current input x_t and previous hidden state h_{t-1}. The learnable parameters are W_h, U_h, b_h, W_y and b_y.

We can view convolution over an arbitrarily long input signal as a very simple recurrent neural network. For brevity let us consider the one-dimensional case. Let $a \in \mathbb{R}^p$ be a p-length filter kernel, and let $x_t \in \mathbb{R}$ be the value of the input signal at time t. Initialize $h_0 = (0, \ldots, 0) \in \mathbb{R}^p$. Then we can implement convolution as a recurrent network with

$$\begin{aligned} h_t &= (h_{t-1}[2:p], x_t) && (210)\\ &= (x_{t-p+1}, \ldots, x_{t-1}, x_t) && (211)\\ y_t &= a^T h_t && (212) \end{aligned}$$

where the output is delayed by $p-1$ time steps, which is how long it takes for the model to process the receptive field of the filter kernel. However, general RNNs are much more expressive.

8.2 Back-propagation Through Time

To train a recurrent neural network we need to back-propagate through each time step. We do this by **unrolling** the network to a given time horizon,

$$\begin{aligned} (y_t, h_t) &= f(x_t, h_{t-1}; \theta)\\ (y_{t-1}, h_{t-1}) &= f(x_{t-1}, h_{t-2}; \theta)\\ &\vdots\\ (y_1, h_1) &= f(x_1, h_0; \theta) \end{aligned} \tag{213}$$

and then back-propagating through the unrolled sequence of operations. Note that output y_t depends on parameters θ and latent state h_{t-1}, which in turn depends on θ and h_{t-2}, which in turn depends on θ and h_{t-3}, and so on. In other words, the parameters affect the output through direct and multiple indirect paths. Assuming that the loss is only applied to the last output, y_t, we have

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial \theta} + \frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial \theta} + \cdots + \frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_{t-1}}\left(\prod_{\tau=1}^{t-1}\frac{\partial h_{\tau+1}}{\partial h_\tau}\right)\frac{\partial h_1}{\partial \theta} \tag{214}$$

where we have recursively invoked the calculation of direct and indirect gradients. If the loss function were applied to intermediate outputs then these would also need to be accounted for in the gradient calculation. An illustration of the unrolled network and the concept of back-propagation through time is shown in Figure 95.

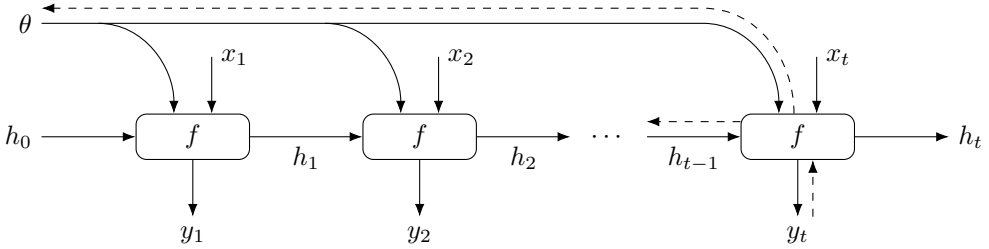


Figure 95: Back-propagation through time by unrolling the recurrent network.

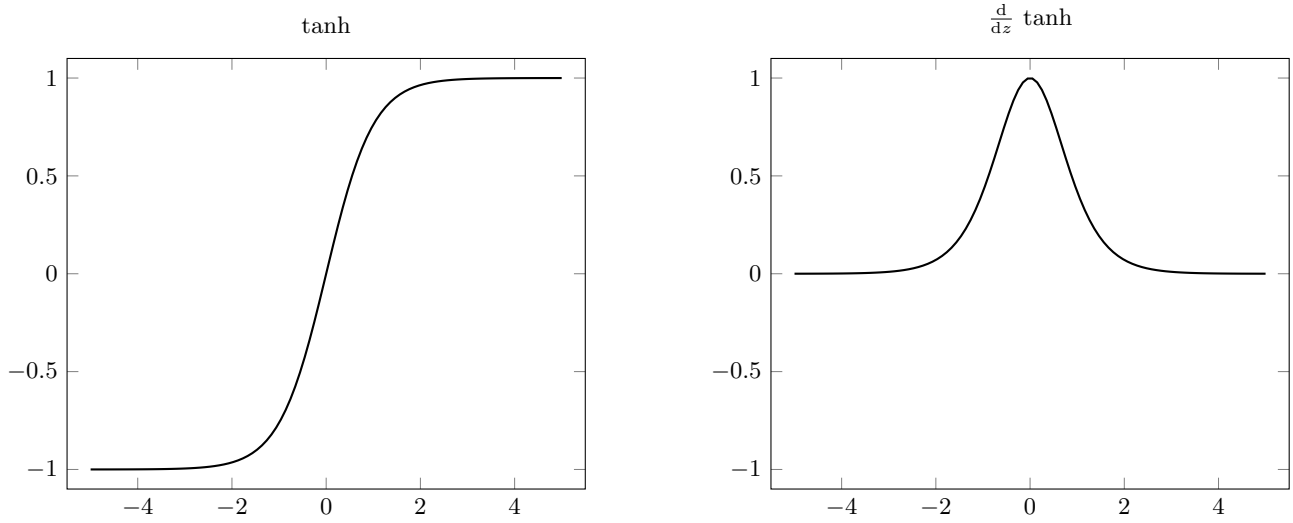


Figure 96: Vanishing and exploding gradients in RNNs can be understood when we examine the gradient of the activation functions, in this example the hyperbolic tangent function.

One of the issues with unrolling a recurrent model is that it creates a very deep network if we unroll for a long time horizon. As we have seen in previous lectures, this exacerbates the vanishing and exploding gradient problem. Consider the gradient contribution from a timestep $s < t - 1$ for the concrete example RNN shown in Figure 94,

$$\frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_{t-1}}\left(\prod_{\tau=s}^{t-1}\frac{\partial h_{\tau+1}}{\partial h_\tau}\right)\frac{\partial h_s}{\partial \theta} \tag{215}$$

Each term in the product $\frac{\partial h_{\tau+1}}{\partial h_\tau}$ is obtained by differentiating the **tanh** activation function,

$$h_t = \mathbf{tanh}(W_h h_{t-1} + U_h x_t + b_h) \tag{216}$$

Using the result, $\frac{\mathrm{d}}{\mathrm{d}z}\tanh(z) = 1 - \tanh^2(z)$, we have

$$\frac{\partial h_{\tau+1}}{\partial h_\tau} = \mathbf{diag}(1 - h_{\tau+1}^2)W_h \tag{217}$$

where $h_{\tau+1}^2$ is the vector $h_{\tau+1}$ with its elements squared. Figure 96 plots both the **tanh** function and its derivative. We can observe from the plots that if any $h_{\tau+1}$ is close to ± 1 or $\sigma_{\max}(W_h) < 1$, then the gradient will vanish. Furthermore, if all $h_{\tau+1}$ are close to zero and $\sigma_{\max}(W_h) > 1$, then the gradient will explode. Here $\sigma_{\max}$ is the maximum singular value of W_h (and has no relation to the same symbol σ used to denote the logistic function in earlier lectures and elsewhere in this lecture).

8.3 Long Short-Term Memory Model (LSTM)

To address the vanishing and exploding gradient problem, Hochreiter and Schmidhuber [42] introduced the long short-term memory model (LSTM). An architecture diagram for the LSTM, known as a cell, is shown in Figure 97. Like blocks in a residual network, the LSTM cell attempts to induce better flow of information by short-circuiting some of the operations that may lead to suppressed gradients. It does this by introducing a cell memory, c_t and gating

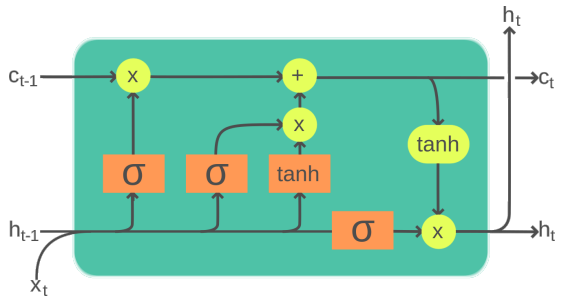


Figure 97: Architecture of the long short-term memory (LSTM) model [42].

functions in addition to the cell output/latent state h_t. Three gating functions are defined, f_t, i_t and o_t, as follows,

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad \text{forget activations} \tag{218}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad \text{input activations} \tag{219}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad \text{output activations} \tag{220}$$

The cell memory c_t and output h_t are then updated as

$$c_t = f_t \circ c_{t-1} + i_t \circ \tanh(W_c x_t + U_c h_{t-1} + b_c) \quad \text{cell memory} \tag{221}$$

$$h_t = o_t \circ \tanh(c_t) \quad \text{cell output} \tag{222}$$

where $\circ$ denotes elementwise multiplication.

In words, the cell memory is updated as a combination of the previous memory and input signal, regulated by the forget and input gates, respectively. If the forget gate is low, then the cell forgets its previous value and is just updated with the input. If the input gate is low, then the cell is retains its previous memory (still modulated by the forget gate). The cell outputs its memory if the output gate is high.

As can be seen in the architecture diagram (and the equations defining the cell), there is a relatively simply path through from the cell memory at the previous time step c_{t-1} to the cell memory at the current time step c_t. This mitigates the problem of vanishing gradients for long sequences.

The LSTM was the architecture of choice for processing sequential data for many years, but has recently been replaced by the transformer (which we will see in the next lecture).

8.4 Sequence-to-sequence Encoder-decoder Model

A sequence-to-sequence model (sometimes abbreviated as "seq-to-seq") is a model that consumes an arbitrary sequence as input and produces an arbitrary sequence as output. There are several variants where the model can either consume the entire input sequence before producing the output sequence, or the model can start producing elements of the output while still consuming input elements. The former can be regarded as an encoder-decoder model where we combined a many-to-one model with a one-to-many model (see Figure 98). Here an encoder recurrent neural network consumes the arbitrary length input sequence to produce a latent state vector that represents the input. This vector is then used to initialize the hidden state of a decoder recurrent neural network, which produces an arbitrary length output sequence. The only output of the encoder RNN is the latent state and the only input to the decoder RNN is the state being fed back in.

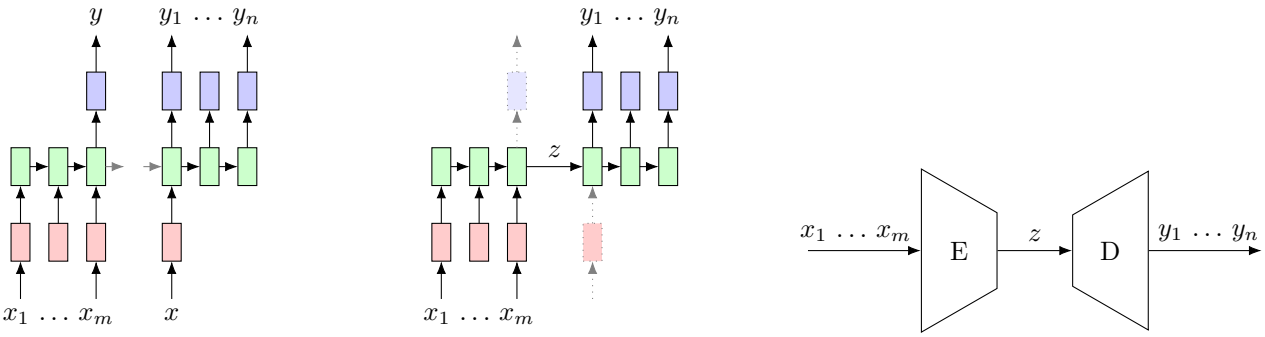
8.5 Language Models

Language models are sequence-to-sequence models that produce an output for each input in the sequence. Specifically, a language model predicts the next token in a sequence where a token can be an individual character, an complete word, or anything in between (e.g., an n-gram) that is suitably encoded as a vector (to be discussed shortly). At each time step, an RNN outputs a probability distribution over all possible tokens. For example,

$$h_t = f(h_{t-1}, x_t) \tag{223}$$

$$y_t = \textbf{softmax}(Ah_t + b) \tag{224}$$

where each element in y_t represents the probability that the next token has index corresponding to that element and is constructed as a single-layer perceptron (logistic regressor) with the RNN hidden state as input. We then take the most likely token as the predicted output or sample a token weighted by its probability. A depiction of a language model is shown in Figure 99. Since the RNN is producing a probability distribution, it can be learned using a cross-entropy loss. Note that the target output at time t is the same as the model's input at time $t+1$. As such, language models can be trained in a self-supervised manner on a large corpus of text.



(a) many-to-one and one-to-many (b) sequence-to-sequence (c) same as encoder-decoder

Figure 98: A sequence-to-sequence model can be constructed as a many-to-one model followed by a one-to-many model. An alternative view is as a sequence encoder followed by a sequence decoder.

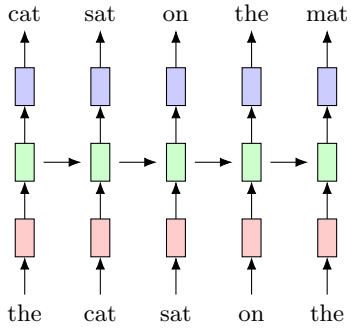


Figure 99: Language models. The output of the RNN estimates a probability distribution for the next word in the sequence (in this example, the sentence "*the cat sat on the mat*"), from which we can draw the most likely next word or sample weighted by the probability.

8.5.1 Encoding Words as Vectors

Neural networks process numerical data; they cannot process symbolic data such as words. As such we need a mechanism to encode (and decode) words as vectors. Consider a vocabulary $\mathcal{V}$ consisting of m words. We can encode each word in $\mathcal{V}$ using one-hot encoding as illustrated below,

$$\begin{array}{lcccccc} \text{“dog”} & \to & e_1 & \to & \boxed{Ax+b} & \to & z_1 \\ \text{“cat”} & \to & e_2 & \to & \boxed{Ax+b} & \to & z_2 \\ \text{“duck”} & \to & e_3 & \to & \boxed{Ax+b} & \to & z_3 \\ & & \vdots & & \vdots & & \end{array} \tag{225}$$

That is, the i-th word in the list maps to $e_i = (0, \ldots, 1, 0, \ldots) \in \mathbb{R}^m$.

Since the first layer of an RNN is typically a linear layer, $z = Ax + b$, with $A \in \mathbb{R}^{n \times m}$, we can map directly from the word to the output of this linear layer.[24] The i-th word mapping becomes

$$z_i = Ae_i = a_i + b \tag{226}$$

where a_i is the i-th column of A. Note that b can be absorbed into A as $A \leftarrow A + b\mathbf{1}^T$ for the special case of multiplying only by one-hot vectors, and so can be ignored for the purpose of learning embeddings. So, in summary, instead of going through the step of encoding words using a one-hot vector, we can just directly learn **word embeddings**, where $z_i \in \mathbb{R}^n$ represents the i-th word in the vocabulary. The matrix A is sometimes called the embedding matrix.

8.5.2 Training Language Models

Getting back to language model training, we mentioned above that they can be trained using a cross-entropy loss. Roughly speaking, we wish to learn the parameters of the model that best predict the next token in a sequence, a task known as **next-token prediction**. The approach follows a teacher-forcing paradigm, where irrelevant of the next-token distribution predicted by the model at any time step, the learning algorithm provides the (supervised)

[24] The A and b here are different from the A and b in Equation 224.

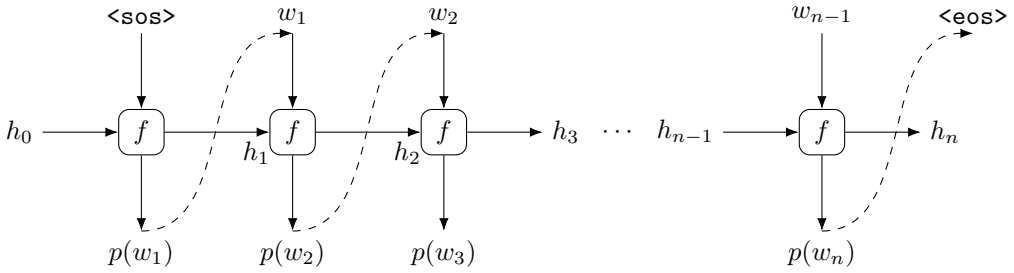


Figure 100: Auto-regressive language generation. The output (bottom) of the model at each time step is a probability distribution, which is sampled to produce the input (top) for the next time step.

ground-truth token as input for the next time step. The objective is then to minimize the cross-entropy loss over next-token distributions for all time steps,

$$L(\theta) = -\sum_{t=1}^{T} \log \left[\mathbf{softmax}(Af(h_{t-1}, x_t) + b)\right]_{w_t} \tag{227}$$

where w_t is the index of the correct ground-truth word (or token). The loss function is defined over all parameters θ, which include A, b, and the parameters of recurrent function f.

As we've already discussed, back propagating through long temporal sequences can be problematic from a vanishing gradient perspective, and also from a memory perspective—we have to store all intermediate calculations from the forward pass. One way to get around this difficulty when training language models is to chunk long sequences into segments of length T, by caching h_s for some $s > 0$ and treating it as a constant for the next chunk,

$$h_s = f \circ \ldots \circ f \circ f(h_0, x_1, x_2, \ldots, x_s) \tag{228}$$

$$L(\theta) = -\sum_{t=s+1}^{s+T} \log \left[\mathbf{softmax}(Af(h_{t-1}, x_t) + b)\right]_{w_t} \tag{229}$$

The word embeddings are usually pre-trained via a separate process and kept frozen during language model training. We will discuss learning embeddings later in this lecture and in the next lecture.

8.5.3 Auto-regressive Generation and Beam Search

Output sequences can be generated from recurrent neural networks in an auto-regressive fashion as shown in Figure 100. We have already seen that the recurrent network for a language model predicts a probability distribution over the next word in the sequence given the current state and previous word, $p(w_t \mid h_t, w_{t-1})$. To generate an output sequence we initialize the latent state h_0 and set the previous word to a special start-of-sequence token (<sos>). We then sample the first word from $p(w_1 \mid h_0, \texttt{<sos>})$. This word (and the updated latent state) is then provided as input for the second time step. Proceeding in this way we keep sampling the next word w_t from $p(w_t \mid h_t, w_{t-1})$. Generation stops when the model samples another special token, the end-of-sequence token (<eos>).

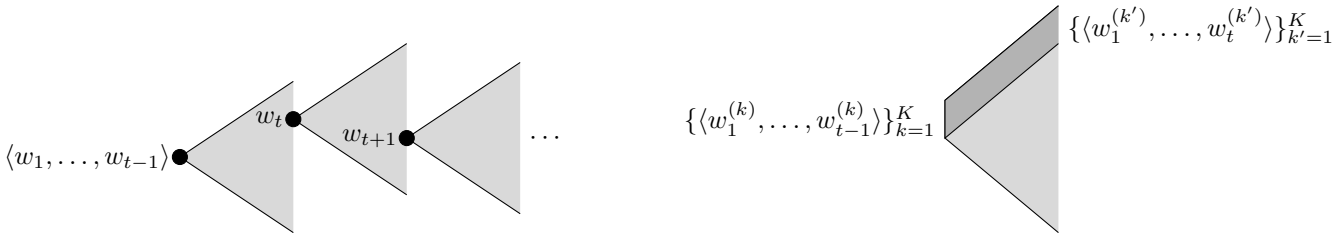
One problem with auto-regressive generation is that it is greedy. Even if we sample the most likely word at each time step, $w_t \in \arg\max_w p(w \mid h_t, w_{t-1})$, this may not result in the highest likelihood overall sequence, $\langle w_1, w_2, \ldots, w_n \rangle$. This may happen, for example, if a word early in the sequence has high probability given the preceding text, but then results in a fairly uniform distribution over future words for the remainder of the sentence.

Beam search is a technique used to mitigate this problem. Here a set of K partially generated sequences, called a **beam**, is maintained, i.e., $\{\langle w_1^{(k)}, \ldots, w_{t-1}^{(k)} \rangle\}_{k=1}^{K}$ at time step $t-1$. Then at time step t we sample several next words for each partial sequence in the beam. The newly created t-length sequences are scored and all but the top K discarded. In principle, we are reducing the set of mK newly created sequences (m next words from a vocabulary of size m for each of the K partial sequences in the beam) back down to a set of K (high scoring) sequences in the beam. An illustration of this process and comparison with greedy search is shown in Figure 101.

A typical scoring metric for partial and full sequences $\langle w_1, w_2, \ldots, w_t \rangle$ is the joint log-probability,

$$\sum_{i=1}^{t} \log p(w_i \mid h_i, w_{i-1}) \tag{230}$$

Other more sophisticated scoring functions that take into account sequence length can also be used. At the end of the generation process, the most likely sequence from the set of K completed sequences is chosen as the model output.



(a) greedy search (b) beam search

Figure 101: Beam search for finding a high probability complete sequence by maintaining several partially completed sequences.

(a) Karpathy and Fei-Fei [46] (b) Anderson et al. [7]

Figure 102: Image captioning models combine a CNN image encoder with an RNN language decoder.

An interesting extension to beam search proposed for guided caption generation is constrained beam search [6]. The idea here is that we want to constrain the output sequence to contain a certain word, or more generally match a pattern encoded by some regular expression. However, since the sequence is generate auto-regressively, we cannot expect the pattern to necessarily be present at the start of the sequences. Indeed, forcing the pattern to appear at the start may result in a very low probability sequence instead of allowing the pattern may appear anywhere. To get around this difficultly Anderson et al. [6] propose to maintain multiple beams, each beam representing partially matched patterns.[25] As words are added to the sequences in the beams, the longer sequences can move to different beams indicating that a pattern is now satisfied or not. Once generation of the sequences completes, only sequences in beams representing matched patterns are candidates for the model output. A subtle tweak to the method just described is to additionally force sampling of transitions between beams (i.e., states in the regular expression automaton) at each time step, thus ensuring that sequences satisfying the constraint always exist.

8.6 Applications

We now discuss three applications for sequence models using recurrent neural networks in computer vision.

8.6.1 Image Captioning

One of the first applications of recurrent neural networks in computer vision was to automatically describe the contents of images in natural language. In other words, image captioning. Conceptually, an image captioning model starts by encoding the image into some latent space followed by decoding the latent representation, by a sequence model, into a sentence that describes the image. Indeed, this was exactly the recipe followed by the early deep learning captioning model of Karpathy and Fei-Fei [46]. Here a convolutional neural network (CNN) is used to encode the image, which provides the initial hidden state for a recurrent neural network (RNN) language model decoder. The CNN is typically pre-trained on the ImageNet dataset[26] and the RNN trained on paired image and caption data to minimize cross-entropy loss of predicted captions. An illustration of the model is shown in Figure 102.

Anderson et al. [7] significantly improved over the earlier captioning models by incorporating a bottom-up and top-down process, and introducing the notion of attention when generating each word.[27] See Figure 102(b). This model is still the de facto standard against which new captioning models are compared.

Briefly, the idea of bottom-up top-down (BUTD) attention is to focus on different parts of the image as the caption is generated so as to condition the next-word probability distribution. A set of candidate salient regions $V = \{v_1, \ldots, v_n\}$ are generated from two sources: a standard $m \times m$ CNN feature map and bounding boxes from an object detector, e.g., Faster-RCNN. See Figure 103. Features on the salient regions from this bottom-up process are combined using

[25] More formally, the beams are associated with states in the finite-state automaton for the regular expression that encodes the pattern.
[26] The final classification layer is stripped from the network after pre-training and the output of the penultimate layer used as the latent representation for the image.
[27] The notion of attention will come up again when we discuss transformers in the next lecture.

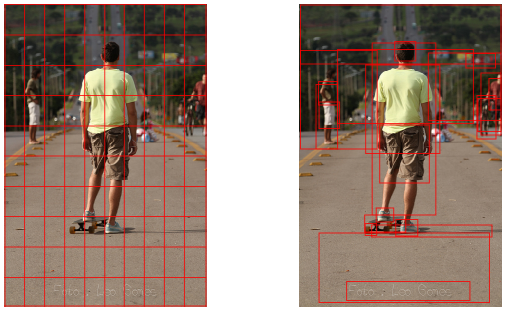


Figure 103: Candidate regions for the bottom-up top-down captioning model [7] are obtained from an $m \times m$ grid (left) and object detection bounding boxes (right).

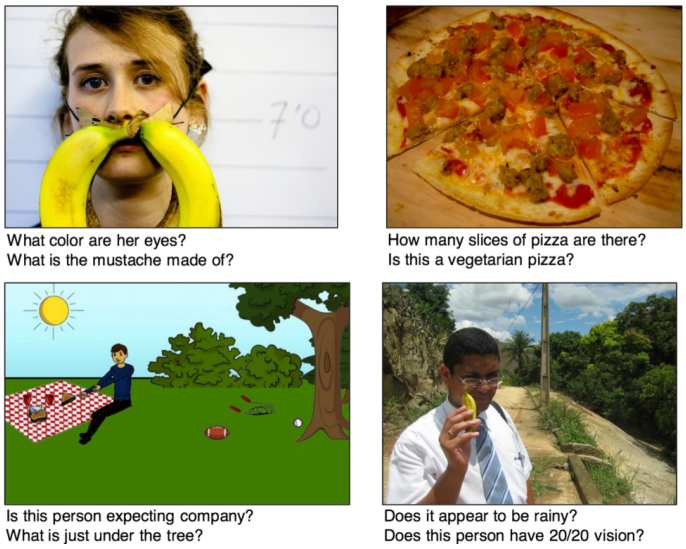


Figure 104: Visual question answering (VQA) requires a model to answer a natural language question about an image [9].

a top-down process that weights them based on current context supplied by the RNN latent state, h_t, as

$$\hat{v}_t = g(h_{t-1}; V, \theta) \tag{231}$$

where g is a learnable attention function, e.g., multi-layer perceptron. The method then follows the standard language model approach to generate the next word,

$$x_t = (w_{t-1}, \hat{v}_t) \tag{232}$$

$$p(w), h_t = f(h_{t-1}, x_t, \theta) \tag{233}$$

$$w_t \sim p(w) \tag{234}$$

As discussed above, beam search can be used to improve the probability of the generated sequence, and hence the quality of the captions.

Several automatic metrics have been proposed to evaluate captioning models. Most of these (BLEU, ROUGE, METEOR, and CIDer) are based on comparing n-grams in the generated caption with those from a reference set of human provided captions. An alternative approach called SPICE [5] is to construct a scene graph [54], which encodes entities and their relationships in an image, and then evaluates how well the nouns, verbs and adjectives in generated caption covers the scene graph. However, caption evaluation is notoriously difficult because images can be described in numerous different ways. Ultimately, human assessment is the best judge, albeit very expensive on a large scale.

8.6.2 Visual Question Answering

Like image captioning, visual question answering (VQA) [9] combines image and language modalities, in this case to answer an arbitrary question about an image. The question and answer are phrased in natural language, or sometimes answers can be selected from a discrete set of options. Unlike image captioning, the input to the model is both an image and a text sequence, i.e., the question. Examples are shown in Figure 104.

Typical models for VQA involve encoding the image and question into latent representations using a pre-trained CNN and RNN, respectively. The combination of image and question latent representations is then decoded into an

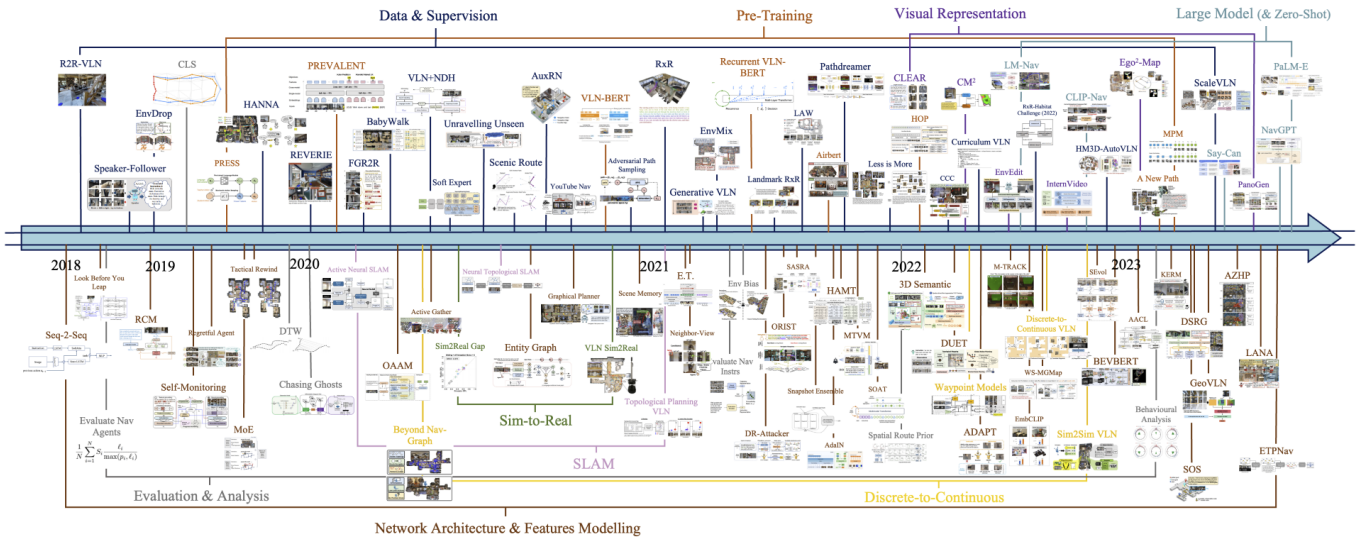


Figure 105: Summary of research in vision and language navigation (VLN). Reproduced from Hong [43].

answer. For open-ended answers a language decoder model is used, whereas for multiple-choice the task is reduced to a multi-label classification problem and an MLP with softmax output layer is used.

Early VQA datasets and models were plagued with biases, e.g., answering "two" whenever it was asked how many of anything is in the image. This has been addressed with bigger datasets and more careful collection to avoid simple biases. More advanced models also include specialized modules for recognising logos and reading text so that they can answer more sophisticated questions about the image.

8.6.3 Vision and Language Navigation

Vision and language navigation (VLN) is a very interesting application that requires reasoning over both visual and language modalities. It was first introduced by Anderson et al. [8]. Here the task is for an agent to navigate through an environment following a natural language instruction, where the instruction refers to visual features of the environment (e.g., "exit the room through the glass doors"). Initial research focused on virtual environments with the agent only allowed to visit a discrete set of locations in the environment. At each location the agent would get to observe the environment from that location and then have a small set of choices of where to go next (or to stop).

The field of VLN has since exploded to entertain many variations, including the ability to pre-explore and map out an environment, operate in a continuous environments, and enhancing instructions with actions to take when reaching the destination (e.g., "bring me a spoon"). Hong [43] has created a visual summary of the history and current state of VLN research. See Figure 105. His PhD thesis provides comprehensive details.

8.7 Joint Vision and Language Embedding

The importance of models that can reason over both visual and language modalities has prompted researchers to investigate whether it is possible to embed images and text in the same latent space such that the visual and language representations of the same concept appear close in the embedding space. This joint embedding ability is the key to enabling prompt-based image generation technologies. It also facilitates so-called zero-shot tasks such as prediction where the model can label images with categories for which it has not been explicitly trained, image captioning, and prompt-based image retrieval.

A common approach introduced by Radford et al. [74] known as CLIP and later improved by BLIP [60], uses a metric learning method called **contrastive learning**. Here the method is given a training set of (image, text)-pairs, $\mathcal{D} = \{(I_i, T_i)\}_{i=1}^N$. These could be collected, for example, by scraping the Internet for captioned photographs. We assume that images are mapped into the latent space using an image encoder network f^{img} and text is mapped into the latent space using a text encoder network f^{txt} with parameters θ^{img} and θ^{txt}, respectively. Then given the training set we can define a contrastive loss between any image I_i and any text T_j,

$$\ell_{i,j}(\theta^{\text{img}}, \theta^{\text{txt}}) = -\log\left(\frac{\exp\phi(I_i, T_j)}{\sum_{k\neq i}\exp\phi(I_i, T_k)}\right) \tag{235}$$

where ϕ is a temperature scaled similarity measure. Let $z_i^{\text{img}} = f^{\text{img}}(I_i; \theta^{\text{img}})$ and $z_j^{\text{txt}} = f^{\text{txt}}(T_j; \theta^{\text{txt}})$ be the embed-

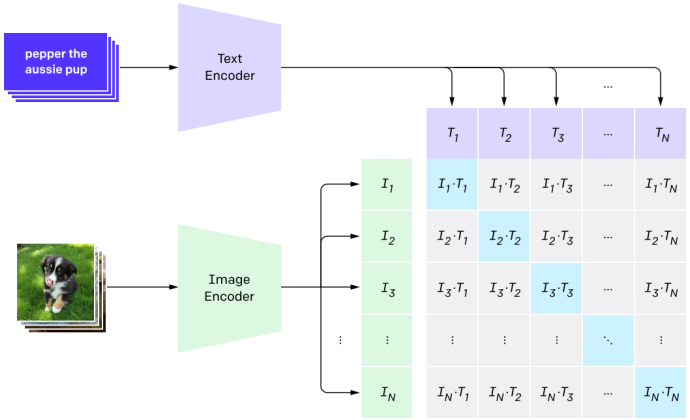


Figure 106: Contrastive language-image pre-training (CLIP) for jointly embedding images and text [74].

dings for image I_i and text T_j, respectively. Then a typical choice for ϕ is cosine similarity,

$$\phi(I_i, T_j) = \frac{1}{\tau} \left(\frac{z_i^{\text{img}}}{\|z_i^{\text{img}}\|_2} \right)^T \left(\frac{z_j^{\text{txt}}}{\|z_j^{\text{txt}}\|_2} \right) \tag{236}$$

Here $\tau > 0$ is the temperature hyper-parameter.

Minimizing this loss on the dataset (i.e., summing over $\ell_{i,i}$ for $i = 1, \ldots, N$) brings embeddings for image I_i and text T_j close to each other for $i = j$ and far apart for $i \neq j$. The method is illustrated in Figure 106 where we want high similarity for pairs along the diagonal of the image-text matrix and low similarity for off-diagonal pairs.

One may naturally ask about the validity of the loss function. Couldn't it be the case that some pairs (I_i, T_j) should be close in feature space even if $i \neq j$, for example two different images of a dog, I_i and I_j? Yes, off-diagonal terms in the comparison matrix depicted in Figure 106 may indeed refer to the same concept and so should not be driven apart. It turns out, that the relative occurrence of such examples is overshadowed by the number of times that the concepts are different coupled with the fact that the loss is only applied to batches of data, and so the method still works as a very good approximation to what we really want (without having to exhaustively label all possible N^2 combinations).

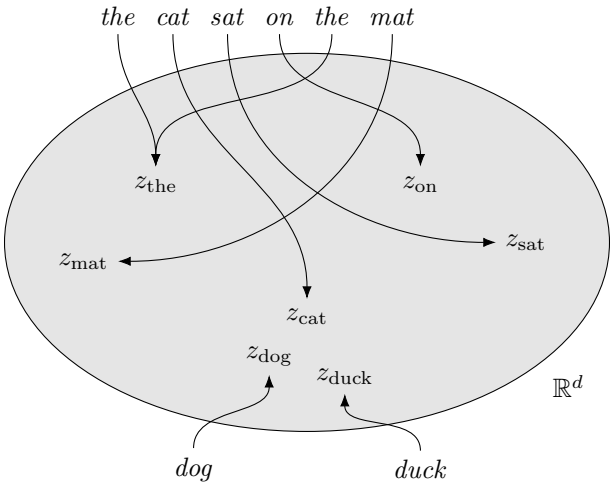


Figure 107: Tokenization breaks long sentences into a sequence of words (or sub-words) and the represents each as a vector. Observe that the same word (e.g., "the" in the sentence above) gets mapped to the same embedding vector irrelevant of where it appears in the sentence.

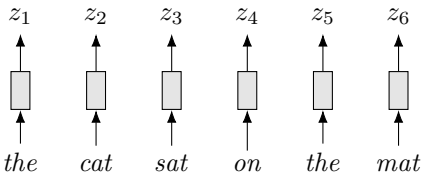


Figure 108: Tokenization of a sequence of words results in a sequence of n-dimensional tokens (vectors).

9 Transformers

This lecture discusses the transformer model, which is the current state-of-the-art model for deep natural language understanding and generative AI. We begin by revisiting the idea of representing words as vectors. We then introduce the dot-product attention mechanism at the heart of the transformer. After covering language models, we show how images can also be represented as sets of vectors and processed by the transformer. The lecture ends with a review of applications that use transformers in computer vision.

Recurrent neural networks (RNNs) have been successfully applied to many sequence classification and generation tasks. However, they suffer from a couple of major drawbacks, specifically the vanishing gradient problem and the fact that all information, including long-range correlations, needs to be captured succinctly by the latent state vector. Transformers [85] overcome these drawbacks by processing all elements of a sequence concurrently—although this introduces some computational challenges of its own, namely, quadratic compute and memory. But despite their high computational cost, transformer models are at the cutting-edge of artificial intelligence, powering technologies like large language models (LLMs) and generative AI.

9.1 Tokenization

The transformer models was originally designed for natural language processing tasks, and as we saw in the last lecture, the first step in being able to process language data is to encode text into a form amenable for numerical calculations in a deep learning model. The mechanism for converting symbolic data into vectors is called **tokenization**. One way to think about this is as a mapping from the discrete set $\{1, 2, \ldots, m\}$ to the space of d-dimensional vectors, $\mathbb{R}^d$. Here each element in the set $\{1, 2, \ldots, m\}$ indexes a discrete item from the universe of symbols, e.g., a vocabulary of words or n-grams. An example is shown in Figure 107.

To make this more concrete, let $E : \mathcal{V} \to \mathbb{R}^d$ be an encoder from fixed vocabulary $\mathcal{V}$ to n-dimensional vectors. This can be implemented, for example, via a lookup table. Then a sequence of T words $\langle w_t \in \mathcal{V} \mid t = 1, \ldots, T \rangle$ gets encoded as a sequence of tokens $\langle z_t = E(w_t) \in \mathbb{R}^d \mid t = 1, \ldots, T \rangle$, as illustrated in Figure 108. In practice, we tokenize sub-words (or n-grams) rather than full words. Pre-processing is sometimes carried out before converting a piece of text into a sequence of vectors to reduce words to their **stems**, i.e., part of the word that captures lexical meaning, and filter **stop words**, i.e., common words that carry little meaning.

We can also decode from tokens back to words using a decoder model, $D : \mathbb{R}^d \to \mathcal{V}$. Here we need to worry about the fact that some vectors in $\mathbb{R}^d$ may not correspond to vectors hit by the encoder E. Mathematically, we say that the encoder function E is not **surjective**, meaning that it only maps onto part of $\mathbb{R}^d$. There are two standard approaches to get around this difficulty. The first approach is to take the nearest encoded vector to an arbitrary vector $z \in \mathbb{R}^d$,

$$D(z) = \arg\min_{w \in \mathcal{V}} \|E(w) - z\|^2 \tag{237}$$

which effectively divides $\mathbb{R}^d$ into piecewise constant regions associated with each word $w \in \mathcal{V}$. The second approach is to treat the decoder as a stochastic function. Instead of a deterministic decoder function D, we define (or learn) a discrete distribution P_D over the embedding space and sample a word from this discrete distribution,

$$w \sim P_D(w \mid z) \tag{238}$$

For example, if z and $E(w)$ are both normalized, then we could define P_D as

$$P_D(w \mid z) = \mathbf{softmax}(E(w)^T z \mid w \in \mathcal{V}). \tag{239}$$

There are several research works that propose how to assign vectors to words, i.e., how to structure the embedding space. In the skip-gram model [66], the high-level idea is to train an encoder model on a large text corpus, $\mathcal{T}$, with the desired property that words appearing close to each other in the corpus should have vector representations that are close to each other in embedding space. Consider word w_i with neighbouring words

$$\mathcal{N}_i = \{w_{i-n}, w_{i-n+1}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+n-1}, w_{i+n}\} \tag{240}$$

for some neighbourhood size n, e.g., $n = 4$. We want w_i to be a good predictor of $w_j \in \mathcal{N}_i$, i.e., we want to maximize the log-likelihood of the conditional probability of adjacent words to a given word for every word in the corpus,

$$\sum_{w_i \in \mathcal{T}} \sum_{w_j \in \mathcal{N}_i} \log P(w_j \mid w_i) \tag{241}$$

Given a training text corpus $\mathcal{T}$, this is a self-supervised objective. The distribution P can be modelled as the **softmax** over embedding similarity, i.e.,

$$P(w_j \mid w_i) \propto \exp(z_i^T z_j) \tag{242}$$

where z_i and z_j are the vector representations of w_i and w_j, respectively, and are the free parameters to be optimized.

Many others approaches and variants of this idea have been proposed, such as GloVe [72] and BERT [21]. For the purposes of these lectures, we will assume that a pre-trained embedding model is supplied, and that the embedding space also contains special tokens such as the start-of-sequence and end-of-sequence tokens discussed in the last lecture.

9.1.1 Positional Encoding

In addition to encoding a word (or n-gram) as a vector, transformers also encode the location that each word appears in the sequence. This is done with a clever technique known as **positional encoding**. Here, the k-th position in a sequence is encoded as a d-dimensional vector ψ_k with $(2i)$-th and $(2i+1)$-th elements[28] determined as

$$\psi_k(2i) = \sin\left(\frac{k}{n^{(2i/d)}}\right) \tag{243}$$

$$\psi_k(2i+1) = \cos\left(\frac{k}{n^{(2i/d)}}\right) \tag{244}$$

for $i = 0, \ldots, \frac{d}{2} - 1$. The elements in the positional encoding vector represent a geometric progression of wavelengths from 2π to $2\pi n$ as illustrated in Figure 109 (Vaswani et al. [85] set $n = 10^4$).

The position encoding vectors are added to the word embedding vectors for each token to give

$$w_k \leftarrow w_k + \psi_k \tag{245}$$

as the vector representation of the token at the k-th position.

The transformer architecture itself has no notion of order. As such, without positional encoding (or masking, which we will see later) the transformer is technically a set processing model rather than a sequence processing model. Positional encoding allows the model to reason about the order of tokens in the set, thus defining a sequence.

[28] indexing here is zero-based

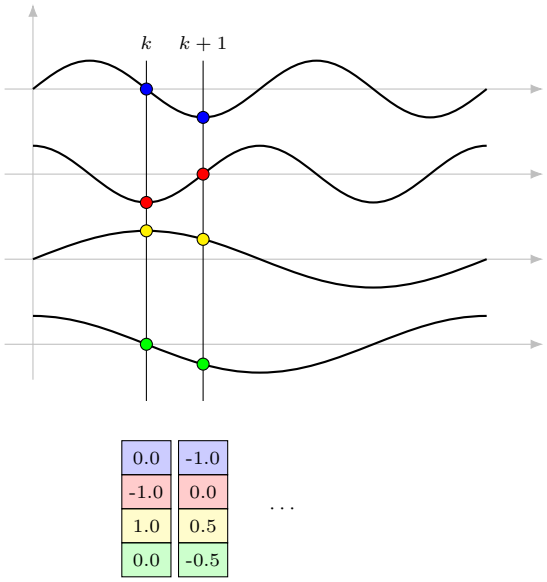


Figure 109: Positional encoding for transformer models.

9.2 Scaled Dot-product Attention

At the heart of the transformer model is a mechanism known as scaled dot-product attention [85]. It replaces the recurrence or convolutions we saw in previous sequence-to-sequence models. The basic idea is shown in Figure 110(left) and can be thought of as a soft content-addressable memory. Let $q_i \in Q$ be some query vector. Then compute the similarity between q_i and every key $k_j \in K$ by computing dot-products, $q_i^T k_j$. Return a weighted combination of values $v_k \in V$, where weights are determined by softmax applied to the similarity scores with appropriate temperature scaling.

Multiple queries can be processed simultaneously. If we arrange queries, keys and values as rows in a matrix,

$$Q = \begin{bmatrix} — & q_1^T & — \\ — & q_2^T & — \\ & \vdots & \\ — & q_m^T & — \end{bmatrix} \in \mathbb{R}^{m \times d}, \quad K = \begin{bmatrix} — & k_1^T & — \\ — & k_2^T & — \\ & \vdots & \\ — & k_n^T & — \end{bmatrix} \in \mathbb{R}^{n \times d}, \quad V = \begin{bmatrix} — & v_1^T & — \\ — & v_2^T & — \\ & \vdots & \\ — & v_n^T & — \end{bmatrix} \in \mathbb{R}^{n \times p} \tag{246}$$

then we can write scaled dot-product attention as

$$\textbf{attention}(Q, K, V) \triangleq \textbf{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V \tag{247}$$

Here the softmax operation is applied row-wise with temperature $\tau = \sqrt{d}$, where d is the dimensionality of the query and key vectors. This helps to keep the softmax values in a reasonable range.[29] Note that keys and values are in a one-to-one correspondence with each other. The number of output vectors (or tokens) is equal to the number of queries, and the number of input vectors (or tokens) is the same as the number of keys and values. Typically in a transformer model we set the number of inputs to be the same as the number of outputs ($m = n$) and the dimensionality of the output vectors to be the same as the dimensionality of the input vectors ($p = d$). As we will see shortly, the queries, keys and values come from tokens in a sequence. But before that let us study the **attention** function in greater detail.

The first operation in the attention function is to compute the similarity between all queries and keys,

$$QK^T = \begin{bmatrix} — & q_1^T & — \\ — & q_2^T & — \\ & \vdots & \\ — & q_m^T & — \end{bmatrix} \begin{bmatrix} | & | & & | \\ k_1 & k_2 & \dots & k_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} q_1^T k_1 & q_1^T k_2 & \dots & q_1^T k_n \\ q_2^T k_1 & q_2^T k_2 & \dots & q_2^T k_n \\ \vdots & \vdots & & \vdots \\ q_m^T k_1 & q_m^T k_2 & \dots & q_m^T k_n \end{bmatrix} \tag{248}$$

Here each row corresponds to matching a single query, q_i, with every possible key, k_j for $j = 1, \dots, n$. Applying

[29] The value of $\sqrt{d}$ is justified theoretically as follows. Let u and v be two zero mean, unit variance random vectors in $\mathbb{R}^d$. Then $u^T v$ has zero mean and variance d. So the distribution of $u^T v / \sqrt{d}$ has zero mean and unit variance.

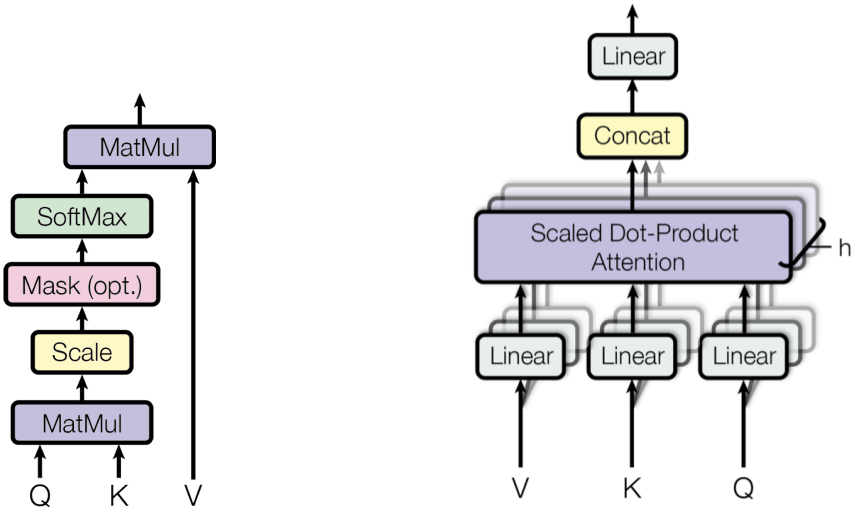


Figure 110: Dot-product attention (left) and multi-head attention (right) [85].

softmax row-wise gives

$$\textbf{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) = \begin{bmatrix} \textbf{softmax}\left(\frac{q_1^T k_1}{\sqrt{d}}, \frac{q_1^T k_2}{\sqrt{d}}, \ldots, \frac{q_1^T k_n}{\sqrt{d}}\right)^T \\ \vdots \\ \textbf{softmax}\left(\frac{q_m^T k_1}{\sqrt{d}}, \frac{q_m^T k_2}{\sqrt{d}}, \ldots, \frac{q_m^T k_n}{\sqrt{d}}\right)^T \end{bmatrix} = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix} \tag{249}$$

where each row can be thought of a providing a normalized weight vector $a_i \in \mathbb{R}^n$. Finally, the attention function constructs each output as a weighted combination of the value vectors, $\{v_1, \ldots, v_n\}$,

$$\textbf{attention}(Q, K, V) = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix} V = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix} = \begin{bmatrix} a_{11} v_1^T + a_{12} v_2^T + \ldots + a_{1n} v_n^T \\ a_{21} v_1^T + a_{22} v_2^T + \ldots + a_{2n} v_n^T \\ \vdots \\ a_{m1} v_1^T + a_{m2} v_2^T + \ldots + a_{mn} v_n^T \end{bmatrix} \tag{250}$$

That is, the i-th output is constructed as

$$\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \tag{251}$$

where α_j is the j-th component of a_i. The transpose in the equations above come from the fact that we have arrange the inputs and outputs as rows.

9.2.1 Masking

Without masking every token can attend to every other token in the sequence. Masking allows us to control how each token interacts with other tokens so can be used, for example, to enforce causal attention—output tokens can only be affected by input tokens that appear earlier in the sequence.

The masked attention operator is defined as

$$\textbf{masked-attention}(Q, K, V, M) = \textbf{row-normalize}\left(\exp\left(\frac{QK^T}{\sqrt{d}}\right) \odot M\right) V \tag{252}$$

where $M \in \{0, 1\}^{m \times p}$ is a 0-1 matrix. This is equivalent to setting masked values in QK^T to $-\infty$ and then applying softmax as we did with unmasked attention since $e^z \to 0$ as $z \to -\infty$, i.e., the value vectors corresponding to masked entries receive zero attention weight.

To implement causal attention we can provide a lower triangular mask with all non-zero elements equal to one,

$$M = \begin{bmatrix} \blacksquare & & & & \\ \blacksquare & \blacksquare & & & \\ \blacksquare & \blacksquare & \blacksquare & & \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{bmatrix}$$

Other sparsity patterns are also possible and may be used depending on the application.

Figure 111: Self-attention (left) versus cross-attention (right).

9.2.2 Multi-head Attention

If the attention mechanism is so good, then maybe more of them will be better. Multi-head attention performs dot-product attention on multiple smaller dimensional queries, keys and values in parallel. We have,

$$Z_i = \textbf{attention}(QW_i^{(Q)}, KW_i^{(K)}, VW_i^{(V)}) \tag{253}$$

for $i = 1, \ldots, h$ where h is the number of heads. The queries, keys and values for each head are linear transformations of the original queries Q, keys K and values V. Here $W_i^{(Q)}$, $KW_i^{(K)}$, $VW_i^{(V)}$ are the parameters of the linear transform.[30] Note that in the expression above we post-multiply by the parameter matrices to perform the linear transform since tokens are arranged row-wise.

An illustration of multi-head attention is shown in Figure 110(right). Each head produces an output $Z_i \in \mathbb{R}^{m \times p}$ (where m is the number of inputs). These outputs are linearly combined to produce the final output of the multi-head attention layer as,

$$y = \begin{bmatrix} Z_1 & \ldots & Z_h \end{bmatrix} W^{(O)} \tag{254}$$

$$\in \mathbb{R}^{m \times d} \tag{255}$$

where $W^{(O)} \in \mathbb{R}^{hp \times d}$ are learnable parameters. Once again we post-multiple. Usually we set the dimensionality of each head to $p = d/h$.

9.2.3 Self-attention versus Cross-attention

Let us return to the question of where the queries, keys and values come from. We saw at the start of the lecture that we can tokenize a sequence of, say, words, and represent them as vectors in n-dimensional space. Each of these tokens is associated with a query, key and value, which we obtain via a linear transform. For the i-th token we have,

$$q_i = W^{(Q)T} x_i, \quad k_i = W^{(K)T} x_i, \quad v_i = W^{(V)T} x_i \tag{256}$$

or more succinctly,

$$Q = XW^{(Q)}, \quad K = XW^{(K)}, \quad V = XW^{(V)} \tag{257}$$

where as usual for transformers we have stacked tokens (transposed) row-wise to form X.

This is called **self-attention** because the queries come from the same source of information as the keys and values. An alternative mechanism is **cross-attention** where the queries are constructed as a linear function of tokens from another branch (i.e., separate input sequence),

$$q_i = W^{(Q)T} z_i, \quad k_i = W^{(K)T} x_i, \quad v_i = W^{(V)T} x_i \tag{258}$$

where $\langle x_1, \ldots, x_m \rangle$ and $\langle z_1, \ldots, z_{m'} \rangle$ are different sequences. An illustration is shown in Figure 111.

9.2.4 Visualising Attention

There have been various attempts to understand the workings of transformers by visualising the attention mechanism. Here researchers are usually interested in which input tokens influence which output tokens. Ignoring multi-head attention, recall that an output token is generated by comparing its query with keys for every other token and then using the normalized similarity to form a weighted combination of associated values,

$$y_i^T = \textbf{softmax}\left(\frac{q_i^T k_1}{\sqrt{d}}, \frac{q_i^T k_2}{\sqrt{d}}, \ldots, \frac{q_i^T k_n}{\sqrt{d}} \right)^T V \tag{259}$$

So one way to visualise the influence of tokens on each other is to visualise the attention weights as show in Figure 112. This can also be done by displaying the attention matrix as a heat map, and various other visualisation techniques have also been proposed in the literature.

[30] A bias parameter is generally not used since layer normalisation, which we will meet shortly, removes any linear offsets.

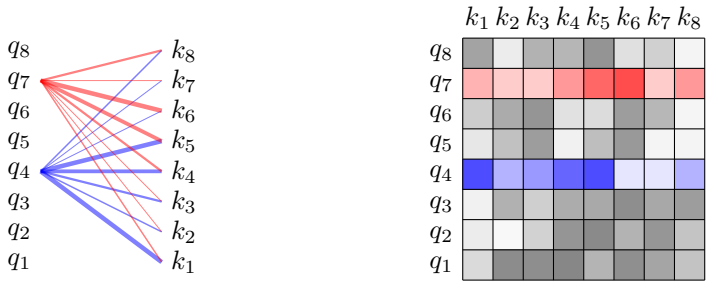


Figure 112: Visualising attention. Each output token y_i *attends* to its most influential input tokens x_j in a weighted fashion. This can be visualised as the cosine similarity between the queries and keys (bipartite graph or row-normalised heatmap).

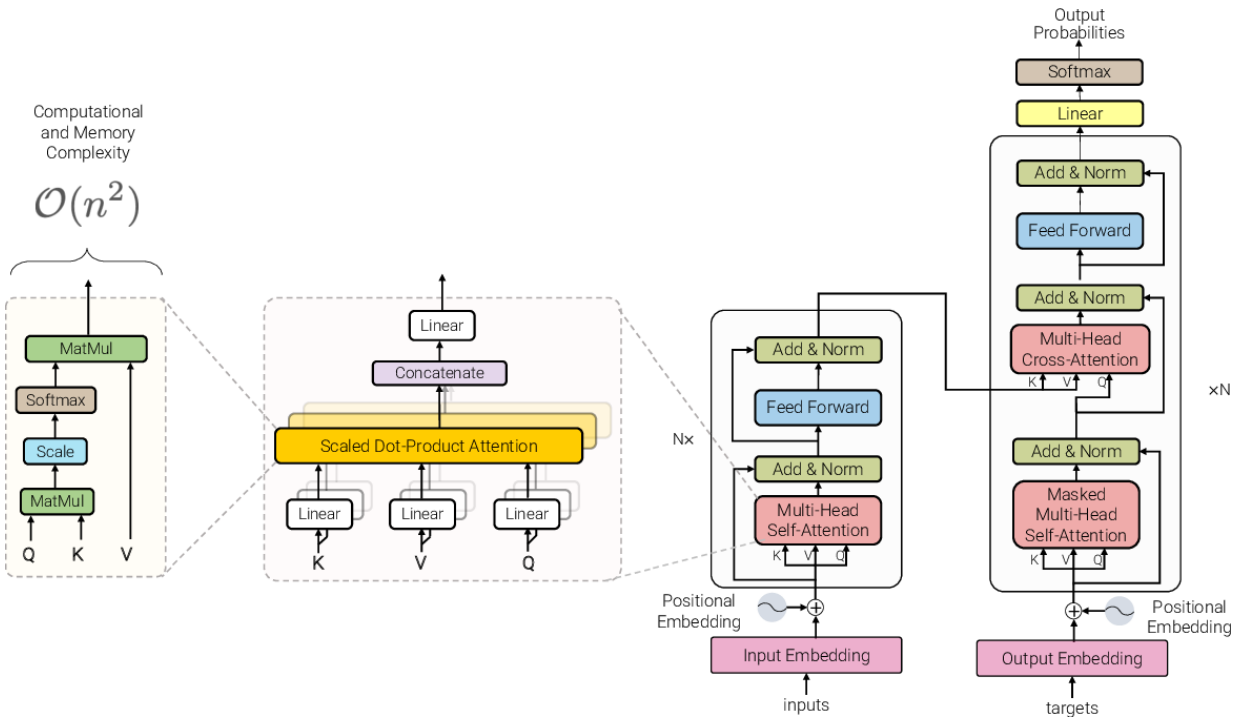


Figure 113: The transformer model [85].

9.3 The Transformer

We are almost ready to present the complete transformer architecture. There is just one more component that we need to discuss, **layer normalisation**. Unlike batch-norm that we saw in earlier lectures, layer normalisation is applied to feature values within a single sample. Samples within a batch are treated independently and thus the operation can be applied at test time without violating any independence assumptions or leaking information from one sample to another.

Let $y \in \mathbb{R}^n$ and $x \in \mathbb{R}^n$ be the input and output for a single sample processed by layer normalisation. Then,

$$y_i = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{260}$$

where μ and σ are the mean a standard deviation of x_i, respectively. Quantities γ and β are learnable parameters. We have written the expressions in vector form for simplicity. Extending to tensor feature maps is straightforward, and performed by computing the mean and standard deviation over every element in the feature map (but still treating each sample in the batch independently).

We have now seen all the building blocks used in the transformer model. It's time to put the blocks together. The full transformer architecture [85] is shown in Figure 113. Originally designed for language translation, the transformer accepts two sequences, an input sequence and a target sequence.[31] These can be two different languages, for example, English and French. The target sequence may be a partial sequence, and just like language models discussed in the previous lecture, the goal is to predict the (probability distribution over the) next token. The important thing about

[31] We'll see many uses beyond language translation later in the lecture.

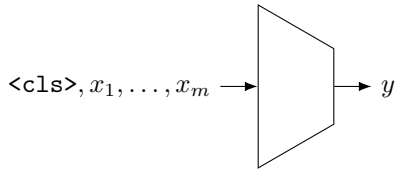


Figure 114: Using transformers for sequence classification.

transformers is that the prediction is conditioned on a very long sequence of previous tokens, not the simple five or six word sentences that we saw previously, making them very powerful AI language models.[32]

The input sequence goes through various layers of multi-head scaled dot-product attention. Here queries, keys and values are obtained by self-attention. The output of each multi-head attention block is added back to the input and layer normalisation applied. Further processing by a two-layer perceptron with ReLU activation function and residual connection, followed by layer normalization then prepares the output for the next layer. This stack, typically consisting of 6–12 blocks is called the transformer encoder.

On the target sequence side, similar processing is applied, in what we call the transformer decoder. In the decoder stack, each block consists of masked multi-head attention to ensure that generated sequences cannot look forward in time. The block also uses cross-attention in a second stage of multi-head scaled dot-product attention processing, the keys and values being obtained from the corresponding layer from the encoder side. The same architecture of a residual single-layer perceptron and layer normalisation is used to produce the output for the next block.

At the top of the decoder is a linear layer followed by softmax. This gives output probability vectors associated with each token. As we will see, the distribution is typically trained to predict the next token in the target sequence, but other training objectives are also possible.

9.3.1 Training and Inference

Training of transformers for language translation works in the same way as training a RNN language model. The first language is tokenized to produce an input sequence for the encoder, $\langle w_1, \ldots, w_m, \texttt{<pad>}, \ldots \rangle$, which may include some special padding tokens if the output sequence is longer than the input. Likewise, the second language is tokenized to produce a target sequence for the decoder, $\langle \texttt{<sos>}, v_1, \ldots, v_n, \texttt{<eos>} \rangle$. The output of the decoder is then trained to minimize the next-token cross-entropy loss where t-th output after passing through softmax is a probability distribution of the next work given the full first language sequence and the partially completed second language sequence up to position $t-1$, i.e., $p(v_t \mid w_1, \ldots, w_m, v_1, \ldots, v_{t-1})$.

There is another way to train transformers for tasks such as (conditional) language generation (e.g., ChatGPT), where we only require the decoder stack. This proceeds as above but without any input tokens or cross-attention, i.e., with the encoder removed. Other variants include bi-directional prediction and masked token prediction where we predict a token somewhere in the middle of a sequence. The resulting models are then used as generic text encoders for some downstream task.

Inference for language translation or language generation is done using auto-regressive next token prediction as we did with RNN language models. Here the target sequence is updated each time a new token is sampled. The input sequence is left unchanged.

Sometimes we are interested in using transformer models as classifiers (e.g., to classify the sentiment of a piece of text or, as we will see later, classifier an image that has been appropriately tokenized). Here we introduce a special <cls> token, usually added at the start of the sequence. We only care about the output associated with this token, which is further processed through a multi-layer perceptron head and trained using cross-entropy loss. Rather than auto-regressive prediction, the transformer now behaves as a sequence classifier as shown in Figure 114.

9.4 The Visual Transformer and Applications

Unlike language, images are already in a numeric format, but they do not present as a 1D sequence. One option is to serialize the image as a sequence of pixels (i.e., flatten the image in some raster scan order) and treating each pixel as a token. However, pixels by themselves are very uninformative. Compare the information in a single pixel, for example, to that of an English or Chinese word. Not to mention the fact that there are way too many pixels in an

[32] Ilya Sutskever (of AlexNet fame) makes this point with a compelling story: Image that you are reading a crime novel. In the first chapter the characters are introduced and a murder occurs. A detective appears. Throughout the remainder of the novel clues are presented but the identity of the murderer remains a mystery. In the final chapter, the detective is about to reveal whodunit. But you, as the reader, are already guessing. You are piecing together the complicated facts presented through the novel to make your prediction. You have read tens of thousands of words to get to this point. Accurately predicting the name of the murderer is good evidence that you have, in some sense, understood and reasoned about the story.

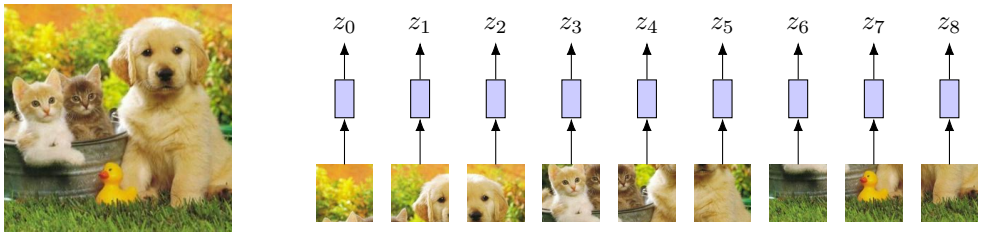


Figure 115: Tokenizing images.

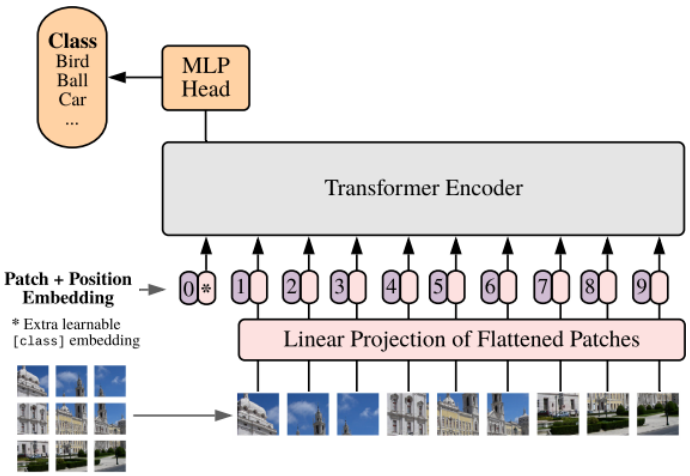


Figure 116: The visual transformer (ViT) [22].

image to treat each one as a token. A better approach is to divide the image into patches and represent each patch as a visual word, the combination of which describe the image.

This is image tokenization. We divide a $(3 \times H \times W)$-image into a grid of $P \times P$ patches. Each patch can be represented by a vector of length $3P^2$ by flattening the pixels in the patch. We can then arrange the patches in some pre-defined raster scan order (e.g., top-left to bottom-right) as shown in Figure 115. The $3P^2$-dimensional vectors representing each patch are passed through a learned linear layer to obtain the image tokens.

The visual transformer (ViT) model proposed by Dosovitskiy et al. [22] uses this approach to tokenize images. It then uses a transformer encoder model (with special class token `<cls>`) as an image classifier as shown in Figure 116. Just like a language model transformer, the visual transformer includes position encoding added to the patch embeddings, and the output token associated with the `<cls>` token is passed through a multi-layer perceptron with softmax activation.

9.4.1 Self-supervised Image Feature Learning

We can train the ViT model using self-supervision by a process known as model (self-)distillation unlike language models, which train for next-token prediction (also in a self-supervised manner). Once trained the `<cls>` token gives a good image representation that can be used for downstream tasks such as image retrieval, video segmentation, and zero-shot prediction. Remarkably the attention maps in DINO have also been found to be useful for (unsupervised) image segmentation and salience detection.

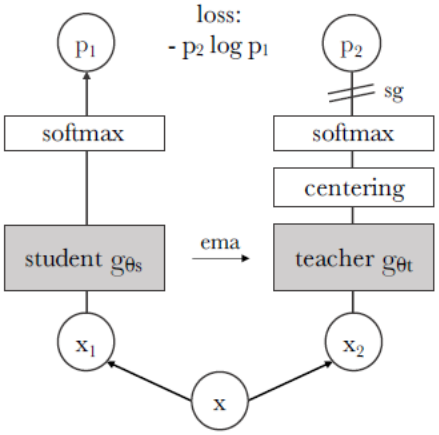
The DINO algorithm [14] (later improved by DINOv2 [69]) uses a pair of models with the same architecture, known as the **teacher** and **student** models. The idea is for the two models to learn to produce the same representation of slightly different random crops and transformations of an image. Figure 117 shows the DINO training setup and pseudo-code. Mathematically, we can write

$$x_1, x_2 = T_1(x), T_2(x) \tag{261}$$

$$\theta_{t+1}^{(1)} = \theta_t^{(1)} - \eta \frac{\mathrm{d}}{\mathrm{d}\theta^{(1)}} L^{\mathrm{ce}}\left(p(x_1; \theta_t^{(1)}), p(x_2; \theta_t^{(2)})\right) \tag{262}$$

$$\theta_{t+1}^{(2)} = \alpha\theta_t^{(2)} + (1-\alpha)\theta_{t+1}^{(1)} \tag{263}$$

where subscript 1 indicates the student network and 2 indicates the teacher network consistent with Figure 117. The first line extracts two different random crops from the same image; the second line updates the student network parameters via gradient descent on a cross-entropy loss between student and teacher outputs; the last line updates the



Algorithm 1 DINO PyTorch pseudocode w/o multi-crop.

```
# gs, gt: student and teacher networks
# C: center (K)
# tps, tpt: student and teacher temperatures
# l, m: network and center momentum rates
gt.params = gs.params
for x in loader: # load a minibatch x with n samples
    x1, x2 = augment(x), augment(x) # random views

    s1, s2 = gs(x1), gs(x2) # student output n-by-K
    t1, t2 = gt(x1), gt(x2) # teacher output n-by-K

    loss = H(t1, s2)/2 + H(t2, s1)/2
    loss.backward() # back-propagate

    # student, teacher and center updates
    update(gs) # SGD
    gt.params = l*gt.params + (1-l)*gs.params
    C = m*C + (1-m)*cat([t1, t2]).mean(dim=0)

def H(t, s):
    t = t.detach() # stop gradient
    s = softmax(s / tps, dim=1)
    t = softmax((t - C) / tpt, dim=1) # center + sharpen
    return - (t * log(s)).sum(dim=1).mean()
```

Figure 117: Self-supervised image feature learning using the DINO algorithm [14].

paramaters of the teacher network as an **exponential moving average** (EMA) of the student network paremeters instead of gradient descent. Here α is a momentum factor that controls how slowly the teacher parameters track the student parameters, and typically set close to one. This helps prevent trivial solutions such as the model collapsing to produce a constant feature output (i.e., ignoring the input). To further mitigate against **model collapse** the teacher output is sharpened using softmax with a low temperature parameter. The logits from the teacher network are also centered by subtracting the average of past logits. DINOv2 [69] does a comprehensive analysis of different model components and adds various other tricks to further improve learning and avoid collapse.

There is nothing in the DINO algorithm that is specific to transformers, and the method has been demonstrated on architectures other than ViT too.

9.4.2 Object Detection with Transformers

One really novel application of visual transformers is object detectors. Carion et al. [13] proposed the detection transformer, or DETR, that uses a transformer decoder with learnable queries (i.e., target tokens) whose outputs correspond to latent object detections. These outputs are passed through a prediction head to regress a bounding box and object category (similar to R-CNN).

The full DETR architecture is depicted in Figure 118. Briefly, the input tokens to the encoder are embedded image patches with positional encoding, much like the ViT model discussed above. The output tokens are passed to the decoder stack of the transformer where the tokens are learnable parameters and part of the model rather than data. DETR calls these object queries. You can think of them as latent anchor proposals that condition the model to make bounding box predictions. The output tokens are decoded into bounding boxes and object scores using a multi-layer perceptron.

The model parameters (including the decoder object queries) are trained end-to-end using a bipartite matching loss to avoid duplicate detections. Here a bipartite graph is constructed between predicted and ground-truth detections. The best matching predictions are considered positives and the remaining detections treated as negatives.

9.4.3 Other Applications

The use of transformers in computer vision is a very active research area. Other applications include video understanding, image and video generation, and 3D modelling. Novel applications and release of new pre-trained models are being reported almost every day.

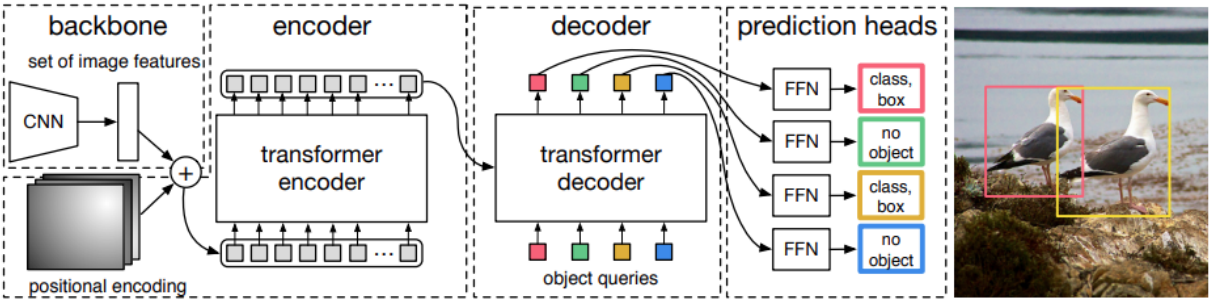


Figure 118: The detection transformer (DETR) [13].

10 Advanced Topics

Time permitting, in the final weeks of the course we cover a selection of advanced topics that build on the fundamentals studied thus far. Some topics may be presented by expert guest lecturers.

An example of possible topics include:

- variational auto-encoders (VAEs) [49, 50]
- diffusion models and image generation [81, 41, 77]
- differentiable optimization and deep declarative networks [34, 35, 4, 3, 32]
- neural radiance fields (NeRFs) [67] and gaussian splatting [47]
- implicit neural representations (INRs), coordinate networks and signed distance functions (SDFs) [70, 53]
- foundation models for monocular depth estimation [88, 37] and geometric 3D vision [86, 58]

Concluding Remarks

This course has taken you from the basics to some of the latest and most advanced topics in deep learning. You are now ready to explore, understand and build upon the remarkable progress that has been made in this field over the last decade. Some models, especially large language models and generative AI, will appear magical when first encountered. But don't get sucked into the hype—these intelligent machines are not (yet) sentient or self-aware. To paraphrase the philosopher and cognitive scientist Daniel Dennett, they have competence without comprehension. Under the hood they are nothing more than a sequence of simple mathematical operations—parametrized linear transformations and non-linear activations—with some clever engineering and trained on lots and lots of data.

References

[1] M. Abadi and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Susstrunk. SLIC superpixels compared to state-of-the-art superpixel methods. *PAMI*, 24(11):2274–2282, 2012.

[3] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter. Differentiable convex optimization layers. In *NeurIPS*, 2019.

[4] B. Amos. *Differentiable Optimization-Based Modeling for Machine Learning*. PhD thesis, Carnegie Mellon University, 2019.

[5] P. Anderson, B. Fernando, M. Johnson, and S. Gould. SPICE: Semantic propositional image caption evaluation. In *ECCV*, 2016.

[6] P. Anderson, B. Fernando, M. Johnson, and S. Gould. Guided open vocabulary image captioning with constrained beam search. In *EMNLP*, 2017.

[7] P. Anderson, X. He, C. Buehler, D. Teney, M. Johnson, S. Gould, and L. Zhang. Bottom-up and top-down attention for image captioning and visual question answering. In *CVPR*, 2018.

[8] P. Anderson, Q. Wu, D. Teney, J. Bruce, M. Johnson, N. Sunderhauf, I. Reid, S. Gould, and A. van den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *CVPR*, 2018.

[9] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. Lawrence Zitnick, and D. Parikh. VQA: Visual question answering. In *ICCV*, pages 2425–2433, 2015.

[10] Y. Ben-Shabat, C. H. Koneputugodage, and S. Gould. DiGS: Divergence guided shape implicit neural representation for unoriented point clouds. In *CVPR*, 2022.

[11] S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge, 2004.

[12] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[13] N. Carion, A. Kirillov, F. Massa, G. Synnaeve, N. Usunier, and S. Zagoruyko. End-to-end object detection with transformers. In *ECCV*, 2020.

[14] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin. Emerging properties in self-supervised vision transformers. In *ICCV*, 2021.

[15] J. Carreira and A. Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *CVPR*, 2017.

[16] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *CVPR*, 2016.

[17] A. Criminisi. Microsoft Research Cambridge (MSRC) object recognition pixel-wise labeled image database (version 2), 2004.

[18] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control Signal Systems 2*, 1989.

[19] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39, 1977.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *ACL*, 2019.

[22] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[23] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

[24] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[25] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2009 (VOC2009) Results, 2009.

[26] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient graph-based image segmentation. *IJCV*, 2004.

[27] R. Girshick. Fast R-CNN. In *ICCV*, 2015.

[28] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

[29] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, 2010.

[30] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In *ICML*, 2013.

[31] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[32] S. Gould. Lecture notes on differentiable optimisation in deep learning, 2024. URL `https://users.cecs.anu.edu.au/~sgould/papers/isaac22-lecture-notes.pdf`.

[33] S. Gould, T. Gao, and D. Koller. Region-based segmentation and object detection. In *NeurIPS*, 2009.

[34] S. Gould, B. Fernando, A. Cherian, P. Anderson, R. Santa Cruz, and E. Guo. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. Technical report, Australian National University (arXiv:1607.05447), July 2016.

[35] S. Gould, R. Hartley, and D. Campbell. Deep declarative networks. *PAMI*, 2021.

[36] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004.

[37] J. He, H. Li, W. Yin, Y. Liang, L. Li, K. Zhou, H. Liu, B. Liu, and Y.-C. Chen. Lotus: Diffusion-based visual foundation model for high-quality dense prediction, 2024.

[38] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *CVPR*, 2015.

[39] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[40] K. He, G. Gkioxari, P. Dollar, and R. Girshick. Mask R-CNN. In *ICCV*, 2017.

[41] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. In *NeurIPS*, 2020.

[42] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[43] Y. Hong. *Learning Language-Guided Visual Navigation*. PhD thesis, Australian National University, 2023. URL `https://openresearch-repository.anu.edu.au/items/d9706162-cfe2-49a6-81bd-047657268587`.

[44] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[45] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[46] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *CVPR*, 2015.

[47] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023.

[48] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2014.

[49] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[50] D. P. Kingma and M. Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019.

[51] A. Kirillov, K. He, P. Dollar, R. Girshick, and C. Rother. Panoptic segmentation. In *CVPR*, 2018.

[52] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollar, and R. Girshick. Segment anything. In *ICCV*, 2023.

[53] C. H. Koneputugodage, Y. Ben-Shabat, D. Campbell, and S. Gould. Small steps and level sets: Fitting neural surface models with point guidance. In *CVPR*, 2024.

[54] R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, M. S. Bernstein, and F.-F. Li. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *IJCV*, 123:32–73, 2017.

[55] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[56] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

[57] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *IEEE*, 1998.

[58] V. Leroy, Y. Cabon, and J. Revaud. Grounding image matching in 3d with MASt3R, 2024.

[59] F.-F. Li, M. Andreeto, M. Ranzato, and P. Perona. Caltech 101. Technical report, California Institute of Technology, 2022.

[60] J. Li, D. Li, C. Xiong, and S. Hoi. BLIP: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *CVPR*, 2022.

[61] T. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and C. L. Zitnick. Microsoft COCO: Common objects in context. In *ECCV*, 2014.

[62] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. SSD: Single shot multibox detector. In *ECCV*, 2016.

[63] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

[64] A. López-Cifuentes, M. Escudero-Viñolo, J. Bescós, and Á. García-Martín. Semantic-aware scene recognition. *Pattern Recognition*, 2020.

[65] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

[66] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *ICLR*, 2013.

[67] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[68] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. *NeurIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[69] M. Oquab, T. Darcet, T. Moutakanni, H. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski. DINOv2: Learning robust visual features without supervision, 2023.

[70] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *CVPR*, 2019.

[71] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017.

[72] J. Pennington, R. Socher, and C. Manning. GloVe: Global vectors for word representation. In *EMNLP*, 2014.

[73] S. J. Prince. *Understanding Deep Learning*. The MIT Press, 2023.

[74] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision. In *ICML*, 2021.

[75] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *CVPR*, 2016.

[76] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[77] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *CVPR*, 2022.

[78] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015.

[79] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

[80] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *NeurIPS*, 2014.

[81] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *NeurIPS*, 2015.

[82] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014.

[83] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. Selective search for object recognition. *IJCV*, 104(2):154–171, 2013.

[84] V. N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, 1998.

[85] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[86] S. Wang, V. Leroy, Y. Cabon, B. Chidlovskii, and J. Revaud. DUSt3R: Geometric 3d vision made easy. In *CVPR*, 2024.

[87] B. Widrow and M. E. Hoff. Adaptive switching circuits. *IRE WESCON Convention Record*, 1960.

[88] L. Yang, B. Kang, Z. Huang, X. Xu, J. Feng, and H. Zhao. Depth anything: Unleashing the power of large-scale unlabeled data. In *CVPR*, 2024.

[89] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023.

[90] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang. Random erasing data augmentation. In *AAAI*, 2020.