Layer-wise Iterative Pruning for Neural Network

Wei Chu

Australian National University, ACT, Australia
u6133110@anu.edu.au

Abstract. In this work, we present two methods for layer-wise iterative pruning: the cosine pruning and the Taylor expansion pruning. These pruning algorithms will automatically select unimportant neurons to prune and dynamically reconcile the connected layers. The cosine pruning is more volatile than the Taylor expansion pruning, oftentimes, need retrain to recover the accuracy. The Taylor expansion pruning is much stable, the size of networks can be reduced without comprising any accuracy even with no retraining. The Taylor expansion pruning can reduce 90% of the hidden neurons for a fully-connected network trained on bank marketing dataset, a 28% reduction to AlexNet and a 40% reduction to VGG16 on ImageNet dataset.

Keywords: prune · redundant · iterative · CNN

1 Introduction

In the last couple years, deep learning has gained considerable attentions and achieved state-of-art solutions in many research fields. More and more often we see applications employ very large and complicated neural networks in order to enjoy some accuracy benefits [9, 10]. Adding more parameters do not make the networks efficient with respect to speed and size. Oftentimes, in the real world deployments, tasks are carried out in a timely fashion on a resource limited platform. Minimizing the size of network is absolutely necessary. We propose an iterative pruning approaches that can reduce the size of networks while without comprising the model's predicting ability.

The following section will start with a discussion to illustrate the effects of redundant neurons. Then, we move to the two pruning strategies: the cosine pruning[3] and the Taylor expansion pruning[6]. Comparison will be made between the two on a feed-forward network. Further research will be carried with Taylor expansion approach solely to explore pruning for large networks like VGG[10].

2 Effects of redundant neurons

Pruning is to eliminating redundant neurons that do not contribute much to the predictions. Here, neurons can be considered as intermediate channels that make deliveries to both ends, via forward and backward propagation. If all neurons are delivering the same message, then there will not be much to learn for the networks. This is the basic idea of redundant neurons and we show its effects with the mathematical expressions as follows. Demonstrations are constrained to feed-forward nets, more general proof could be extended similarly.

2.1 Effects of Redundant Neurons

The setting is a 3-layer feed-forward net with input x and output of a binary variable of y, activation function is *sigmoid*, and cross-entropy is adopted for the loss. The complete information flow process is described with equations below,

$$
\begin{aligned}
&W^1 x = z^1 \\
&a^1 = sigmoid(z^1) \\
&W^2 a^1 = z^2 \\
&a^2 = sigmoid(z^2) \\
&J(a^2, y) = -y \log a^2 - (1-y)\log(1-a^2)
\end{aligned}
$$

where capital letters represent matrix and small letters represent vector. The bias is concatenated into input $x = [x_1\ x_2\ \cdots\ 1]^T$.

Assuming W^1 is the matrix that has redundant neurons. Let's examine the bidirectional information flows relating to W^1.

The forward information flow is straightforward, any identical rows $v_i \in W^1$ will produce equivalent z_i^1. The backward information flow is a bit complicated which involves gradients. Calculate the back-propagation regrading W^1, we have

$$\frac{\partial J}{\partial z^2} = a^2 - y \tag{1}$$

$$\frac{\partial J}{\partial z^1} = W^{2T}\frac{\partial J}{\partial z^2} * (a^1 * (1 - a^1)) \tag{2}$$

$$\frac{\partial J}{\partial W^1} = \frac{\partial J}{\partial z^1}x^T \tag{3}$$

$*$ is hadamard product. Expand using chain rule, we obtain,

$$\frac{\partial J}{\partial W^1} = [W^{2T}(a^2 - y) * (a^1 * (1 - a^1))]x^T$$

update W^1 with gradient descent, we obtain

$$W^1 = W^1 - \alpha\frac{\partial J}{\partial W^1}$$

Without loss of generality, we assume $W^1 \in \mathbb{R}^{2\times 2}$, $W^2 \in \mathbb{R}^{1\times 2}$, then a^2 and y become scalars. A visualization to the setting is shown in Fig.1.

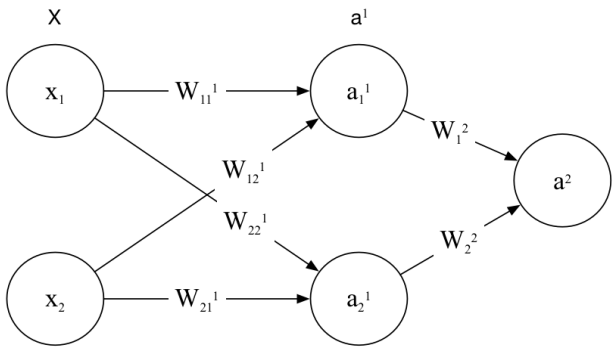


Fig. 1: A 3-layer Feed-Forward Network Producing Scalar Output

Rewrite the gradients of W^1 to matrix form, we have

$$dW^1 = \begin{bmatrix} (a^2 - y)a_1^1(1 - a_1^1)W_1^2x_1 & (a^2 - y)a_1^1(1 - a_1^1)W_1^2x_2 \\ (a^2 - y)a_1^1(1 - a_2^1)W_2^2x_1 & (a^2 - y)a_2^1(1 - a_2^1)W_2^2x_2 \end{bmatrix}$$

If W^1 (before updating) has included some redundant units, for example, the second row is a clone of the first row,

$$W^1 = \begin{bmatrix} W_1^1 & W_2^1 \\ W_1^1 & W_2^1 \end{bmatrix}$$

then the gradients will update symmetrically. In other words, no matter how many epochs spent on training this model, neuron one and neuron two will never produce distinctive outcomes. Note that to achieve this theoretical result we must have $W_1^2 = W_2^2$. This is a reasonable assumption because if the neurons generating identical information, then the weights assigned to them for the next layer should also be identical. In practice, it might differ.

3 Methods

A modified *distinctiveness*[3] approach and a Taylor expansion approach[6] are presented for iteratively pruning networks. *Distinctiveness* method provides ranks on neurons based the cosine similarity. The most similar or dissimilar neurons will be pruned. Alternatively, we can replace cosine with other metrics like Spearman's correlation[1], mutual information[6] etc. Molchanov et al. [6] proposed an optimization based pruning strategy – ranking neurons with Taylor expansion. Details of both methods will be discussed in the following sections.

3.1 Cosine Similarity for Recognizing Redundant Neurons

Cosine similarity takes two neurons as inputs, then outputs a cosine score. The detail algorithm is described as below.

Algorithm 1 Iterative Pruning Algorithm

procedure PRUNING(a, b) ▷ a, b are two neurons
    Objective matrix W to prune
    normed_W ← W / norm(W, 1)
    W_cosine ← $normed\_W \cdot normed\_W^T$
    abs_W_cosine ← abs(W_cosine)
    remove_ids ← Compare abs_W_cosine to low and high separation angular ▷ i.e., a list such as [[1,2], [2,4], [3,5]...]
    connoted_ids ← Graph search find all connected components ▷ i.e., a list such as [[1,2, 4], [3,5]...]
    for connected_set ∈ connoted_ids **do**
        if low angular seperation **then**
            $W[connected\_set[-1]] \leftarrow sum(connected\_set[:-1], 1)$
            $W[connected\_set[:-1]] \leftarrow 0$
            $next\_W[connected\_set[-1]] \leftarrow sum(connected\_set[:-1], 1)$
            $next\_W[connected\_set[:-1]] \leftarrow 0$
        if high angular seperation **then**
            $W[connected\_set] \leftarrow 0$
            $next\_W[connected\_set] \leftarrow 0$
    W ← Remove rows if all row elements are 0
    next_W ← Remove columns if all columns elements are 0
    Retrain the model with the new W and new next_W
    Repeat above process

Compute Cosine Similarity Row-wise normalization is conducted on the targeted weight matrix. The purpose is to convert matrix product into cosine similarity results. The calculation takes the following formula,

$$cos\theta(v_1, v_2) = \frac{v_1 v_2}{\|v_1\|_2 \|v_2\|_2} = v_1 v_2 \qquad \textbf{if v1 and v2 have norms equal to 1}$$

$$product = normedW^1 \cdot normedW^{1T} = \begin{bmatrix} v_1v_1 & v_1v_2 & \cdots & v_1v_n \\ \vdots & \ddots & & \vdots \\ v_nv_1 & v_nv_2 & \cdots & v_nv_n \end{bmatrix}$$

The *product* outcomes will be comparing to the angular separation thresholds, from here we can get elimination candidates. Regarding the thresholds, we follow the construction in [3]. Angular separation smaller than 15° is considered similar, while angular separation larger than 165° is considered complementary, both are going to be eliminated. The downside of thresholds setting is that, for not too large networks, we might end up with no neurons to prune as no neurons are similar or complementary.

A greedy heuristic is employed for removing neurons. If the thresholds indicate neurons $\{(2,3),(3,5)\}$ are similar, we can either remove $(2,5)$ neurons or remove (3) neuron. The greedy heuristic guarantees to perform the former elimination to eliminate as many as we can. One downside for the greedy approach is that it is likely to remove more than what we actually needed.

Once a neuron is removed, its effects will add back to the network via the similar neurons that kept. This ensures the same amount of information is passed forward, but loss inevitably due to *cos* similar is not exactly 1.

3.2 Taylor expansion

Pruning can be considered as an optimization problem. That is, after the pruning, we do not want the accuracy to change too much, translating into mathematical expressions it is

$$\min_{W'} |\mathcal{C}(\mathcal{D}|W) - \mathcal{C}(\mathcal{D}|W')|$$

where, $\mathcal{D} = \{\mathcal{X}, \mathcal{Y}\}$ is inputs and labels, $\mathcal{C}$ is the cost, W is the layer-wise weights, and W' is the weights after pruning. Another way to understand this model is that, we want to find out which neurons contribute the least to the prediction, and conducting the removal in such fashion.
Using first-order Taylor expansion to simplify the objective function, we have

$$objective = |\mathcal{C}(\mathcal{D}|W) - (\mathcal{C}(\mathcal{D}|W) - \frac{\partial \mathcal{C}}{\partial W} W)| \tag{4}$$

$$= |\frac{\partial \mathcal{C}}{\partial W} W| \tag{5}$$

According to the orignal work by Molchanov et al. [6], high-order items from Taylor expansion can be safely dropped due to the widely-used ReLU diminishing the effects of the reminder. Implementation of the algorithm is presented as below.

Algorithm 2 Rank-based Iterative Pruning Algorithm

```
procedure PRUNING(model)
    neuron_rankings ← sum(grad_l * weight_l)        ▷ accumulation effects over entire train set
    normalized_rankings ← neuron_rankings/L2-norm   ▷ Make it comparable across layers
    prune_candidates ← heap_n_smallest(normalized_rankings)
    prune candidates and reconcile connections
    Repeat above process
```

4 Results and Discussion

4.1 Data

Data is obtained from UCI publication[7] and ImageNet[2]. UCI data is used for feed-forward networks and ImageNet data is used for Convolution Neural Networks (CNN).

The UCI one is a bank marketing dataset with a total of 45211 instances and each instance is labeled to indicate if that client subscribes for the service. 16 attributes are presented in the data, 9 of them are categorical and the rest are numerical. Categorical data is converted to numerical values and then fed into neural networks along with other numerical attributes as features. Data has been split 70% for training and 30% for testing, and then prepossess with mini-batches.

ImageNet data is taking from bees and ants two categories. Each category has 120 images for training and 75 images for testing. Data augmentations and normalization are taken on these images, specifications is handled by `PyTorch DataLoder` inlcuding the mini-batch.

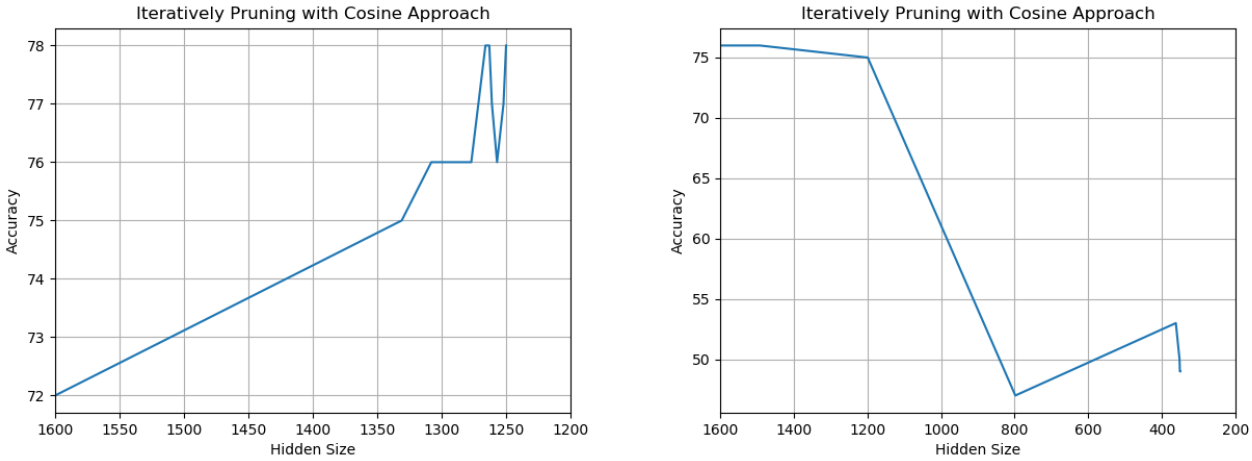
4.2 Experimental Results

The hypermeters are set as follows. We use SGD optimizer for training, momentum is set to 0.9, learning rate is set to $10e-4$. Number of epochs for initial training (or fine-tuning) is set to 20. For iterative pruning, retraining is enforced and each retraining takes 5 epochs.

The feed-forward net is constructed with 3 fully-connected layers, where layer 1 is the inputs, layer 2 is $W^1 \in \mathbb{R}^{in\_features \times 1600}$, and layer 3 is $W^2 \in \mathbb{R}^{1600 \times num_classes}$. We show the results after a single iteration of pruning in Table 2. Note, the Taylor-expansion method is ranking-based, pruning size therefore is adjustable.

Table 1: Test Accuracy With a Single Pruning on Bank Marketing Data

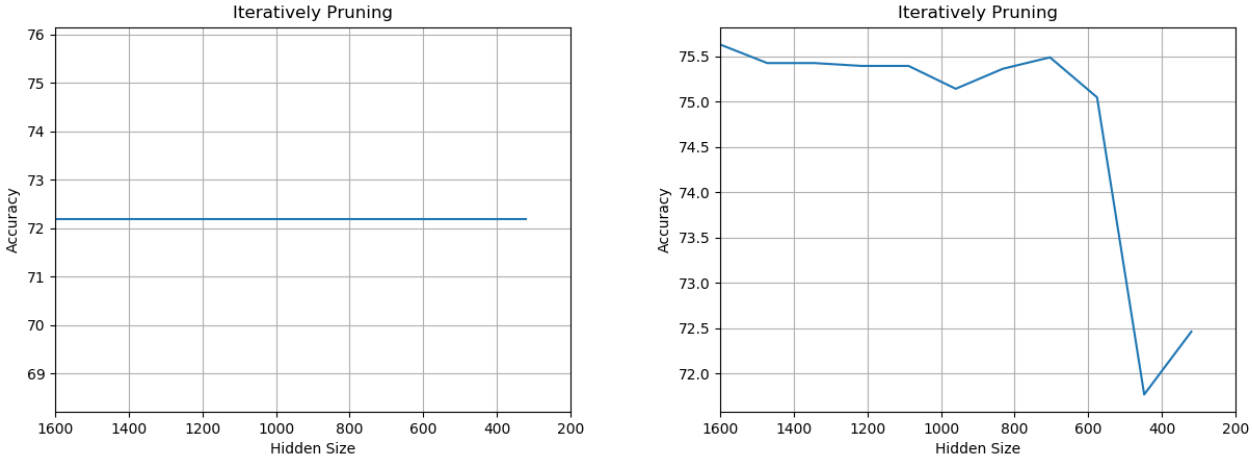
	Cosine(Batch 1)	Taylor(Batch 1)	Cosine(Batch 32)	Taylor(Batch 32)
Hidden Size before pruning		1600		
Hidden Size after pruning		1331		1492
Accuracy before pruning		72%		75%
Accuracy after pruning (no retraining)	59%	72%	57%	75%



(a) Batch size of 1

(b) Batch size of 32

Fig. 2: Iterative Pruning on Bank Marketing Dataset with Cosine-based Approach



(a) Batch size of 1

(b) Batch size of 32

Fig. 3: Iterative Pruning on Bank Marketing Dataset with Taylor-expansion-based Approach

Performance of the Taylor expansion pruning is much stable, the accuracy has maintained without retraining. While, for the cosine pruning, the accuracy cannot be guaranteed without retraining, as we see a sharp drop of

accuracy by 18%. The loss of accuracy indicates that simply adding the redundant information back to non-removed neurons cannot recover all information.

The usage of mini-batch is shown to encourage more aggressive pruning, according to Figure 2 and Figure 3. Without mini-batch, cosine pruning can reduce the size of hidden neurons to 1250; with mini-batch, the number of hidden neurons can be reduced to 400 or lower. The performance of the Taylor expansion pruning is not immune to mini-batch which showing in Figure 3, but impacts of mini-batch is not as server as to cosine pruning, as we only see 3% drop of accuracy.

Implementation of cosine pruning to CNN is complicated and hard to tune, so for now, we only test CNN models with the Taylor expansion pruning.

The Taylor expansion pruning can dramatically reduce the size of large CNN structures. Taking VGG16 as an example, we manage to reduce the number of filters from 4224 to 2560, the size from 537.1MB to 212.7 MB, at a cost of 6% accuracy rates. Comparing to the previous work[5], the author has managed to reduce the size of VGG16 from 538MB to 150MB at a cost of 1% accuracy. Such difference might due to fine-tuning and the data set as well, as in that work, the author uses data from *Kaggle Dogs vs. Cat*.

Table 2: Performance of Taylor Expansion Method on CNN Models

	SimpleConvNet	**VGG16**	**AlexNet**
Num filters pruned per iteration	4	128	32
Accuracy (before)	66.7%	95.42%	92.16%
Accuracy (after)	63.4%	89.54%	88.89%
Num of filters (before)	48	4224	1152
Num of filter (after)	24	2560	832
Model size (before)	20.1MB	537.1MB	228.1MB
Model size (after)	4.4MB	212.7MB	168.7MB

where the *SimpleNet* is structed as the follows.

```
self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(32 * 56 * 56, 50),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(50, 2)
        )
```

5 Conclusion and Future Work

In this work we have evaluated two iterative pruning algorithms: the cosine pruning and the Taylor expansion pruning. Both shown an excellent reducing abilities. The cosine pruning is volatile especially when running along with mini-batch. The Taylor expansion pruning is stable, but when runing along with mini-batch, accuracy is going to be affected more or less. With the bank marketing dataset, we manage to reduce the size to one-tenth of the original network, the cosine pruning did it at a cost of 20% accuracy rates, while the Taylor expansion pruning did it at a cost of 3% accuracy rates. Further experiments of Taylor expansion pruning also show its powerful abilities on pruning CNN filter, overall it can reduce 30% to the original models.

Further work could be done to expand the pruning for language related models like RNN. Besides, knowledge distill models could be a good comparison, as the fundamental purpose is the same as pruning, which is to obtain a smaller network from a larger model.

References

1. Babaeizadeh, M., Smaragdis, P., Campbell, R.H. NoiseOut: A simple way to prune neural networks. *arXiv*:1611.06211 (2016)
2. Deng, J. and Dong, W. and Socher, R. and Li, L.-J. and Li, K. and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. *CVPR09*. URL: http://www.image-net.org/, 2009.
3. Gedeon, T.D. and Harris, D. Network reduction techniques. *Proc. Int. Conf. on Neural Networks Methodologies and Applications*, AMSE, San Diego, vol.2,pp.25-34.1991.
4. Glorot, X. and Bengio, Y. Understading the difficulty of training deep feedforward neural networks. *In Proc. AISTATS*, vol.9,pp.249-256, 2010.
5. Gildenblat, J. Pruning deep neural networks to make them fast and small. Blog, 2017.
6. Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. In *ICLR*, 2017.
7. Moro, S., Cortez, P., and Rita, P. A data-driven approach to predict the success of bank telemarkeing. *Decision Supporting System*, Elsevier, 62:22-31, June 2014.
8. Ng., A. Y., Feature selection, L1 vs. L2 regularization, and rotational invariance. In *ICML*, 2004.
9. Shazeer, N., Azalia, M., Krzysztof, M., Davis, A.,Le, Q.,Hitton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:*1701.06538, 2017.
10. Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.
11. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, pages 19291958, 2014.