Detecting Dead Weights and Units in Neural Networks

by

Utku Evci

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Computer Science Department
New York University
May 2018

Professor Leon Bottou

Feed your head

Acknowledgements

During the process of working towards this work and especially towards the end, I kept asking myself how and why I decided to devote my last semester in NYC writing this thesis. I could take classes instead and have a less stressful last semester. This work was optional and at the time I had already accepted a software development engineer position and It seemed kind of unnecessary. However, I decided to do it for various reasons which are mostly related to a possible future Ph.D. application at the end. Looking back now, I think it was one of the best decisions I have made during my time at NYU.

NYC is a special city and I was one of the people floating in this urban jungle for a while. NYC is grand and there is no time to waste. You can always fail here spending a fortune or work more and more to make it here, you can't even feel comfortable. There are always things to do. Sometimes it feels like what NYC offers is not meant for you. You feel like you are missing stuff all the time and everyone else is doing better than you and you don't have any friend here. It is hard to live here and it is even harder to leave. I will miss waking up on Christopher, biking to CDS on 5th Avenue and spending all of my days on the 5th floor, in front of the computer. I will definitely miss having coffee in Blue Bottle Coffee and Joe's enjoying the first days of the spring.

I would like to thank Jason Weston for his feedback on SVM feature selection. Levent Sagun provided quite valuable comments and feedback about the first two chapter. I would like to thank him for his valuable advice and support during my studies at NYU. He has an important influence on my involvement in Machine Learning research.

Computer Vision class thought by Prof. Fergus was the first time I heard about the pruning idea and I still remember the excitement I had after his class about model compression. After his class, I decided to reproduce the recent pruning research as my class project and had valuable comments from him. I like to thank Prof. Fergus for being my the second reader for the thesis and all his support.

Finally, I would like to thank Prof. Bottou for all his patience and time advising this work. His guidance shaped this thesis and made it something readable. I learned many things about Machine Learning through our conversations and his reading suggestions. More important than that I learned how to motivate my work, how to visualize data and how to decide which experiments to do. He guided me towards the dead unit idea and introduced me to the bias propagation idea. Efficient approximation of the loss change due to this propagation (MRS) was also his idea. This work would have been quite different without his guidance and support. I feel lucky.

Abstract

Deep Neural Networks are highly over-parameterized and the size of the neural networks can be reduced significantly after training without any decrease in performance. One can clearly see this phenomenon in a wide range of architectures trained for various problems. Weight/channel pruning, distillation, quantization, matrix factorization are some of the main methods one can use to remove the redundancy to come up with smaller and faster models.

This work starts with a short informative chapter, where we motivate the pruning idea and provide the necessary notation. In the second chapter, we compare various saliency scores in the context of parameter pruning. Using the insights obtained from this comparison and stating the problems it brings we motivate why pruning units instead of the individual parameters might be a better idea. We propose some set of definitions to quantify and analyze units that don't learn and create any useful information. We propose an efficient way for detecting dead units and use it to select which units to prune. We get 5x model size reduction through unit-wise pruning on MNIST.

Contents

List of Figures

List of Tables

Introduction

With increasing availability of data and computing power, Neural Networks became the most successful tool for learning complex functions. Networks with millions of parameters can be trained through first-order, gradient-based algorithms to minimize a given loss function. In the history of neural networks, many research proposed various architectures and optimization methods to come up with better results. With the increasing computation power, researchers were able to try larger architectures and utilize larger datasets to achieve significant improvements on many benchmarks and practical problems. A downside of these very successful and very big networks is that each network needs a considerable amount of storage and compute power to run. However, we can easily show that the resulting deep networks are highly redundant and had ill-conditioned parameter matrices. Can we reduce this redundancy by removing unnecessary parameters? Optimal Brain Damage paper[23] emphasizes the importance of removing connections (pruning) as a regularizer and a performance-enhancer. It proposes deleting parameters with small saliency and shows that second-order approximations might improve over simple magnitude based saliency measures. It also shows the similarity between weight decay[13] and pruning.

In order to utilize this very large networks in resource-limited real-world environments like mobile devices, we should be able to reduce their storage and energy requirements. Han showed that the famous ImageNet winner AlexNet could be compressed up to 40x[12][11]. However since there is no restriction on which connection to remove, pruned parameter matrices do not follow any specific pattern and therefore require sparse representation to utilize sparsity. One drawback with this approach is that the current sparse matrix operations are not as fast as their dense counterparts. Attacking this issue, various groups focused on channel pruning and introduced some structured sparsity techniques[37],[26],[39].

It is very interesting to see that one can get around 90% sparsity with retraining on many different networks and it looks like this number does not depend on architecture or data. Even a small network with a very small number of convolutional filters end-up having most of its filters dying out during training and we know that starting the training with an overparameterized network is crucial for the best performance. It looks like our optimization methods are not perfect and they are far from utilizing the full power of our networks.

Given a resource and time budget what is the best performance we can get? We can always use the biggest network we can run and then prune, quantize, distill or use other techniques to reduce the size of the network for the repetitive inference task. However, if we know that these low importance connections appear during the early training, we can prune the network during the training. Doing this would bring us no advantage in terms of computation time since the dense operations usually run much faster than the sparse operations below 90% sparsity. So we are better off keeping the dense representation and doing unnecessary calculations on the pruned parameters during the training even if we prune the network early in the training. However, if we prune the whole channels or units at once, we can reduce the size of the parameter matrices right away and decrease the training time without harming the performance.

This thesis demonstrates, compares and criticizes the two main pruning approaches mentioned above: parameter-wise(unconstrained) pruning and unit-wise pruning. We also focus on finding**dead regions: weights/units that are likely to be pruned** that appear the early-training and stay dead after.

In Chapter 1 , we provide a short introduction to neural networks and provide a literature review on pruning neural networks. In Chapter 2 we motivate the idea of dead units by showing that low saliency parameters seem like developing around some particular units. We also compare various saliency measures and conclude that the magnitude-based saliency measure performs best at pruning a group of parameters at once. In Chapter 3 we formally define the dead units as units with low mean replacement penalty (MRP: defined in Chapter 3). We propose using the first order approximation in order to detect the dead units efficiently. We generate dead units by introducing high learning rate for an arbitrary unit during some training and show that the unit most likely stays dead throughout rest of the training. After defining and generating dead units, in Chapter 4, we repeat the experiment in Chapter 2, but this time pruning entire units at once using the efficient scoring function defined in the previous chapter. As a result, we get a greater reduction in network size without requiring a sparse format.

Chapter 1

Background

This chapter introduces necessary notation, concepts and presents the previous work.

1.1 A Very Small list of Abbreviations

1. ANN: Artificial Neural Network
2. CNN: Convolutional Neural Network
3. GD: Gradient Descent
4. SGD: Stochastic Gradient Descent
5. RNN: Recursive Neural Networks
6. MRP: Mean Replacement Penalty
7. MRS: Mean Replacement Saliency
8. #: number of

1.2 Artificial Neural Networks and Notation

Artificial Neural Networks (ANN) is a biologically inspired algorithm for transforming numbers from an input space X to an action space A.

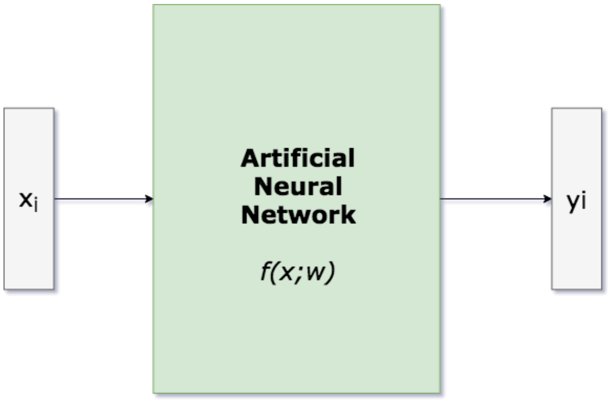


Figure 1.1: A very basic illustration of an ANN as a block diagram

With the advancements in Neuroscience and the invention of computers, the idea of mimicking neurons in our brains with the help of computers attracted quite a bit attention. In 1958, Rosenblatt introduced the building block of ANN's: the Perceptron[33]. In the following years, advancements in the field made it possible to connect many perceptrons **units** together and train them jointly. With the advancements in acceleration hardware and increased availability of data; deep and large ANN's became the best performing algorithm in a wide variety of machine learning tasks. ANN's can have hundreds to millions of units and they are mostly trained through backpropagation. I would like to point out Schmidhuber's review [35] for further reference on the history of ANN's.

One can formulate an ANN as a function parameterized with a vector w, mapping an input x_i to an output $\hat{y}_i$, where i represent the index of the data from a dataset X [Figure 1.1].

$$f(x_i; w) = \hat{y_i}$$

Then we would define a loss function with a range of non-negative real numbers calculated from the individual $\hat{y_i}$'s. Usually there are y_i's we wish $\hat{y_i}$'s to match.

$$L(w) = \frac{1}{m} \sum_{x_i, y_i \in D} l(f(x_i; w), y_i)$$

where m is the total number of samples and $l(\cdot)$ is a non-negative function penalizing bad estimations.

In the supervised setting, this would correspond to the observed target values. We denote them with y_i and the corresponding target dataset as Y. In unsupervised methods, the loss function would depend on only X with size N. We represent the loss function as $L(w)$ removing the data that the loss function intrinsically depends on from the notation.

ANN's can have many different layers/modules inside them. Recursive Neural Networks(RNN)'s allow units to have states and those states are used to calculate the output along with inputs. Units in RNNs learn both how to transform the input, given the state and how to update its state, given the input. Convolutional Neural Networks(CNN)'s are able to utilize spatial dependency in inputs. They use convolution operation and parameter sharing to learn how to extract translation-invariant features from the inputs. The output of CNN's and RNN's are usually transformed further by some matrix transformation and this simple layer is usually named as **Linear** layer. Most of the modules mentioned above would also have nonlinearities applied to their outputs. There is not a strict line between all of these definitions and usually real-world architectures include a combination of different modules and nonlinearities; but they all have one thing in common: All the operations used in these networks are sub-differentiable; so that we can apply chain rule to get $\frac{\partial L(w)}{\partial w_i}$; i.e. gradient of the loss function with respect to every parameter in the network. Efficient calculation of $\frac{\partial L(w)}{\partial w_i}$ for every parameter in the network is named as backpropagation algorithm [34] and it is a fundamental method for optimizing ANN's.

1.2.1 Training DNN's

Optimizing the loss function of an ANN or in other words training an ANN is hard due to its non-convex energy landscape and its huge dimensionality. Almost all ANN's are trained with some version of gradient descent(GD). GD uses backpropagation algorithm to calculate $\frac{\partial L(w)}{\partial w_i}$'s and then updates the parameters with the following formula. The calculation of the $f(x_i; w)$'s for a given dataset is called **forward-pass**. Then we would calculate the average loss $L(w)$. Using chain rule we would then calculate the gradient of the loss $\frac{\partial L(w)}{\partial w}$ and this is called the **backward-pass**. After calculating the gradient we would update the parameters with the following rule.

$$w_i^t = w_i^{t-1} - \alpha \frac{\partial L(w^{t-1})}{\partial w_i^{t-1}}$$

where α is a small constant called **learning rate** and t in the upper-script denotes the time and we call each of these updates as **step**. Remember that the loss above calculated over the entire dataset X. However, the size of a dataset hits easily millions in today's problems and calculating the exact gradient at each step poses technical difficulties. Therefore stochastic version of GD(SGD) is used to approximate the gradient direction over a subset of the dataset X instead of the complete dataset. We name these subsets as mini-batch and denote them as X_t emphasizing its dependence to step number. Number of elements in X_t is called as **batchsize** and denoted with m. At each step optimization algorithm chooses a random mini-batch X_t and performs the update with the partial derivatives calculated over this mini-batch. A complete-pass over the dataset is called an **epoch**.

1.2.2 Taylor Approximation of Loss

To understand why GD or SGD might work, let's write down the quadratic approximation of our loss function around the point w. Assuming the loss value is calculated each time with the complete dataset (X, Y) or with a mini-batch (X_t, Y_t) and the parameter set is represented by vector w of size M the approximation of $L(\dot{w})$ would be:

$$L(\dot{w}) = L(w) + (\dot{w} - w)^T \nabla L(w) + \frac{1}{2}(\dot{w} - w)^T H(w)(\dot{w} - w) + O((\dot{w} - w)^3) \quad (1.1)$$

where H(w) is an M by M matrix with elements $h_{i,j} = \frac{\delta L(w)}{\delta w_i \delta w_j}$.

Let's imagine we want to calculate how much the loss function will change if we change some weights by Δw such that new weights: $\dot{w} = w + \Delta w$. Substituting above gives us

$$L(\dot{w}) = L(w) + \Delta w^T \nabla L(w) + \frac{1}{2}\Delta w^T H(w)\Delta w + O(\Delta w^3) \quad (1.2)$$

and the change in the loss function $\Delta L = L(\dot{w}) - L(w)$is given by bringing the original loss to the left side.

$$\Delta L = \Delta w^T \nabla L(w) + \frac{1}{2}\Delta w^T H(w) \Delta w + O(\Delta w^3) \tag{1.3}$$

We can similarly write the change in the loss using the first-order approximation and we would get following.

$$\Delta L = \Delta w^T \nabla L(w) + O(\Delta w^2) \tag{1.4}$$

Note that the error term $O(\Delta w^2)$ increases polynomially with Δw and we need to ensure that this error term is small if we want to do anything with this approximation. Let's pick Δw as the gradient $\nabla L(w)$ itself. Then we would have a $\| \cdot \|^2$ norm of the gradient which is guaranteed to be positive. However we want to decrease the loss and ensure that Δw is small, so let's scale the direction with a small negative number α. Then we get what we need.

$$\Delta L \approx \alpha \|\nabla L(w)\|$$

$$\Delta L < 0$$

Having a step update rule on the negative gradient direction guarantees a decrease in the loss for a sufficiently small α. However when we use gradient calculated over a mini-batch note that our gradient directions are noisy and there is no such guarantee. However with some tricks and right set of tools SGD is undoubtedly the most successful tool for optimizing ANN's.

The goal of optimization is to minimize the loss over a dataset X and we called this dataset as **training dataset**. However, we are usually interested in minimizing the generalization error, which is usually approximated with a separate **test dataset**.

One can use various regularization methods to ensure a good generalization and it is usually believed/observed that ANN's trained with SGD generalizes well through some implicit(e.g. early stopping) and explicit(e.g. Dropout [17]) regularizer methods. However, all of these known sources are not enough to completely explain the generalization performance of neural networks [20] and why ANN's generalizes so well is one of the biggest mysteries of the ANN's.

1.2.3 Units(Neurons): Building block of DNN's

Parameter Name	**Tensor Shape**	W_i^l **or** b_i^l
Linear.weight	$I * O$	$[:, i]$
Linear.bias	O	-
Conv2d.weight	$I * O * H * W$	$[:, i, :, :]$
Conv2d.bias	O	-

Table 1.1: Tensor shapes of various layer parameters. I: input features, O: output features, W: width of the filter, H: height of the filter

As mentioned before ANN's consists of cascaded layers. Lets partition our parameter vector w into matrices W^l and b_l for each layer where l denotes the layer number and b is the *bias* vector. The shape of W^l depends on the type and size of the layer. By convention, the first two dimensions are used for #input features (I)and #output features (O) respectively. For convolutional layer, the third and fourth dimension would represent the filter width (W) and height (H) (Table 1.1). Each part in these layers that does a scaled summation of input features called as a **unit**(also neuron) and their parameters are denoted as W_i^l and

b_i^l. We denote this summation value with h_i^l

$$h_i^l = W_i \cdot a_i^{l-1} + b_i^l$$

The values generated at each unit is then passed to some non-linearities denoted with $g(\cdot)$. Value after the nonlinearity called as the **activation** a_i^l of a unit.

$$a_i^l = g(h_i^l)$$

Units are defined to be the independent components of a layer. However, they might get coupled as a result of some non-linearities which combine outputs from many units like SoftMax.

1.2.4 Saliency and Pruning

Saliency S_i of a parameter w_i is used to capture the contribution of individual parameters to the loss.

$$S_i = L(w_{-i}) - L(w)$$

where w_{-i} is the same vector as w except w_i is replaced with 0. A high saliency would mean that removing the parameter would introduce a high penalty on the loss functions, so it has a high contribution. On the other hand, a small saliency would mean that the contribution of the parameter is small(or even negative!) and can be safely removed. So basically, saliency captures the **penalty** for removing a parameter from the network. Removing a fraction of parameters (setting them to zero) according to the ranking of saliency scores is called **pruning**. Pruning is usually done after training as an inference optimization to make the network smaller. However, one can also prune the network during the training to improve on final performance or reduced the training time. Various saliency measures trying the approximate the quantity above are compared and explored in the next chapter(Chapter 2).

Pruning parameters according to the parameter-wise saliency above assumes independence between parameters. However, this independence assumption is not correct. Depending on the network architecture pruning a single parameter would change the saliency ranking of the remaining parameters and therefore make the initial ranking invalid. To prevent this effect, one can perform iterative pruning, where each time the parameter with the smallest saliency score(or smallest k of them) is removed and the saliency scores are calculated again before each pruning iteration. However, this would be very time consuming considering the size of the ANN's in today's problems.

Another problem with parameter-wise pruning is that the resulting sparse parameters would be still stored in a dense tensor and the computational speed-up would not be proportional to the pruning factor. One can decide to represent the pruned tensors in a sparse format; however, this would not provide speed-up unless pruning factor becomes considerably high.

An alternative to the parameter-wise pruning is to prune units. If we can calculate saliencies for units, i.e. penalty of removing the whole unit, we can remove units from the network. Removing a whole unit would put us in a much better position at doing iterative pruning (number of units is much smaller than the number of parameters) and replace the need for fast sparse operations. However, we can do even better.

In what conditions units end up having a low penalty for removal? This may happen when the unit constantly generates 0's or all the fan-out weights are so small that the network ignores the unit. It could be also possible that all fan-out units themselves have very high magnitude. What if the unit generates a value c with very low variance. If a unit constantly generates similar values then we can subtract the mean value it generates from its bias and we can propagate the same value to the fan-out units W_j^{l+1} by updating the bias of the units with the following rule.

$$b_j^l = b_j^l - \mathbb{E}[y_i^l] \tag{1.5}$$

$$b_j^{l+1} = b_j^{l+1} + g(\mathbb{E}[y_i^l] * W_j^{l+1}) \tag{1.6}$$

After doing this **bias-propagation** we can now have our saliency measures calculated again and we would get a very low penalty for removing the particular unit i, since it would generate always zeros. So, we can do better than just removing the units by calculating the **mean-replacement penalty** for each unit and applying bias propagation during removal. My thesis focuses on this saliency measure and we investigate our strategy in Chapter 2 and Chapter 3.

1.3 Literature Review

The magnitude of weights proposed as a simple measure of saliency in the earlier literature and its connection to the structured weight decay highlighted in [13]. [24] used second-order Taylor approximation at minima along with the diagonal approximation of the hessian to calculate saliencies for network parameters and emphasized the importance of pruning as a regularizer and performance-optimizer.

Later [15] used inverse of the hessian to get a better approximation of saliencies for network parameters. The experiments performed in these early works were relatively small tasks like the famous XOR problem.

Later it was the ice age for the neural networks and the majority of the research focused on the support vector machines(SVM). Various feature elimination methods are utilized to find out the minimal set of relevant features that give the same performance on the dataset. For a review on feature elimination and a reference paper, we would like to direct the reader to [18] and [10].

With increasing computational power, interest in neural networks increased again towards the second decade of 20th century. Many research attacked the difficulties we face training the neural networks and come-up with a bag of tools that helped us training bigger networks. With the increasing amount of data, tricks, and architecture, neural networks started to dominate the field again.

With the increased size of networks, many research highlighted the redundancy in the parameters. [5] showed that the whole network can be predicted from a small fraction of parameters. We can attack this redundancy in various ways and try to minimize the size of a network. The main motivation for doing so is to make deep neural networks available in hardware or time-limited platforms/scenarios. [36] trains very big gated network(Sparsely-gated Mixture-Of-Experts) consists of many small experts. It is more important now to learn how to make small networks as efficient as possible and we can always combine them.

One can reduce the size of a network by matrix decomposition [6], pruning [12] [11], quantized/low precision networks [25], [4], [32], [31] or distillation [8],[16].[6] reports 2x speedup on forward propagation, [12] reports around 30x compression through pruning, weight quantization and Huffman coding. [40] performs pruning during training and reports similar compression rates. [7] uses custom sparse-matrix multiplications after decomposition to speed up-convolutional layers. [3] compares various quantization methods and obtaining upto 50x quantization compression. [1] focuses on pruning bayesian neural networks. [9] shows that stochastic quantization helps preventing black box attacks on neural networks.

It seems like there is something that is going wrong during the training of neural networks, which high redundancy in our parameters and enables all the methods mentioned above to reduce the size of the networks.

[37] uses group lasso penalty to get some structured sparsity that enables reducing the size of dense tensors. [26] focuses on unit-wise pruning in the context of transfer learning and compared some first-order saliency measures using Spearman correlation coefficient. Their work focuses on pruning instead of the mean-score-replacement method, which we define in Section 3.1 formally. [39] proposes the idea of forcing the units to produce constant values in a neural network and then does *bias propagation* to remove them without a loss penalty which is basically the same idea we are proposing, since it is the generalized version of unit removal.

If a unit stops getting gradient (stuck in local minima) without doing anything useful. We should be able to detect this. This can help us to detect/find right architectures that align well with current optimization methods and provide us good training and validation losses. Therefore we believe that understanding behavior of units is crucial for answering other important question in neural network research. [14] shows that ANN's with more parameters than the number of training samples trained with SGD show good generalization behavior. [28] explores various complexity measures and proposes norm instead of count-based measures. [27] ablate units (which mimics removal) in various units to understand their generalization behaviors. They argue that a high mutual information is crucial for a good generalization.

We would like to point the reader to [2] for a review of current approaches and methods on network compression and acceleration.

Chapter 2

Weight-based pruning and saliency score comparison

We can measure the contribution of individual parameters to our loss value using the second order Taylor approximation (Equation 2.1). Following the notation in Section 1.2, lets denote the change in the loss function with the value ΔL and lets denote the saliency of the i^{th} parameters with S_i where w_i is the zero vector except $\Delta w_i = w_i$.

$$\Delta L = L(\dot{w}) - L(w) \approx \Delta w^T \nabla L(w) + \Delta w^T H(w) \Delta w \tag{2.1}$$

$$S_i \approx \Delta w_i (\nabla L(w)_i + \Delta w_i H(w)_{ii}) \tag{2.2}$$

This looks computationally plausible since during back-propagation, we calculate $\nabla L(w)$ and we know how to calculate an approximation of diagonal values of Hessian in O(N) time [21]. However, saliencies calculated with this approximation may change if anything in these networks changes (i.e. if we prune one parameter). To be able to use this measure in the context of pruning we need to assume pairwise independency between parameters, i.e. changing one parameter would not affect another parameter's saliency. In this section, we show this is far from being true. To be able to capture dependencies and improve the saliency score we propose updating the Hessian part by including the sum of the whole row of the hessian.

$$S_i \approx \Delta w_i (\nabla L(w)_i + \sum_j H(w)_{ij} \Delta w_j) \tag{2.3}$$

The hessian term in equation 2.3 can be calculated in O(n) time using Hessian-vector product [30] where n is the number of parameters in the network.

What should we set Δw_i to? In the context of pruning, we remove weights, i.e. setting them to 0. So we need to set $\Delta w_i = -\epsilon w_i$. When $\epsilon = 1$, this would correspond to weight removal. However, if w_i is not small the Taylor approximation may have a high error. Therefore we may introduce an ϵ term. Note that changing ϵ would not affect the ranking in first order approximation and therefore we don't include ϵ in our first order scoring function.

In the case of general sensitivity analysis, we would set $\Delta w_i = \epsilon * sign(w_i)$ to see how much the loss would change if we make each weight a little closer to zero by a constant amount.

In this chapter, we show that saliency calculations based on the Taylor approximation fail at pruning a group of parameters at once. Magnitude based saliency score works better than the rest. We start by comparing the performance of various first and second order saliency measures used at pruning (Section 2.1). Table 2.1

Saliency Function	$\boldsymbol{S_i}$	**Runtime**
taylor1Scorer	$\Delta w_i \nabla L(w)_i$	0.088s
taylor2Scorer	$\Delta w_i(\nabla L(w)_i + \lambda \sum_j H(w)_{ij} \Delta w_j)$	0.564s
hessianScorer	$\Delta w_i(\sum_j H(w)_{ij} \Delta w_j)$	0.483s
taylor1ScorerAbs	$abs(taylor1Scorer)$	–
taylor2ScorerAbs	$abs(taylor2Scorer)$	–
hessianScorerAbs	$abs(hessianScorerAbs)$	–
magnitudeScorer	$abs(w_i)$	0.00006s

Table 2.1: Summary of various saliency functions and their formula. In the context of pruning we would set $\Delta w_i = -w_i$ and $\lambda = 1$. In sensitivity analysis we would set $\Delta w = -sign(w)$ and $\lambda = \epsilon$ where ϵ is a small number ensuring correctness of taylor approximation. Runtimes are averaged over 10 runs on MacbookPro 2015 calculating scores for all parameters of the network proposed in Section 2.1 using a validation set of size 1000 from the MNIST dataset.

summarizes these saliency measures derived from the equation 2.3. All the scores in the table are implemented in the PYTORCHPRUNER package (see Appendix A). Note that our implementation of second order scoring functions takes 6-7 times more time than a single back-propagation pass on CPU. Run times can be further optimized, but it is important to understand that the second order approximations are in the same time complexity as the first order ones due to the hessian vector product enabled by the auto-grad library of Pytorch [29].

In Section 2.2 we demonstrate pruning a simple network trained on MNIST using magnitude-based scoring and show that pruned parameters have some kind of structure. They gather around some particular units and we leverage this observation and decrease the size of the network by half.

2.1 Saliency Measures and Comparison

In this section, we compare various saliency scores against each other. In theory, pruning individual weights can have a negative effect and we observe indeed this is the case. We then realize that the promised reduction in the loss can be overturned if we prune many weights together. This effect is observed more dominantly later in the training. This tells us that dependency among the individual scores increases over time.

In addition to the scoring functions in Table 2.1 we implemented two additional scoring functions. *lossChangeScorer* calculates the empirical saliency for each parameter by setting them zero one by one and calculating the change in the loss. This is the value *taylor1Scorer* approximates. *randomScorer* assigns random scores to each parameter sampled uniformly from the range $(0, 1)$, so that we can compare our saliency functions with random scoring.

Note that our primary goal is not to approximate *lossChangeScorer*, but to have a low overall penalty when we prune a fraction f of the parameters at once. To capture the dependency between parameters, we introduce *taylor2Scorer* and *hessianScorer*. Even though they don't approximate *lossChangeScorer*, we hope these saliency measures would pick the least dependent parameters and therefore induce a smaller penalty at pruning. Comparing the penalties of various saliency measures (i.e. *taylor1Scorer*, *taylor2Scorer* and *hessianScorer*) in the context of pruning, we observe that absolute values of these measures work better at pruning a fraction of the parameters at once. Finally, we show that *magnitudeScorer* works even better than the absolute-valued measures most of the time and has much better stability.

In each run, we are training our CNN model for MNIST dataset (Appendix C) with 0.01 learning rate and 32 mini-batch size for 5 epochs. At every 50 step, we are calculating the pruning loss without changing the model itself for each scoring function separately. All data dependent scores are calculated on a validation set of size 1000[1]. Finally calculated scores are used to determine which parameters to prune for a given target fraction.

We generated many experiments using various models(i.e. models with ReLU and Tanh), different layers and various scoring functions. It is not feasible to share all of the results here; therefore we share only a subset of the results where we believe represents the whole. All models below uses Relu non-linearity unless stated otherwise.

2.1.1 Negative Loss Changes and Correlation Between Parameters

Ideally, we would like to see a positive change in the loss when we prune a random parameter in a network. This would indicate that each parameter is doing something right and we can't give up any parameter. However, when we look at the results of the *lossChangeScorer*, we see that constantly around 35% of the parameters(of the first convolutional layer) cause a negative change in the loss function when removed (Figure 2.1). This is very interesting since the fraction of negatively scored parameters stay constantly around 35% throughout the training, which may hint us that the network can be pruned during the training, without waiting for the convergence and we may even get some improvements in the loss

[1] This might seem counter-intuitive since the validation set is not the data that we are directly optimizing for. However, our experiments and results don't focus on the comparison of various function and therefore the data we use doesn't make a difference and we confirmed that in separate experiments.

value. Even more striking, it seems like there are some parameters that are not getting the error signal they need to improve. There is something preventing them.

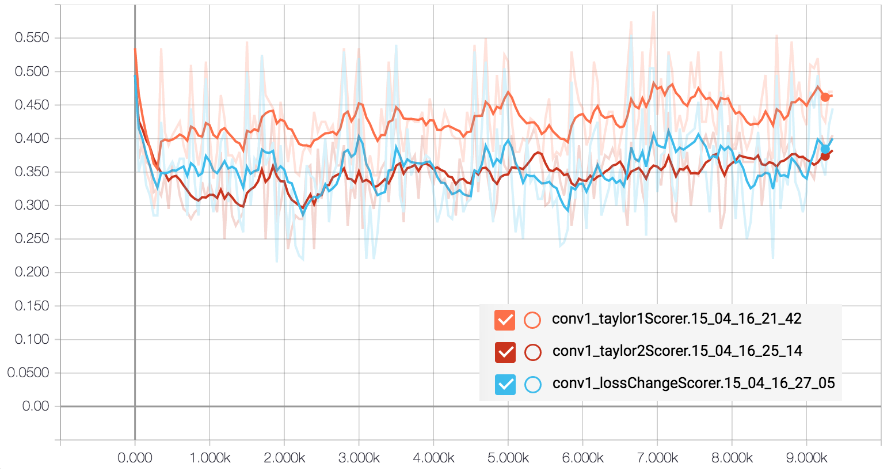


Figure 2.1: Horizontal axis represents the number of iterations passed. Vertical axis is the smoothed version of the fraction of negative saliencies for the first convolutional layer calculated with *lossChangeScorer*, *taylor1Scorer* and *taylor2Scorer*. C

There might be two reasons for the negative loss change. One is that we are overshooting the minimum constantly. Which is not likely since we see the fraction of negative scores even in the early training. More likely is that the path from the parameter to the calculated loss is blocked and the parameter is not getting any gradient signal. Remember that the gradient of a parameter is given by

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y_i} a_i^{l-1} \tag{2.4}$$

where a_i^{l-1} is the activation coming from the previous layer or the input tensor.

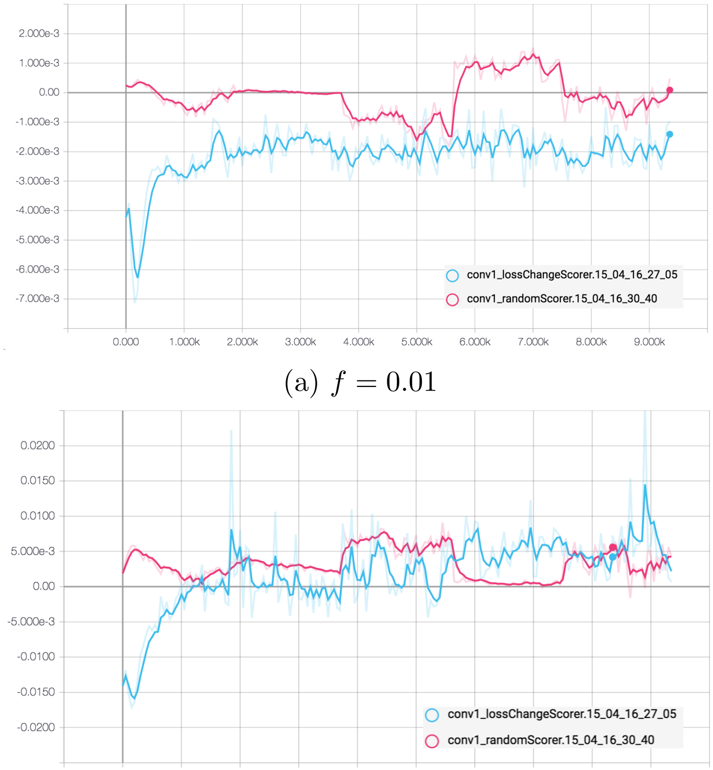
Looking at the back-propagation rule (Eq 2.4) there might be two reasons for very small gradient: input a_i^{l-1} being zero and the output gradient $\frac{\partial L}{\partial y_i}$ being zero. Assuming that the parameter gets some non-zero input across many different samples and mini batches, the reason should be that the unit at the other end has zero gradient itself(i.e. $\frac{\partial L}{\partial y_i} = 0$).

Having no gradient signal cannot be due to converging to a global minima since a good fraction of the parameters has a negative effect on loss once removed. One thing to note here that we average gradients over a mini-batch and it is possible gradients from different samples to cancel each other and get a very small average gradient. This scenario is possible, however considering we repeatedly calculate saliencies and see the same picture it is very unlikely. Therefore we argue the scenario where individual samples introduce zero or close-to-zero gradient.

This may happen again for two reasons. One is that the unit is doing a good job and have a small gradient. Another reason could be that it doesn't get any error signal since outgoing weights are all small and the unit is dead. So how to ensure the $\frac{\partial L}{\partial y_i}$'s being zero only when the whole network is optimal and we are having gradient flowing across all connections. One way is to have skip connections [38]. Another possible way is to detect low $\frac{\partial L}{\partial y_i}$ regions and check whether this small gradient is caused by the fact that they learned their tasks or not. In the case of small mean replacement error, we should fix the problem, since otherwise, we would have so-called dead units; which doesn't do anything useful and prevent learning for its weights.

2.1.2 Pruning can improve loss

Since we get negative empirical saliencies for a good fraction of weights; why not expect a negative change in the loss after pruning? To demonstrate this idea we prune the first convolutional layer using the *randomScorer* and *lossChangeScorer* for two fractions at Figure 2.2. When we prune only 2 parameters ($f = 0.01$) we



(a) $f = 0.01$

(b) $f = 0.05$

Figure 2.2: Small CNN [Appenix C] on MNIST with Tanh non-linearity. y-axis is the change in the loss if we prune fraction f of the conv1 parameters(200 parameter total) using *randomScorer* versus *lossChangeScorer*. At (b) we can also see that the dependency increases over time and we start getting positive change in the loss even though we would get negative change if we prune a single parameter.

observe a constant improvement in the loss value throughout the training. Yes, we can improve our loss value if we prune some small number of weights together. However, when we increase the pruned parameters to 10, we start getting into the positive regime. In some cases, the loss value increases even more than the case where we randomly prune 10 parameters. This observation shows that one can improve the loss value, even after 5 epochs if the number of pruned parameters are small. Note that one can always calculate the saliency again after pruning let say

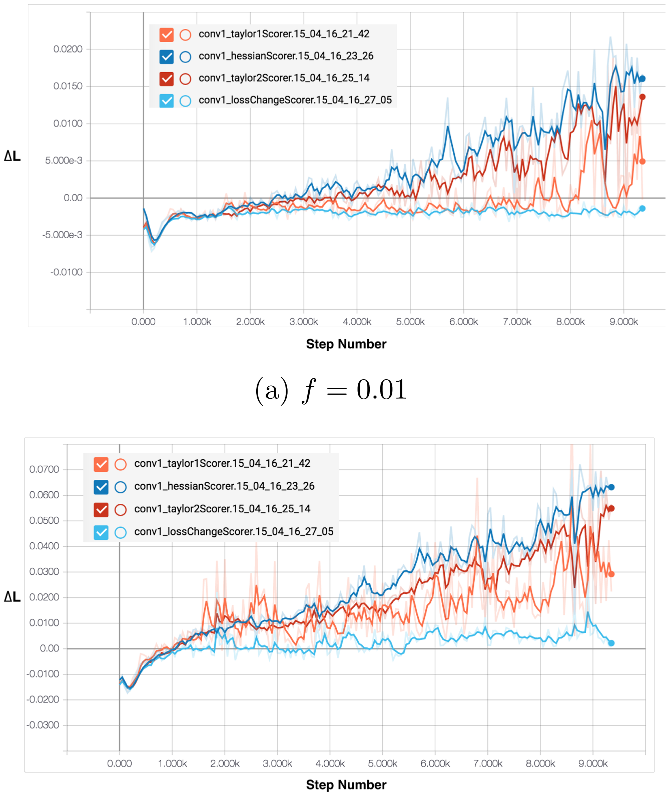
2 parameters. We haven't pursued this idea, however, it would be interesting to see how much one can improve loss function doing iterative pruning.

2.1.3 Dependency between parameters increases over time

Looking into Figure 2.2.b we see that pruning 10 parameters(out of 200) induces a negative change in the loss at the beginning of the training. However, after around 1000 steps, we observe that the dependency between scores increases and we start getting positive changes. This clearly shows that using individual saliency scores to prune a group of parameters is not a good strategy. We should somehow find a group of parameters that incur the smallest loss change when pruned together. This scoring function or approximation is yet to be discovered.

2.1.4 Approximating loss change doesn't really work

Even though in the previous section we showed that *lossChangeScorer* works poorly at pruning group of weights we still like to compare its performance with the performance of its approximations in the context of pruning. Note that one can numerically compare the approximations and measure the correlation between them. We leave this comparison as a future work. In Figure 2.3 we repeat the experiment in Figure 2.2 replacing the *randomScorer* with the three approximations of the saliency score S_i mentioned. We observe that these approximations work quite well early in the training, whereas towards the end of the training we see a relatively bad performance. We also observe that these approximations have occasional peaks which may significantly deteriorate the performance of pruning



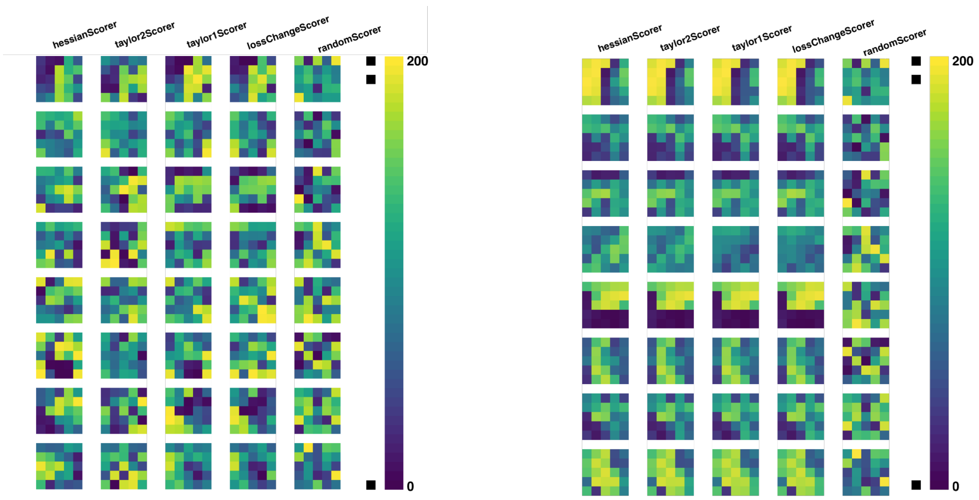
(a) $f = 0.01$

(b) $f = 0.05$

Figure 2.3: Smoothed loss change after pruning with various fractions using various scoring functions which approximates the *lossChangeScorer*. The close relation at the early training disappears towards the end. The three approximation also posses some peaks, which challenges the stability of these functions.

if used in practice. Remembering that all of the pruned parameters have negative scores for $f < 0.35$, higher changes in the loss value is a sign for increased dependency between the scores of the parameters. The answer to the question why parameters selected by saliency approximations have higher internal dependency is yet to be discovered.

One possible explanation is, even though pruning parameters with negative saliencies alone would decrease the loss, they may increase the saliencies of other



(a) after step 50

(b) after step 9350

Figure 2.4: Ranking of parameters among different scoring functions. Note that all scoring functions are trying to approximate *lossChangeScorer*. The first convolutional layer has 8 filters and yellow end of the color scale is used for high saliency ranking. We note that taylor1Scorer looks like best at approximating

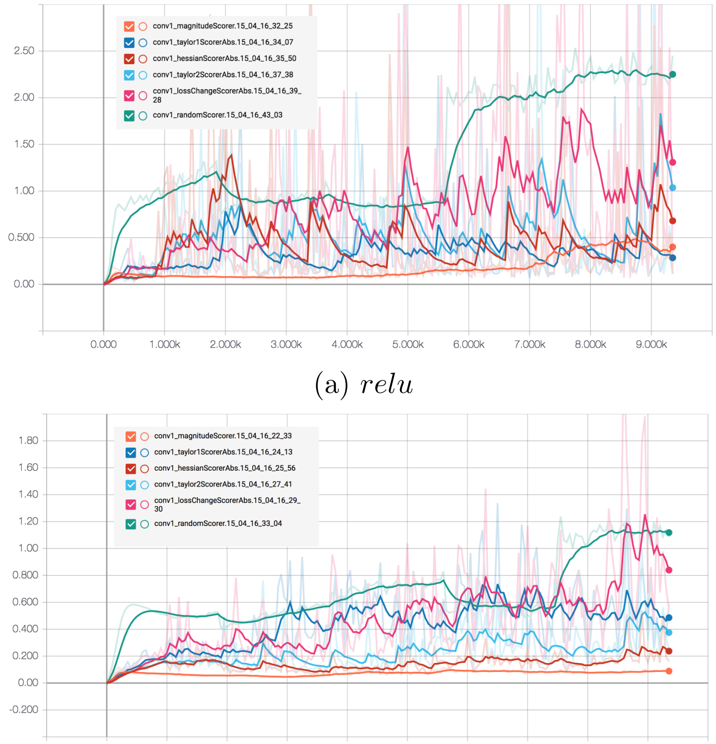
parameters more than the others. It might be the case that pruning negative saliency parameters make other parameters more important for the network. In other words, their absence might affect the saliencies of other parameters somehow more than the others. We may try to find parameters with small effect. In other words, parameters with close to 0 loss-change might bring better performance. We argue this idea in Section 2.1.5.

We believe that some parameters are just happened to have a low saliency in the beginning and they don't change since they don't get a proper gradient signal.

In Figure 2.4 we plot the saliencies of the first convolutional layer for the five scoring functions used in this section. We observe that *taylor1Scorer* looks like best at approximating the *lossChangeScorer* based rankings. *taylor2Scorer* is a combination of *hessianScorer* and *taylor1Scorer* and we are using $\lambda = 1$ in the formula proposed in Table 2.1. One may get better approximations playing with the coefficient λ.

Note that in general, pruning methods use a fraction of the parameters to decide which connections to prune. Therefore the ranking of a parameter is usually what only matters. However one can also use a threshold score to decide how much to prune. This strategy is not used in practice according to our knowledge.

2.1.5 Absolute valued scores works better



(a) *relu*

(b) *tanh*

Figure 2.5: Change in the loss value after pruning 160 of 200 parameters of the first convolutional layer over the training. The x-axis is the iteration number and y-axis represents $\Delta L(w)$. Curves are smoothed.

As observed in the previous section using saliency scores based on the loss-change when that individual parameter is pruned away, is a bad idea since we observe high dependency between selected parameters and get very bad pruning results. How can we find a scoring function which detects the low saliency and loosely dependent parameters? In our experiments, we found out that parameters with low absolute value scores have better results at pruning. We argue that this is due to the fact that parameters with a small effect on the loss(negative or positive) also have a weak dependency with other parameters.

Imagine a weight w^l_{ij} that determines the contribution of a^{l-1}_j at the summation happening in unit i, layer l. If the value generated at the unit u_i is not used or in other words the outgoing weights $w_{i\cdot}$ from the u_i are small we would have a very small change in the loss when we prune the w_{ij} away, in other words w_{ij} would have a small saliency. One can also expect to see parameter w_{ij} having a similar low magnitude saliency score if we prune some other weights. In other words, w_{ij} would have a very small effect on the saliency score of other weights and therefore one can prune w_{ij} and similar low magnitude saliency parameters together expecting to see a smaller change in the loss value. This is observed in Figure 2.5 for a pruning fraction of 0.8. Absolute values of S_i and its approximations work much better than the raw saliency scores in practice when used for pruning. Therefore we compare absolute valued versions of saliency functions in the next section with the magnitude based scoring.

2.1.6 Magnitude based scoring works best

We plotted rankings generated by the various scoring functions at the end of the training in Figure 2.6. We clearly see the resemblance between the *lossChangeScorerAbs* and its approximations *hessianScorerAbs*,*taylor1ScorerAbs* and

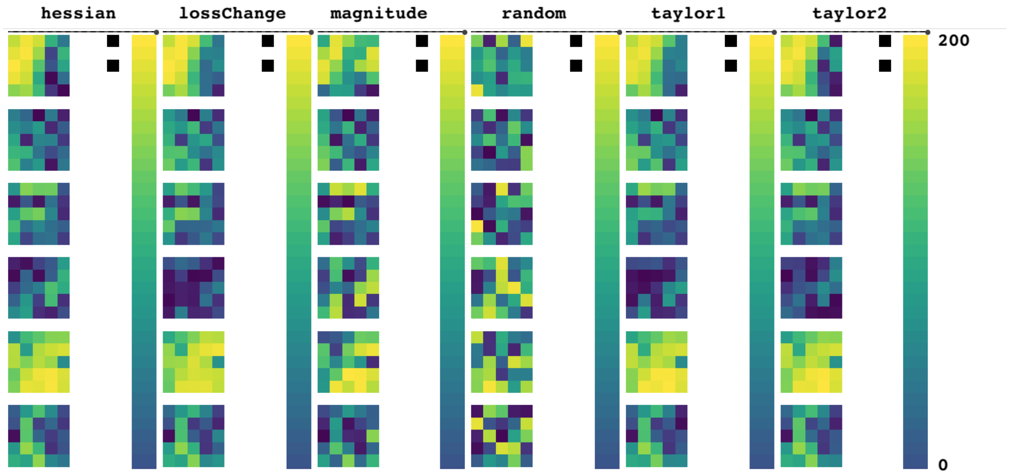


Figure 2.6: Rankings of the parameters of the first convolutional layer determined with the absolute value based scoring functions. At the end of the training we calculate saliencies of filters of first convolutional layer using various saliency functions and then rank them. Yellow means high saliency and blue means low.

taylor2ScorerAbs. They also have some similarities with the *magnitudeScorer*.

In Figure 2.5 we see that *magnitudeScorer* performs best among all other absolute value based scores. However one can see that other scores occasionally incur smaller loss change at the beginning and towards the end. The unexpected oscillations and peaks are problematic for most of the saliency scores and may be avoided using a running average or a bigger validation set. However *magnitudeScorer* seems to be quite stable throughout the training and looks like the simplest and best scoring function we found.

2.2 A Basic Pruning Experiment

Pruning ANN's is not limited to after training. One can prune a network also during training as they can do that after the network is trained. To demonstrate how pruning works during training we set up an experiment using a small CNN trained on MNIST dataset (Figure 2.7). The CNN has two convolutional layers with 8,16 filters respectively followed by max-pooling(stride=2) and Relu nonlinearity. Following are the two fully connected layers with 64 and 10 units respectively. Details of this network and all other networks can be found in Appendix C.

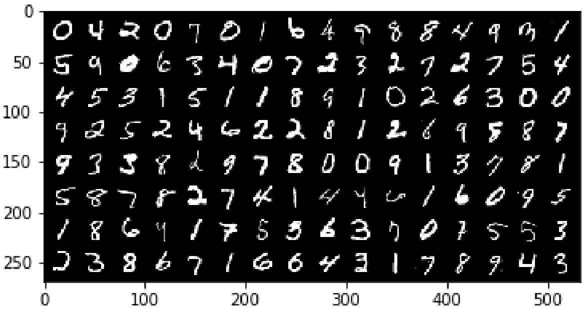


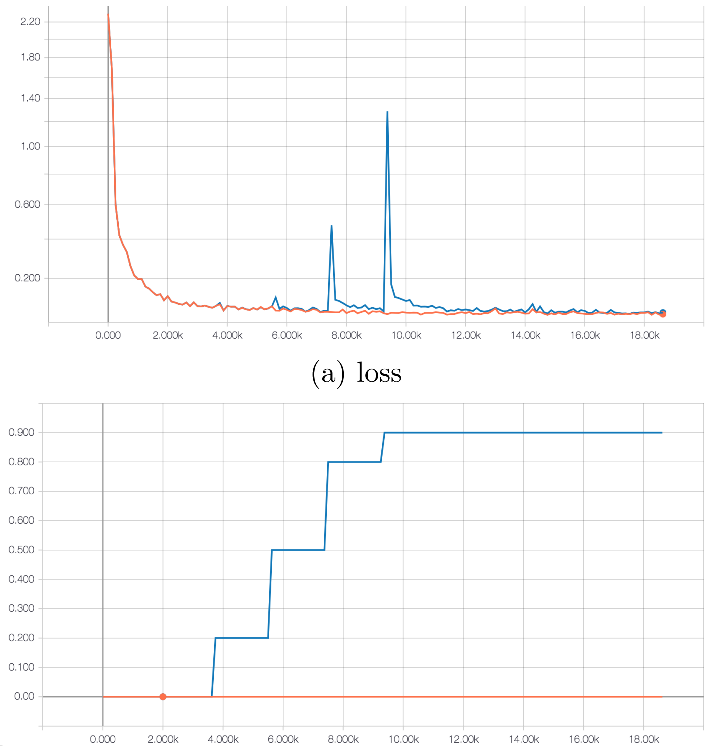
Figure 2.7: some samples from the MNIST dataset[22]

In this example we are using the simple *magnitudeScorer*(Table 2.1) to prune a fraction f of the network parameters each time and we are doing it with the following the schedule in Table 2.2. So after each scheduled epoch is done, we are setting a fraction f of all parameters with the smallest magnitude to zero. The gradient of these parameters is removed after pruning at every iteration to ensure that the weights stay zero and we are simulating weight removal correctly. After the run with pruning, we repeat the same experiment without pruning using the same initial random seed.

Epoch	2	3	4	5
Fraction f	0.2	0.5	0.8	0.9

Table 2.2: Pruning schedule for the MNIST experiment Section 2.1

So overall we perform 10 epochs of training and the loss value is plotted every 50 mini-batch/step(see Figure 2.8a). One can observe fractions smaller then 0.5 doesn't introduce much of a penalty in the loss. For fractions 0.8 and 0.9 we can see the jumps in the loss which is recovered during the rest of the training.



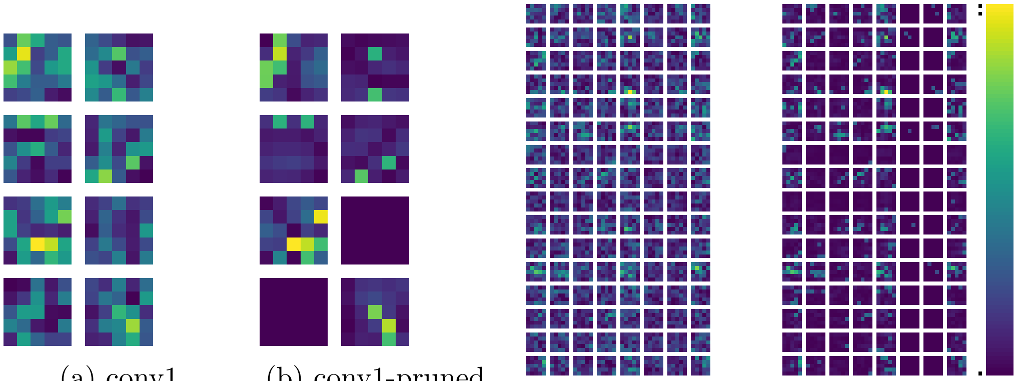
(a) loss

(b) fraction of paramaters pruned

Figure 2.8: Loss and fraction of pruned parameters over time. X axis represents time and parameterized by the step number. Blue curve is the run with scheduled pruning, where the orange curve is the original run without any pruning done. We are using the same seed for random number generator to ensure consistency.

After the training is done weights of the trained network are colored with a colormap; where yellow and dark blue used for high and low values. Before applying the colormap we normalize all the weights within a layer to the range [0,1]. Colormaps of the two convolutional layers are plotted in Figure 2.9.

We can see that the high magnitude weights in the original run are mostly

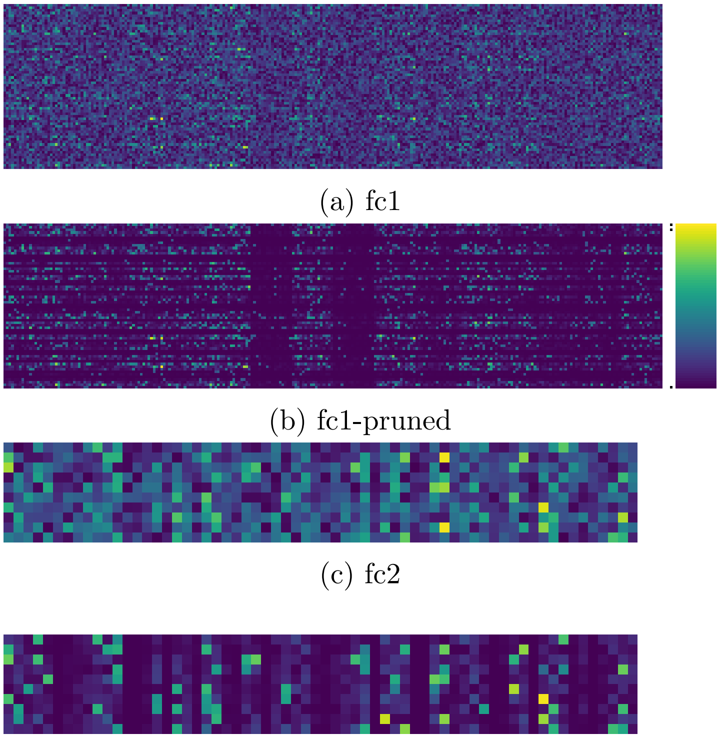


(a) conv1 (b) conv1-pruned (c) conv2 (d) conv2-pruned

Figure 2.9: Convolutional filter weight magnitudes with and without pruning. Yellow is for the maximum magnitude value encountered during training and blue is for the minimum value. *conv1* and *conv2* are the first and second convolutional layers. Each square for *conv1* is a unit. Each row for *conv2* is a unit and each unit has 8 kernels for the incoming 8 channels

preserved in the pruned network. This indicates that the regions contributing to the result most are formed within the first couple of epochs. We can also see in Figure 2.9b that 2 out of 8 filters are completely pruned; so that the corresponding filters in the *conv2*(column 6 and 7) are also mostly 0. Similarly, all filters in two units(rows 7 and 9) of *conv2* layer seems to be dead. One can also observe that usually 3-4 out of the 8 input channels for *conv2* units are almost all zero. This shows us the sparse relation between the units of the two layer.

Fully connected layer *fc1* takes the flattened output of the *conv2* layer. Similar to the colormaps of the convolutional layers, we can spot the high magnitude weights mostly appear in the pruned network, even though we start pruning after the second epoch. We can also detect two vertical stripes at *fc1* layer corresponding to the dead units(7,9) in the *conv2* layer. If investigated carefully one can also relate the horizontal almost all zero(dark-blue) rows in *fc1* layer to the vertical rows in *fc2*.



(a) fc1

(b) fc1-pruned

(c) fc2

(d) fc2-pruned

Figure 2.10: Fully connected weight magnitudes with and without pruning. Yellow is for the maximum magnitude value encountered during training and blue is for the minimum value. *fc1* and *fc2* are the first and second fully connected layers. Each row of the images represent individual units.

At this point, one can expect us to convert the network to the sparse format and enjoy the storage and computational gains. However, at the time of writing this thesis, acceleration libraries written for dense operations are much faster than their sparse counterparts and therefore converting a network with 90% sparsity to a sparse format would not improve the runtime if not harming it.

As the last part of this experiment, we count non-zero incoming and out-going weights for each unit. Then we remove the units with less than a fraction f_{nz} of non-zero incoming or outgoing weights (Figure 2.11). Fraction 0 would correspond to removing units with all zero incoming or outgoing weights. Note that a unit with all zero outgoing weights has no effect on the result and we can safely remove it. Similarly, a unit with all zero incoming weights would always generate the bias value and one can remove the unit after propagating the bias to the next layer. In this experiment, we haven't implemented the bias propagation and therefore we get slightly smaller test accuracy with $f_{nz} = 0$ and this can be easily avoided. We observe that we can make the dense network half size without any performance penalty. This shows that parameters pruned using the magnitude score have some kind of a structure; i.e. they are not completely random.

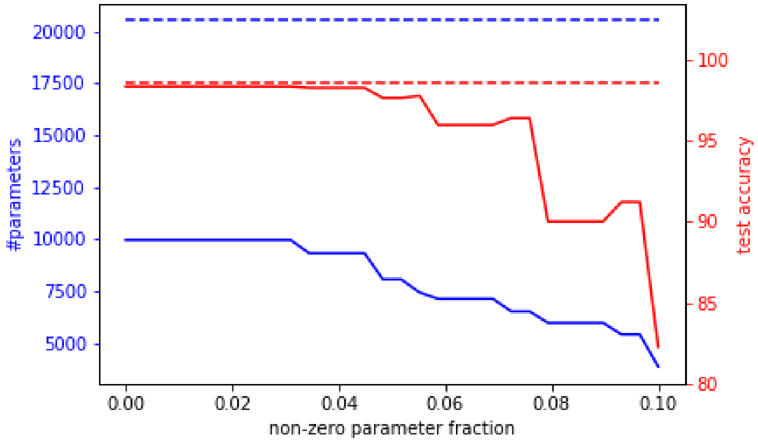


Figure 2.11: Left axis and blue curves represent the number of parameters after removing units for the given f_{nz} on the x-axis. The dashed blue line is the parameters of the original dense network. Right axis with red curves are the test accuracies of the networks after unit removal.

Chapter 3

Generating Dead Units Through High Learning Rate

Pruning individual parameters of an ANN works quite well if we look at the fraction of parameters that is pruned. However, in practice, 90% pruning does not correspond to 10x speedup or 10x compression. This is due to the inefficiency of sparse data structures and operations. However, when we prune the whole unit, we can remove the whole row or filter from the parameter tensor and reduce the size of the network without switching to sparse representations.

In this section, we switch our attention to individual units. Units or in other words neurons are the summing nodes in each layer, where the combination of previous activations happens. In other words, units are the meeting points where the information is generated and distributed. The forward information and the gradient flow through individual units and a flaw might affect the whole group of parameters connecting the unit. Are there units that are useless, preventing the network from learning new information? Is it possible that gradient-based learning algorithms create these dead units? Are there any tricks we use training deep networks help individual units to stay alive? In Section 3.1 we introduce *mean replacement penalty* and *mean replacement score* to quantize and approximate the importance of a unit in terms of their effect on the loss if removed. In Section 3.2 we introduce the *dead/frozen unit* concept and tie the definition of a dead unit with *mean replacement penalty.* Then we argue, that a high learning rate might one of the reasons causing dead units during optimization and define an attack in Section 3.3 to simulate high learning rates on individual units. Finally in Sections 3.4,3.5,3.6 and 3.7, we discuss results of our experiments.

3.1 Mean Replacement Penalty and its First Order Approximation

We defined the saliency for a single parameter in Chapter 1 as the change in the loss value, when the parameter is removed. Now in this section, we would like to do the same for individual units. [26] defines the saliency of a unit as the change in the loss when the entire unit is removed. This is equivalent of replacing the output of a unit with zeros. However, this approach fails for units that generate constant values.

So how should we define the saliency for units? Following the pruning idea, we can try to find units which produce zeros most of the time. If the output of a unit sums to zero for all samples, then we can safely remove it along with all of its outgoing weights. What if the unit produces a constant value instead of a zero? This is still okay, since we can propagate this constant value to the next layer(bias propagation Section 1.2) and remove the unit without introducing any penalty.

Even if, the unit does not generate a constant value, we can still remove the unit and propagate its mean to the next layer and expect to see a smaller change in the loss by doing so compared to the naive zero replacement. We call this removal process as **Mean Replacement** and define the **Mean Replacement Penalty (MRP)** as the following:

Definition 3.1.1. MRP: Mean Replacement Penalty is the change in our loss value due to replacing the value $[h_k]_i$ generated at unit i for the sample k with its mean value $E_{k \in D_{val}}[[h_k]_i]$ calculated over a validation set of D_{val}.

$$\text{MRP}_{k,i} = \sum_{k \in D_{val}} |L(\tilde{h}_{k,i}) - L(h_k)| \tag{3.1}$$

where

$$[\tilde{h}_{k,i}]_j = \begin{cases} E_{k \in D_{val}}[[h_k]_i] & j = i \\ [h_k]_j & j \neq i \end{cases}$$

Note that the definition of MRP involves the absolute value operator, which is different than the naive sum. This detail is important since it makes our penalty even more strict and captures the sensitivity against the mean replacement correctly.

Calculating MRP empirically would require us to run a forward-pass for each unit. Even though number of units are less then number of parameters, it is not feasible for deep networks (i.e. AlexNet has almost 300,000 units) to calculate the MRP for every single unit. Therefore we propose using the first-order Taylor approximation of the MRP. Like we did in Equation 1.4, we can write down the Taylor approximation of the loss value using the output of a given layer. Let's denote the output of some layer for the sample k with h_k vector.

$$L(\tilde{h}_k) = L(h_k) + \Delta h_k^T \nabla L(h_k) + O(\Delta h_k^2) \tag{3.2}$$

By setting $\Delta h_k = \tilde{h}_{k,i} - h_k$ we get the first order approximation for replacing the output at unit i with its mean. Δh_k would have zeros in all indices expect the i_{th} index. We can vectorize this calculation and calculate MRS for every unit of a given layer by setting $\Delta h_k = \tilde{h}_k - h_k$ where $\tilde{h}_k$ is the output vector whose elements are replaced by the corresponding mean values. Lets formally define this.

Definition 3.1.2. MRS: Mean Replacement Saliency for the units in a given layer calculated by

$$MRS_k = |L(\tilde{h}_k) - L(h_k)| = |(\tilde{h}_k - h_k)^T \nabla L(h_k)| \tag{3.3}$$

where

$$[\tilde{h}_k]_j = E_{k \in D_{val}}[[h_k]_j]$$

Our implementation of MRS calculates the mean values of all units and saves the zero-centered output vector(or matrix if it is a minibatch) during the forward-pass. Then, during the back-propagation, we multiply the output gradient of the layer with these centered output values to obtain the saliency vector. Note that since we are taking the absolute values at the end we don't need to negate the centered output values.

We would like to put a side note here. We calculate the mean of the output before the nonlinearity. However, we can also calculate the mean value after the nonlinearity and this would probably introduce a smaller loss penalty. We replace the values with the mean before the nonlinearity, since it was straightforward to implement and as the variance of the output goes to zero the two different approaches become equal to each other. Two different approaches become identical when MRP goes to zero. We also like to point out the possibility of using other operators than the mean, like median, min, max; etc. For example, using the mean output value for a unit that uses *TanH* might not be the best idea.

3.2 *Frozen* and *Dead* Units

In this section we are going to introduce two terms, hoping that they would help us to analyze and understand what is happening in neural networks over the training.

Let's imagine a unit in a fully connected layer. This unit generates a number doing a summation. It scales each of the previous activations, sums them and passes the result to a non-linearity like rectified unit(ReLU) or sigmoid function. The value after the nonlinearity is then named as the activation of the unit. These activations are then used by the units of the next layer in a similar manner.

During the back propagation, the unit gets error signals from the units of the next layer. The gradient of the unit is calculated by scaling and summing these signals as follows where h_i^l denotes the output of i^{th} unit in l^{th} layer.

$$\frac{\partial L}{\partial h_i^l} = \sum_j \frac{\partial L}{\partial h_j^{l+1}} w_{ij} g'(h_i^l) \tag{3.4}$$

We define frozen units as following.

Definition 3.2.1. Frozen Unit

Unit that receives negligible output gradient $\frac{\partial L}{\partial h_i^l}$ for all different samples of the distribution.

$$\frac{\partial L}{\partial h_i^l} \approx 0 \quad \forall x \in D_{val}$$

A unit might be *frozen* in various situations. Overall loss can be zero and all network does not get any gradient signal. This would correspond to the first term in Equation 3.4 being 0 $\forall$ j. However, this case is not really interesting since a zero loss would mean we are at the minima and there is not really much we can do. The interesting case is when the unit does not get any gradient signal even when there is some gradient signal available in next layers. Thinking the chain rule this might happen due to weak connections (small weights) that the unit has with the next layer (second term in Eq. 3.4). And another possibility is when the third term is zero, which would correspond to the case when the entire h^l lies in the flat region of the activation function. This would make the third term $g'(h_i^l)$ zero.

It is also possible that the scaled gradients coming from the next layer cancel each other and the unit gets zero gradient. However, this is also unlikely to happen thinking there are many mini-batches and different combination of samples.

Note that all the layers except the first layer, get their input from other layers. Since we are constantly updating parameters during the training, input distributions of layers may change over time. However, once the input distributions stabilize we know that the frozen units are going to stay frozen. After early training, we expect the input distribution of a frozen unit not to change much. Therefore we also expect a frozen unit to stay frozen and we observed such a behavior in our experiments.

When a unit is frozen, we expect it to stay frozen.

Lets define the concept of dead units. Dead units are the units that doesn't effect the loss value when replaced by a constant. We can employ various strategies at picking the constant and in our definition we employ the definition at Section 3.1 .

Definition 3.2.2. Dead Unit

Units that are replaceable with a constant.

$$MRP_{k,i} \approx 0 \quad \forall k \in D_{val}$$

Let's identify the three situations where a unit introduces a low MRP.

1. Unit generates a constant value

2. All values that the unit generates mapped into a constant value by the activation function.

3. All outgoing weights are zero.

The first situation might happen if we have all near-zero weights or input-weight combinations that cancel each other in the summation. This situation is not very stable. The unit might get gradient signal and get out of this situation. Therefore we don't expect to see this state often during training.

The second situation is particularly interesting to consider since values at the flat region of activation functions would also mask the incoming gradient and therefore the unit would always have zero or near-zero $\frac{\partial L}{\partial h_i}$ for every sample. If a unit gets into this state, it also gets *frozen* since all the gradient is masked by the activation function (the third term in Equation 3.4 being 0). So we should definitely detect these units and either remove or revive since we know they are frozen and they would most likely stay frozen.

The third situation might happen when the network ignores the values unit generates. Assuming the unit generates non-constant values, this situation is stable only when the units in the next layer are frozen. Otherwise, the weights would get some gradient and become non-zero.

Units that don't generate constant values might end up having very low MRP, if the values created has a low variance or/and the outgoing weights are nearly zero.

In the remaining of this text we will use these definitions to explain some concepts and observations efficiently. We omit setting hard thresholds for these two definitions since thresholds depend on the number of units in a layer or the loss function used.

3.3 How can a unit die?: High Learning Rate

There could be various reasons why we may end up having dead units. In this chapter, we would like to investigate one possible reason. We suspect that having a big step size during the training can put the output distribution of the unit into the flat region of the subsequent non-linearity. Most of the practical/popular optimization algorithms use the same learning rate for all the parameters in the network. During the training, this single learning rate may end up being too much for some units and cause them to die. We would like to artificially create this situation by attacking some of our units.

An attack at iteration i on unit j is done by multiplying the default learning rate for the weight parameters W_i^l of unit i without changing the learning rate for the rest of the network and the bias of the unit.

Each update on network parameters using the gradient calculated from the given mini-batch is called an *iteration*. So we would pick a starting iteration number S and perform our first *attack*. Then we would repeat this attack N times with I many iterations in between. So we basically don't modify the learning rate for $I-1$ many iterations after each attack.

In the upcoming sections, we are going to investigate various non-linearities and how they respond to this learning rate attack. For all experiments below we train a small CNN (Appendix C) with CIFAR10 [19] dataset. We use SGD without momentum and a constant learning rate of 0.01. We calculate all data-dependent measures (like the loss and MRS) on a separate validation set of size 1000. We train the network in each experiment for 10 epochs and 30 epochs for experiments using *TanH*. We use PyTorch's default parameter initialization and initialize the network with a uniform distribution from a range $(-\sigma_l^2, \sigma_l^2)$ where the σ_l is the variance calculated for each layer separately and equals to $\sigma_l^2 = \frac{1}{n_{unit}}$. n_{unit} is equal to the number of incoming connections for each unit.

3.4 Killing a Unit with RELU+Batch Norm

Let's start this section by looking what happens during the training of a small CNN with RELU non-linearities and batch normalization. All experiments in this section run for 10 epochs and the networks reach to an accuracy of 67%.

In Figure 3.1 we see various plots for the attacks on 3 different units of the first convolutional layer. Each row represents a different unit. The two orange and the blue curves in the 3 rows respectively, represent the original training without any learning rate change. In common, we see that the norms of the weights increase.

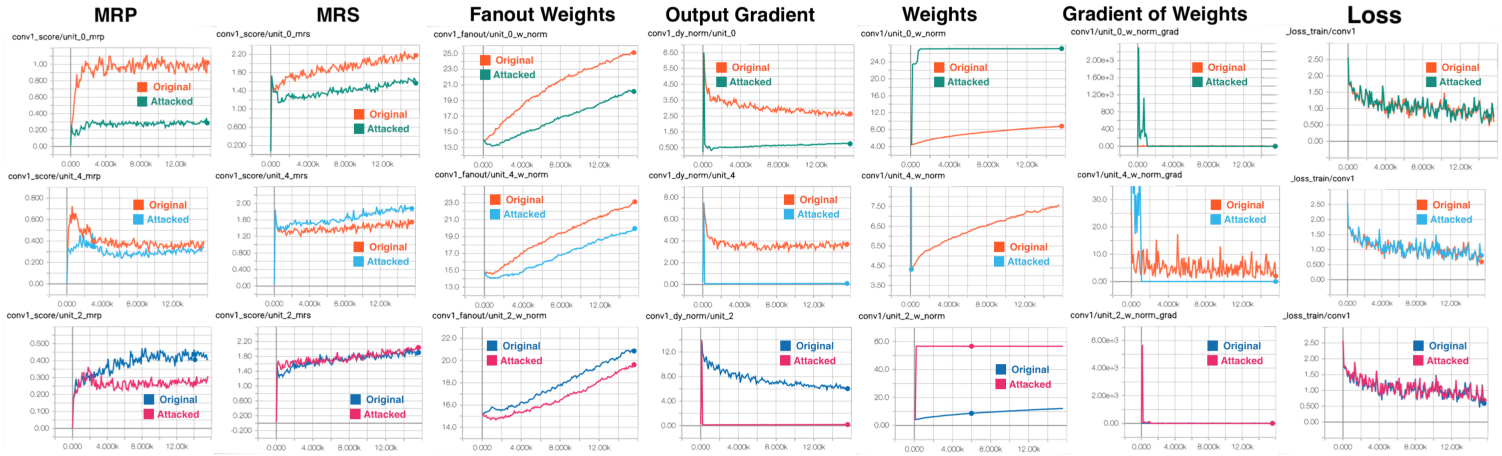


Figure 3.1: What y-axis represents in each figure is written on top of each column. Each row represents different experiments with different units attacked. For vector quantities (row 3,4,5,6), it is the l1 norm of the vector that we are plotting. The x-axis represents the iteration number in every plot. We attack each unit separately starting from 50th step 10 times with 10 step intervals using a learning rate multiplier of 100. For each unit, we plot the *original* behavior as indicated in each legend.

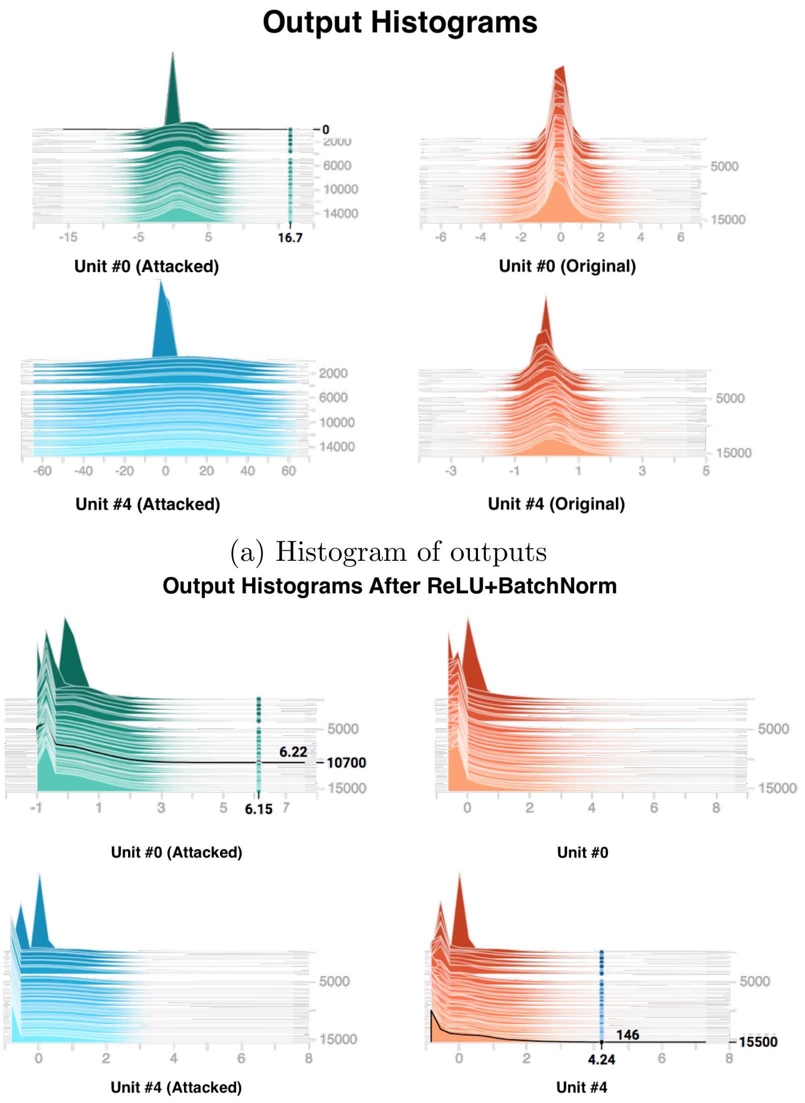
We also see that the MRP generally increases over time. It looks like after around 2000th step, units reach their final MRP values and they oscillate around these values afterwards.

Three units in Figure 3.1 have different responses to the attack that starts on 50th step and repeats 10 times with 10 step intervals. During each attack, we multiply the gradient for the attacked unit with 100. This cause unit to follow a different optimization path. The first thing we notice is various units responded to the attack differently. Let's start with common responses we see. As expected we see a jump in the norm of the weights(column 5). MRP of Unit-0 (first row in Figure 3.1) reduces significantly without affecting training loss (last column). We observe that none of the attacks change the loss curves. This is very interesting, since if these high-learning-rate-updates happen during the training causing units to die, they pass undetected.

Increased norm of the weights cause output values to increase and we get bigger values out of the linear part of the ReLU. This will cause the mean and the variation calculated by the batch normalization to increase. As a consequence, zero activations and nearby values would squeeze further to the left edge. In a sense, we might squeeze what unit learned so far towards left due to the increased values on the linear side(right).

The transformation of the output distributions for a successful and an unsuccessful attack are plotted on Figure 3.2. We see that the same multiplier has two different effects on the two different units and this is very much expected considering the differences in gradients and parameters of two different units.

An abruptly shrunk left side of the output distribution is also going to alter



(a) Histogram of outputs

(b) Histograms after ReLU+BatchNorm

Figure 3.2: Output histograms are created from a validation set of size 1000. First row of histograms(unit 0) is an example for a unit dying due to the attack. Whereas the second row(unit 4) doesn't die after the attack and we can see that the output distribution of the unit 4 is not changed after attack and looks very similar to the original distribution.

the gradients of the outgoing weights since their gradient is given by

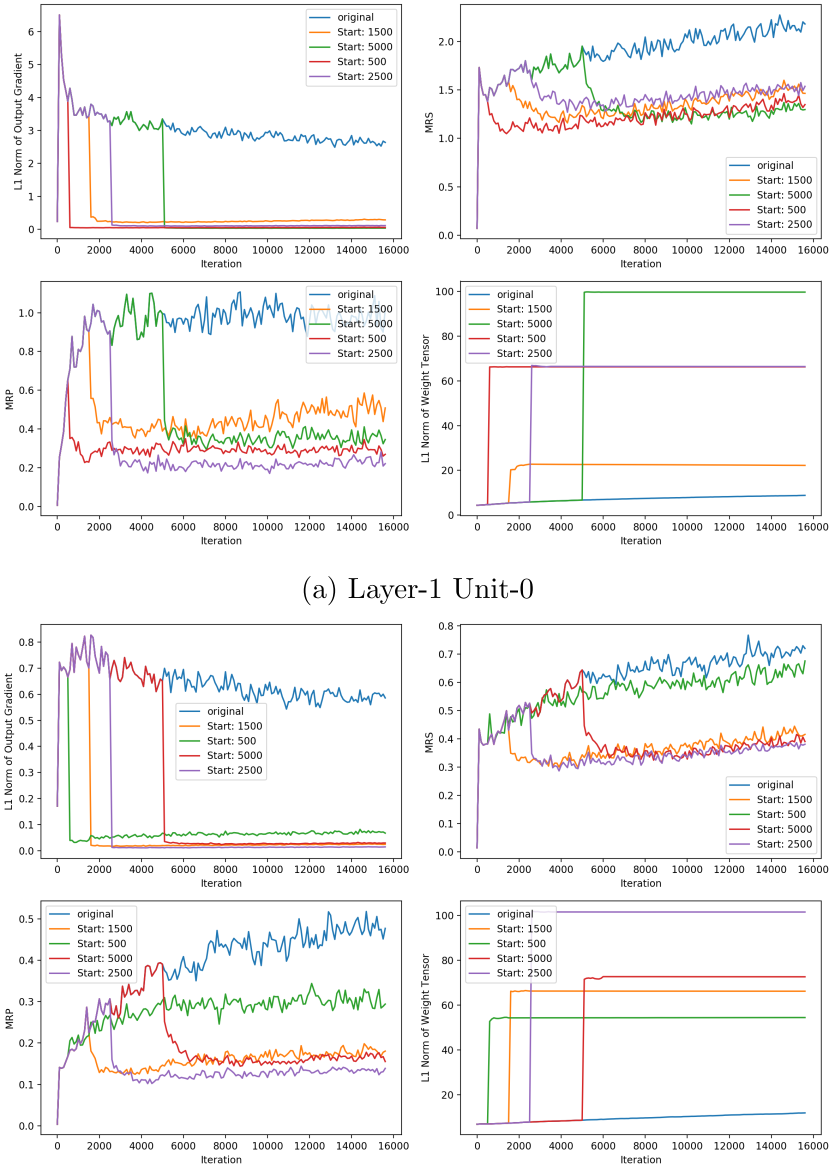
$$\frac{\partial L}{\partial w_{ij}^{l+1}} = \frac{\partial L}{\partial h_j^{l+1}} a_i^l \tag{3.5}$$

where w_{ij} is the weight connecting our unit(i) to the unit(j) in the next layer. a_i^l is the output of our unit and as reasoned above their distribution is expected to shift towards the negative side causing the weight w_{ij} to receive a smaller gradient signals. Therefore we expect the outgoing weights in the attacked version to be smaller. We can see this effect in the third column of the Figure 3.1.

Note that after a successful attack(like unit-0), batch normalization would have a high variance σ_i^2 dividing each value and it would shrink the incoming gradient with the same value σ_i^2. This makes $\frac{\partial L}{\partial h_i}$ to decrease (column 4 of the Figure 3.1). As a result $\frac{\partial L}{\partial W_i}$ decreases, too (column 6 of the Figure 3.1). This clearly shows us that units might not be dead even when they are frozen. Units may have a small gradient signal coming to them, however they might still create some useful information used by the rest of the network and therefore having a high mean replacement penalty.

3.4.1 Effect of the attack is time dependent

In the previous section, we showed the responses of unit 0,4 and 2 of the first convolutional layer against an attack with an early start (attacks starting at 50th step). In this section, we like to continue our experiments by changing the time we start attacking the unit. We are attacking the unit 0 in the first convolutional layer 10 times with 10 iteration intervals starting from 500th, 1500th, 2500th and 5000th steps in different runs and results are plotted in Figure 3.3a. We clearly see



(a) Layer-1 Unit-0

(b) Layer-2 Unit-0

Figure 3.3: Various metrics observed during 4 different attacks each starting at a different time-step. The x-axis represents the time (iteration) and all vector values are plotted after taking their l1-norm. We can clearly see that there is no clear relation between time and the success of a learning rate attack. Corresponding loss curves plotted in Figure D.1 and D.2

that the most successful attack is the one starting on the 2500th iteration (step) and the second most successful attack is the one start at 500th iteration, whereas the attack starting on 5000 and 1500 gets the last two spot. We observe that MRS does not capture the ordering perfectly even though it is successful at separating the original unit from attacked ones.

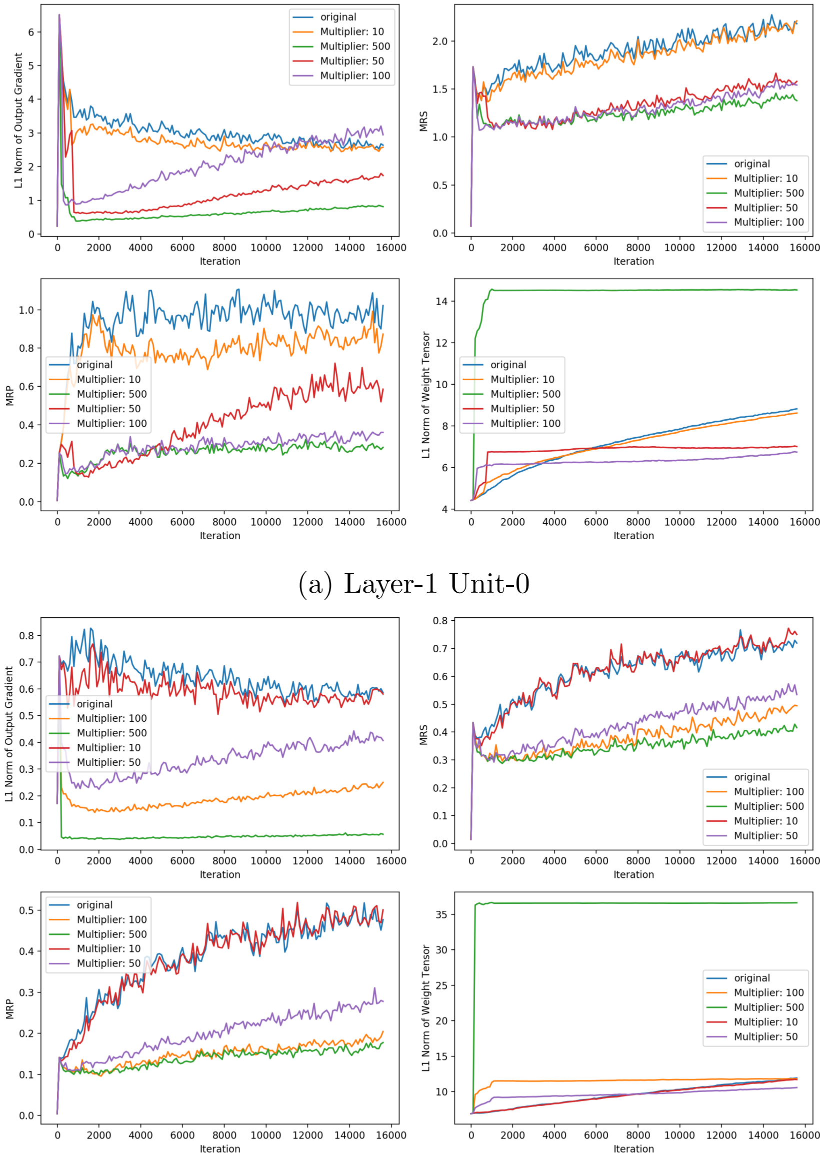
On Figure 3.3b, we have a very similar picture, where the success of an attack seems to be not correlated to the time we start the attack. However this time we see that the MRS makes a very good job separating low MRP units from the rest.

Another interesting observation that we can make in Figure 3.3 is the negative correlation between the weights and gradient signals coming to the unit(output gradient). It looks like output gradient of a unit decreases proportional to the increase in the norms of the corresponding weights and this aligns well with our explanation in the previous section.

3.4.2 Increasing learning rate multiplier causes units to die

In Figure 3.4 we see two groups of plots corresponding to the experiments where we attack the first units of the first and second layers respectively. Looking to the Figure 3.4a, one can clearly see that the success of an attack (drop in MRP) increases with the learning rate multiplier. We don't see any clear relation between the MRP and magnitude of the weights. However, an interesting observation is that the norm of the output gradient is not necessarily small for low MRP units. Looking to the blue curve we see that the norm of the output gradient exceeds the original run towards the end of the training despite having a much lower MRP.

In Figure 3.4a, the first unit of the second convolutional layer responds to the increasing multiplier in a similar manner, i.e. MRP's drops more, weights jump



(a) Layer-1 Unit-0

(b) Layer-2 Unit-0

Figure 3.4: Various metrics observed during 4 different attacks with different learning rate multipliers, each starting at 50th step and repeats 10 time with 100 step intervals with . X-axis represents the time(steps taken) and all vector values are plotted after taking their l1-norm. The success of the attack increases with the multiplier. Corresponding loss curves can be seen in Figure D.3 and D.4

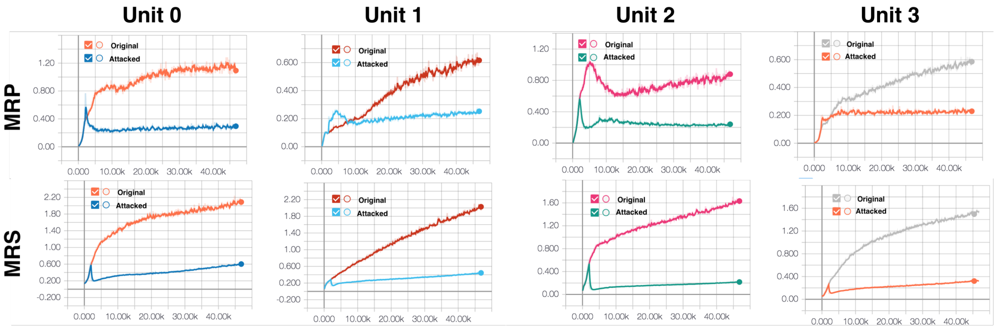


Figure 3.5: MRP and MRS of various units plotted against time X-axes represents the number of steps (iteration) taken. We see all attacks are somewhat successful at limiting the MRP of a unit.

more and MRS do a very good job at approximation the MRP. This time we also see a very correlated picture between output gradients and weights.

Results of the experiments in this section clearly shows us that that it is the attack(high learning rate) that makes units die.

3.5 Killing a Unit with Tanh

In this section, we switch to the *TanH* nonlinearity and perform 10 attacks starting from the 2000th step with 10 step intervals between each attack and using a learning multiplier of 1000. Since it takes more time to train network with *TanH*, we let the experiment run for 30 epochs instead of 10. The network reaches to an accuracy of 64%.

Figure 3.5 shows MRP and MRS of four different units from the first convolutional layer with and without the attack performed. The experiments without an attack labeled as *Original*. Two of the units have final MRP's around twice more than the other two, however, they all drop to an MRP of 0.2-0.3 after the attack

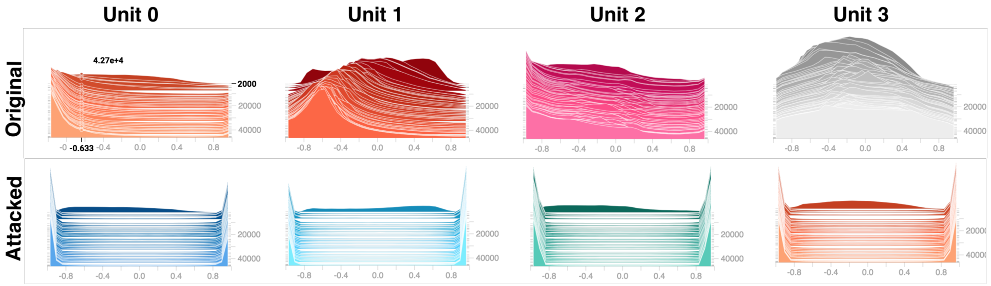


Figure 3.6: First row shows the normalized histograms of activations of each unit on the validation set of size 1000 without an attack. Second row is the same histogram when the unit is attacked 10 times between (2000,2100)th steps. The z-axes of the histograms represent the iteration number.

and MRS does a very good job approximating the MRP.

As happened before the output distribution of the attacked units scaled considerably due to the relatively large updates caused by the attack. This shifts the activation distribution to the edges. In other words, most of the input distribution is mapped into the flat region of *TanH*. In a very basic sense, the amount of information available at the end of the layer decreases considerably. We can see this behavior in Figure 3.6 where we plot the activation distribution's of the attacked units using the same units as in Figure 3.5 and same colors. The output of first three units (0,1,2) in Figure 3.6 shifts toward the left towards the end of the training. This gives us a clue, that if we perform the attack at a different time we might end up in an extremely uneven activation distribution and therefore have even lower MRP's.

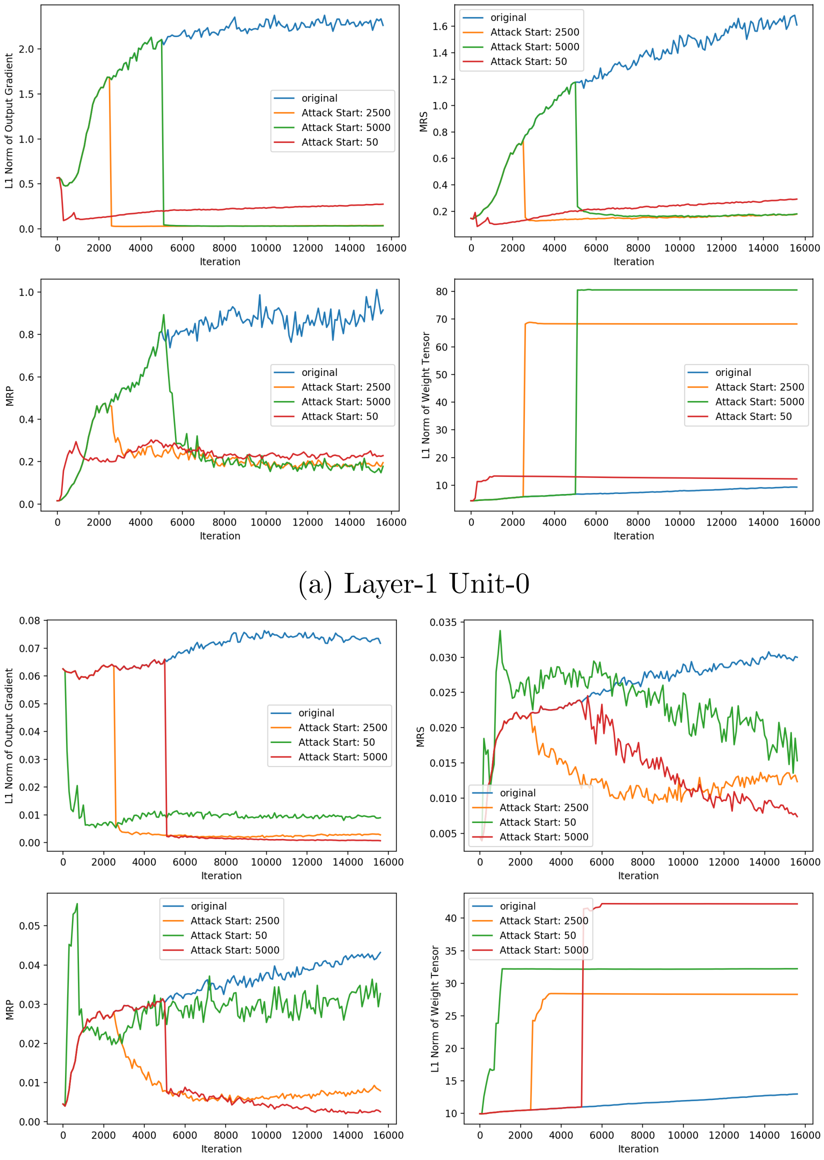
3.5.1 All you need is the right timing

In this section, we attack the unit-0 of the first layer and unit-16 of the third layer (first fully connected layer) 10 times with different starting times. Results taken from these two experiments are plotted in Figure 3.7. Once again we observe correlation between *Output Gradient* and *Weights* after the attack in both experiments.

In Figure 3.7a we see that all three attacks causes similar drops on the MRP score. However, we see significant differences on the norms of weights and output gradients. One subtle observations is that the attack starting at 50th step increases the MRP at the beginning, however the increase stops after a while (red curve). However, we observe that MRS captures the ranking perfectly throughout the training.

In Figure 3.7b attacks starting at 50th step increase the MRP considerably, however, these effects are fadeout later during the training. We suspect this behavior being due to the small-magnitude-gradients. An early attack may not be enough to saturate the activations. To understand this phenomenon further we plot the histogram of the unit outputs and activations of unit-16 in Figure 3.8.

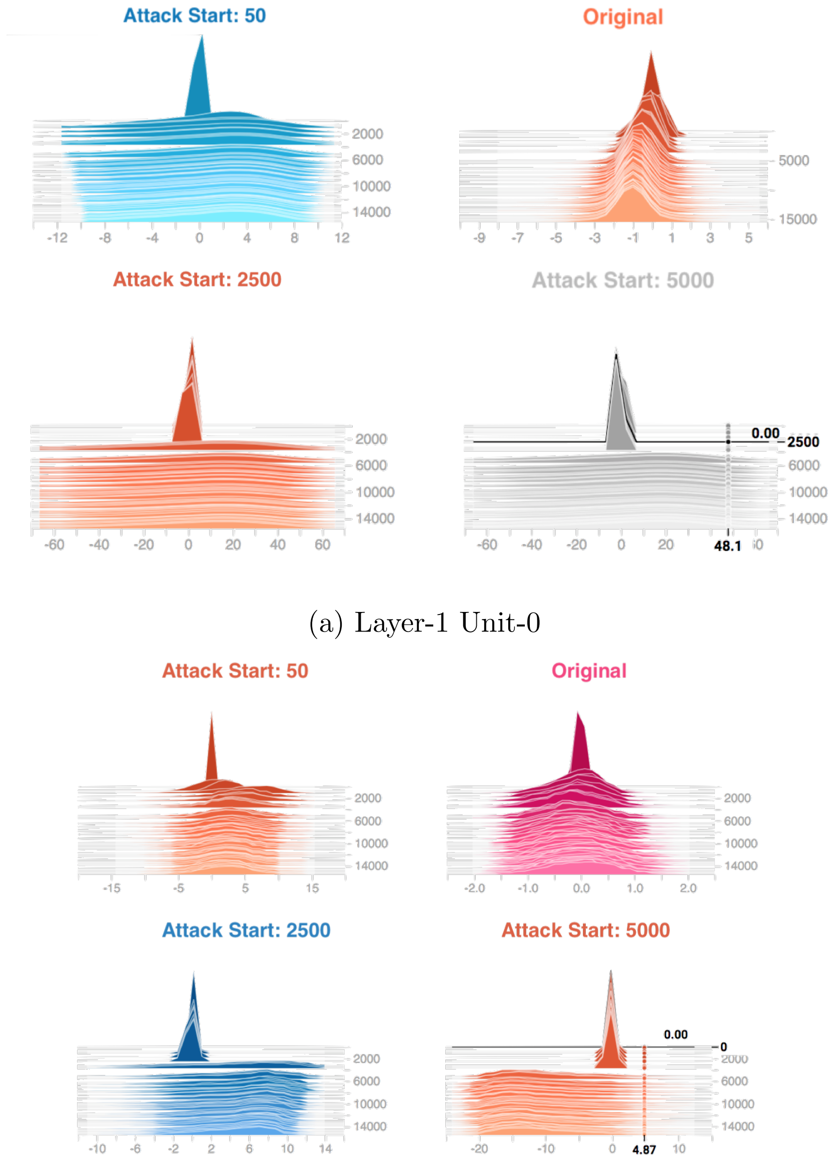
The attacks started at 50th step in both experiments doesn't scale the output distribution considerably and therefore the MRP of the unit increases. However attacks performed later (at 2500 and 5000) cause a wider distribution and cause more outputs to saturate. Note that the attacks starting at 2500th and 5000th step causes activation distributions leaned towards the right and left respectively. This shows that the effect of the learning rate attack depends heavily on the step it is performed.



(a) Layer-1 Unit-0

(b) Layer-3 Unit-16

Figure 3.7: Various metrics observed during 3 attacks with different starting points. Each repeats 10 times with 10 step intervals. X-axis represents the time(steps taken) and all vector values(Output Gradient and Weights) are plotted with l1-norm. Corresponding loss curves plotted in Figure D.5 and D.6



(a) Layer-1 Unit-0

(b) Layer-3 Unit-16

Figure 3.8: Output histograms of two units. x-y-z axes of each histogram represents value, normalized count and step-taken (itereation) respectively. Attacks started later during the training seems like generating more values on the flat regions(left-right edges(-1,1) of the histogram).

3.6 Killing a Unit with Relu

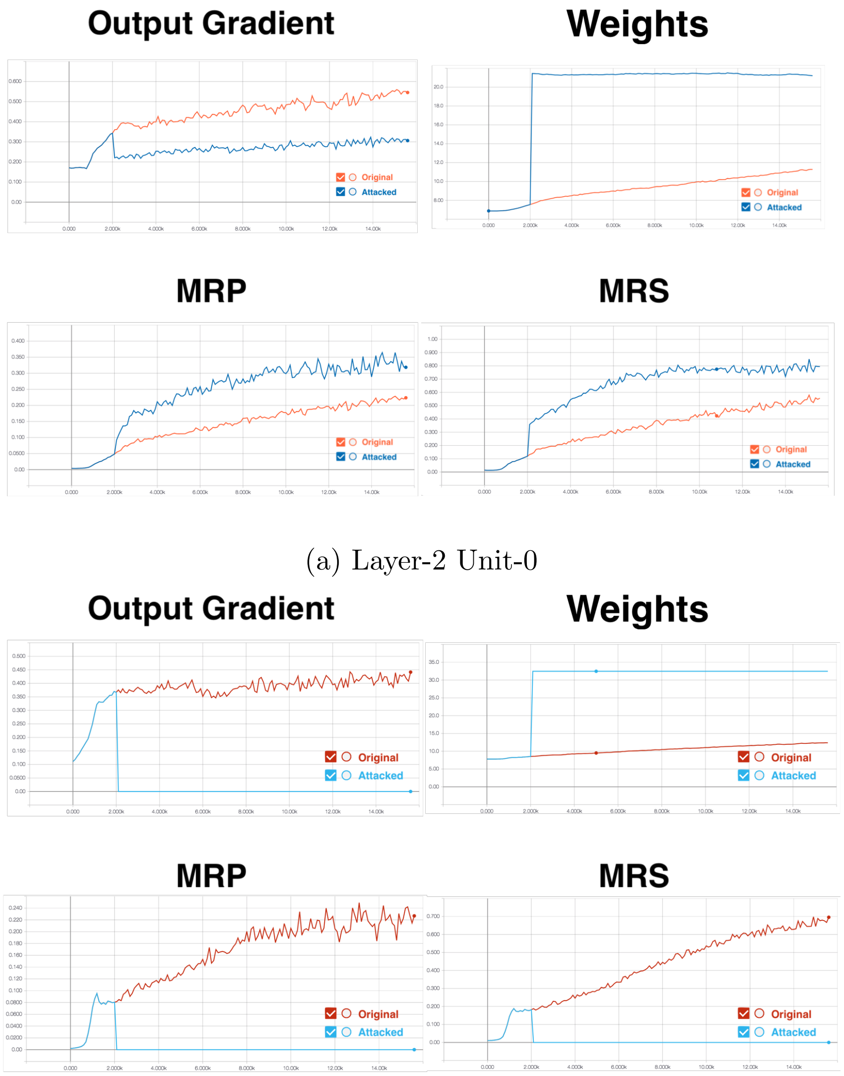
In the previous two sections, we showed that the learning rate attack disturbs the activation distribution and often causes a drop in MRP. We also reasoned that this effect seems like due to the reduced information produced by the unit. In this section, we would like to remove the batch normalization and repeat our experiment on a unit with ReLU activation. All experiments in this section run for 10 epochs and the network reaches to an accuracy of 61% independent of whether an attack performed or not. We perform learning rate attacks starting at 2000th step of the training again 10 times with 10 iteration intervals.

In a network with ReLU nonlinearities, it is harder to lose information since one side of the activation function is not bounded. If our output distribution is centered around 0, a learning rate attack would not saturate the non-zero activations; it would only scale them such that the information is preserved. Moreover increased magnitudes may generate bigger gradients on the outgoing weights and make the information provided by the unit more important for the task. We may even see faster convergence (for real).

3.6.1 Whatever doesn't kills you make you stronger

Due to its unbounded nature, we had hard time attacking ReLU. Often, we observed that attacking a ReLU unit cause an increase in MRP. An example of this effect is given in Figure 3.9a. The attack on Unit-0 of Layer-2 reduces the output gradient, however, increases the MRP. Most of the units we attacked responded the attack in this manner.

However, we observed some units(2 out of 4 at second layer) responded to



(a) Layer-2 Unit-0

(b) Layer-2 Unit-4

Figure 3.9: Various metrics observed during an attack on 2 different units. Both attacks start on 2000th step. The attack on unit 0 makes the unit more important for the network (increased MRP), whereas unit 4 is completely dead after attack with 0 MRP.

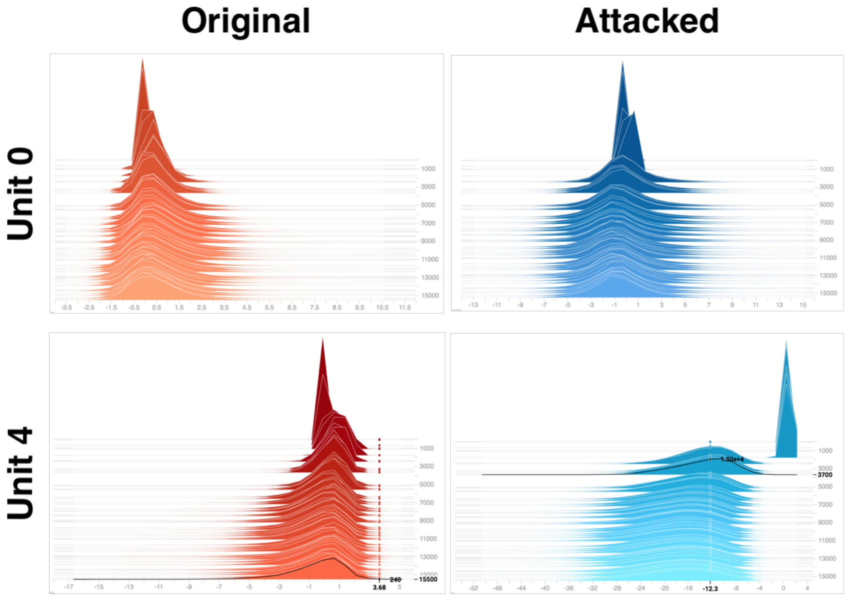


Figure 3.10: x-y-z axes of each histogram represents value, normalized count and step-taken respectively. The attack on the unit makes the original distribution wider with a factor of roughly 2, whereas the attack on unit 4 has a different effect, where the whole distribution is shifted to the negative side.

the attack in a different way. In Figure 3.9b we see a unit that died completely after a learning rate attack. This is extremely interesting since the whole output distribution of the unit is shifted to the negative side after the attack (see Figure 3.10).

This shows us that even though it is less likely to see a successful learning rate attack in networks with ReLU, it is still possible. It is all possible.

3.7 Dead units may develop and MRS is an efficient measure to detect them

In this section, we observed unit with 3 different activation functions. For all of them, we were able to perform successful attacks where we see a considerable drop in MRP. Except for couple instances, MRS did a very good job approximating the MRS and never failed at choosing the smallest MRP unit. We expect MRS to perform even better with smaller MRP's due to the decreased error term in the Taylor decomposition.

We showed that ReLU+Batch Normalization activation behaves like a bounded activation function similar to *TanH*. Therefore their units are more likely to get affected by the learning rate attack. On the other hand, ReLU units are harder to harm with learning rate attack, however possible. We suspect that increased MRP on the attacked units causes slight decrease on MRP of other units. However, we haven't experiment with this hypothesis.

Measures like *Output gradient* and *l1-norm of the weights* are definitely related to the MRP, however, they are worst than the MRS at predicting low MRP units. Contrary to our expectations, we showed that a low MRP unit might have a very big norm (of weights) and pruning a unit according to its norm might be a very bad idea.

We also showed that the success of an attack is very related to the iteration it is made and the learning rate multiplier. We haven't investigated the effect of the outgoing connections on the success of an attack.

Note that we investigated only a small subset of all possible learning rate attacks. However, we believe that our findings are enough to convince the reader that a fix learning rate might cause some units to die and we may detect them efficiently using MRS.

Chapter 4

MRS based pruning with bias propagation

In Chapter 2 we defined and compared various saliency scores for individual parameters. However, when we look at the parameters of the pruned network, we observed that the individual units gather around some specific units. That motivated us to focus on units rather than the individual parameters. In Chapter 3 we defined the dead units and the Mean Replacement Penalty(MRP). Our motivation was that the MRP is the right measure for detecting units that either doesn't create any useful information or ignored by the rest of the network. We argued that the dead units are unlikely to change over the course of the training and we empirically observed this behavior. We also mentioned that replacing a unit with its mean and propagating that to the next layer is better than a bare removal. MRP directly measures this penalty of replacing a unit with its mean.

Calculating MRP for all units in a network is not feasible for large networks. However, we can calculate the MRS, first-order approximation of MRP, quite efficiently and use it to prune units. We can calculate MRS during the backward pass in linear time and in Chapter 3 we observed that MRS works pretty good at approximating the MRP, at least for the experiments we performed. In Section 4.1 we would like to compare MRS with norm-based saliency scores using the *randomScorer* as our baseline. In Section 4.2 we demonstrate unit pruning using the MRS and get superior results compare to the results of Chapter 1.

4.1 Comparing MRS with norm-based saliencies

In this section, we compare MRS saliency score with MRP and norm-based scoring functions. Our investigation is somehow limited since we only compare the scoring functions on a small network trained for MNIST classification. However, we believe that MRS would perform as good as norm-based saliency measures if not better in other networks and settings, too.

[26] compares various saliency scores in the context of unit pruning without bias propagation and concludes first-order Taylor approximation performs best. [39] uses bias propagation(explained in Chapter 1) idea to remove units which generate constant values. As stated earlier, mean replacement is a better way of removing units and should be employed. Combining the idea of mean replacement and the superiority of first-order Taylor approximation, we expect the MRS to perform better than the rest.

Saliency measures considered in this section are MRS, MRP and L1 and L2-squared based norms of the parameter tensors. Table 4.1 summarizes these scores along with their function names in **pytorchpruner.unitscorers** module. We included MRP to see how well the MRS approximates the MRP.

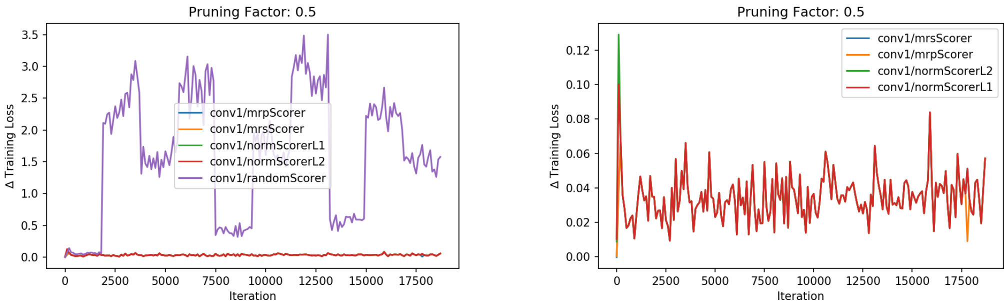
Saliency Score	**Definiton**
mrpScorer	see Chapter 3
mrsScorer	see Chapter 3
normScorerL1	$\|\|w\|\|_1$
normScorerL2	$\|\|w\|\|_2^2$
randomScorer	uniform

Table 4.1: Saliency scores that we compare in Section 4.1.

In our experiments we use the same experimental setup as the one in Section 2.1: small CNN on MNIST trained with SGD, batch-size of 32, the constant learning rate of 0.01 and without using momentum or weight decay for 10 epochs (18k+ iterations). We use ReLU as our activation function unless stated otherwise. We calculate the loss changes after performing the bias propagation calculated on a subset of the training set with size 1000. We expect these 1000 random samples to successfully approximate the training set accurately. Our small CNN reaches to 0.046 training loss and shows a pretty good generalization with 0.039 test loss and 98.8% accuracy. After every pruning followed by the calculation of the loss change, we revert all the changes such that the network trains as normally.

4.1.1 If you have few units, there is not much difference

We would like to start our comparison by looking at the very first convolutional layer. Since MNIST is a relatively easy task, even a small CNN with 8 units in the first layer obtains a pretty good accuracy. We prune 4 out of 8 units and calculate the change in the loss every 100 iterations. After every measurement, we revert the changes so that our measurement address the question of what if we prune at given iteration using a particular saliency measure. Changes in the training loss measured every 100 iterations during the 10 epochs of training for the first convolutional layer plotted in Figure 4.1a. Since there are only 8 units in the first layer, we don't see much difference between different saliency measures. They all seem like selecting the same 4 units during the entire training. But there is a significant difference between the *randomScorer* and the rest (Figure 4.1b). Some units have a very significant influence on the result and therefore pruning them with mean replacement causes a very significant change in the loss value(up to 100x more).



(a) randomScorer included

(b) randomScorer not included

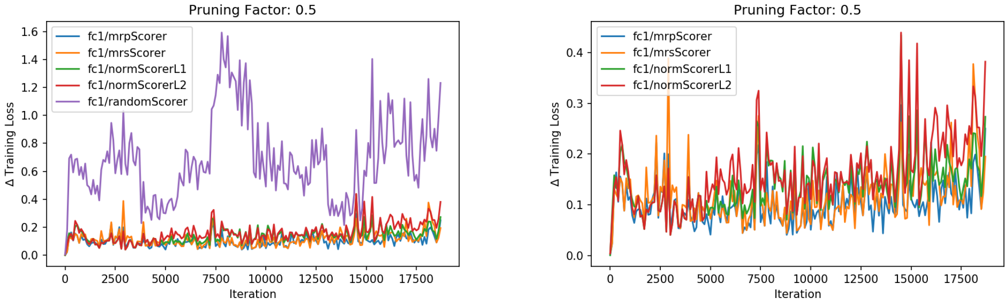
Figure 4.1: Change in the loss after replacing half of the units with their mean values in the first convolutional layer according to various measures explained in Section 4.1. Figure (a) is the same as the (b) except it includes our baseline *randomScorer* in addition to the other four.

4.1.2 Dead units having small norm

The results in 4.1a also suggests that units with low MRP also have small norm. In Section 3.2, we argued that dead units would most likely also be frozen (having no gradient signal). We also know that the parameters of the neural network tend to increase during the training and we also confirmed this tendency in our experiments. These two observations lead us to the opinion that the dead units(low MRS) score units may prevent the norm to increase and therefore making the norm-based saliency scores selecting the same units as the MRP selects.

4.1.3 MRP and MRS works better in general with ReLU

We see a more meaningful picture when we repeat the comparison we made in the previous section for the third layer(first fully connected layer). The results are plotted in Figure 4.2. Again in 4.2a we see a significant difference between the baseline and the rest. So our saliency measures all kind of work and we expect them to give reasonable results. However, if we look closely to 4.2b, we all see that MRP claims the lowest penalty in general and it is followed by the MRS. We also see that all the scoring functions performs quite similarly within the first 5000 iterations and then the norm based scoring functions start to diverge.



(a) randomScorer included

(b) randomScorer not included

Figure 4.2: Change in the loss after replacing half of the units with their mean values in the first fully connected layer according to various criteria. Measurements are made over the course of the training at every 100 iteration(step).

Since there are more units(256 units) in the third layer, selected units diverge more compare to the first layer and therefore it is a good candidate for performing additional experiments. In Figure 4.3 we compare the saliency measures for different pruning factors. Pruning factor is a number between 0 and 1 telling us the fraction of the parameters to be pruned. If we have 256 units and a pruning factor of 0.1, we prune $\lfloor 256/10 \rfloor = 25$ units with lowest saliency scores.

The first thing to observe is that occasionally we improve the loss value through pruning (negative loss change). When the pruning factor is 0.1, we observe the norm-based scoring functions working significantly worst at the beginning, however, they catch up towards the end and we don't see any significant difference between the 4 saliency measures. Increasing the pruning factor we observe the superiority of the MRP and MRS(blue and orange curves) at pruning. They almost constantly get the lowest scores.

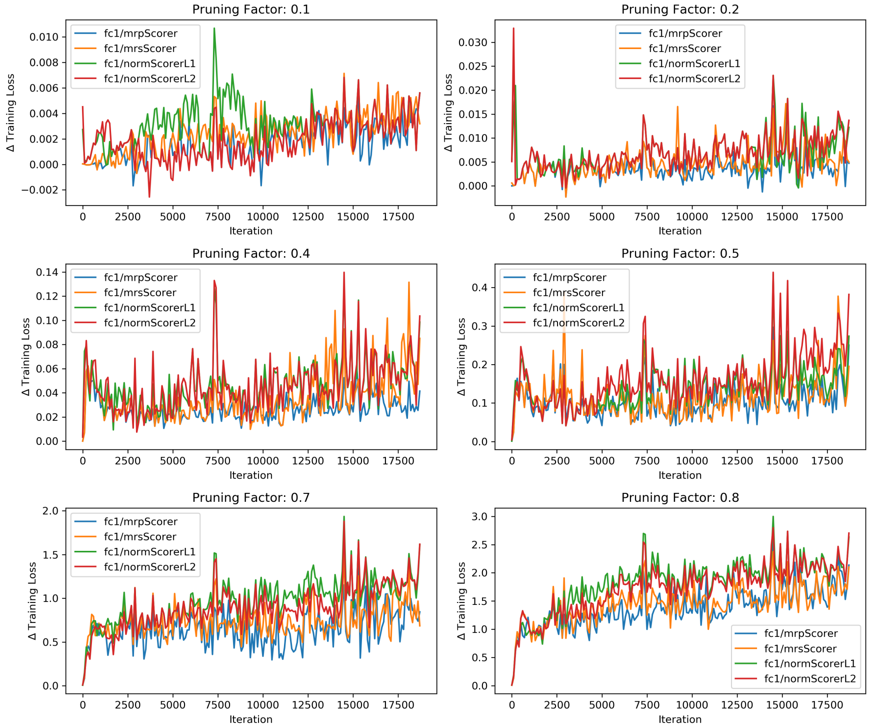


Figure 4.3: We calculate the change in the loss value after pruning the third layer(first fully connected layer) using various saliency measures. Each plot employs a different pruning factor, i.e. fraction of parameters in that layer to be pruned. Δ Training Loss is calculated on a randomly sampled training set of size 1000.

4.1.4 Mean replacement might not be the best strategy for Tanh

In the rest of this section, we continue our comparison with Tanh activation function (it was ReLU in the previous subsections). The small convolutional network with Tanh activation gets 0.053 training loss at the end of the training and gets a loss value of 0.056 and an accuracy of 98.31% on the test set.

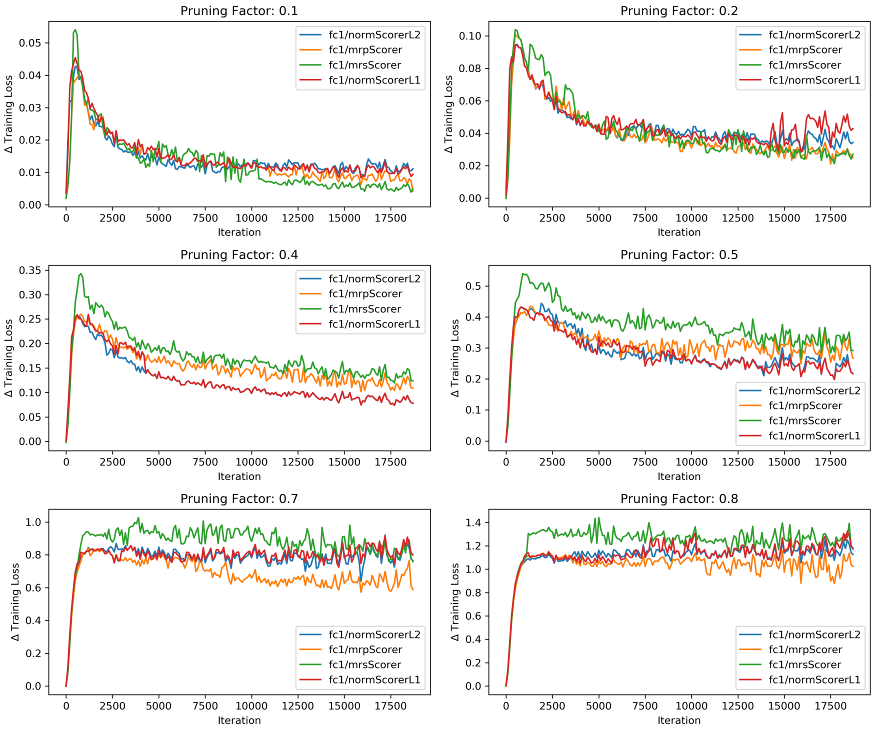


Figure 4.4: This plot has the exact same setting as the Figure 4.3 expect activation function is changed to Tanh. We calculate the change in the loss value after pruning the third layer(first fully connected layer) using various saliency measures. Each plot employs a different pruning factor and Δ Training Loss is calculated on a randomly sampled training set of size 1000.

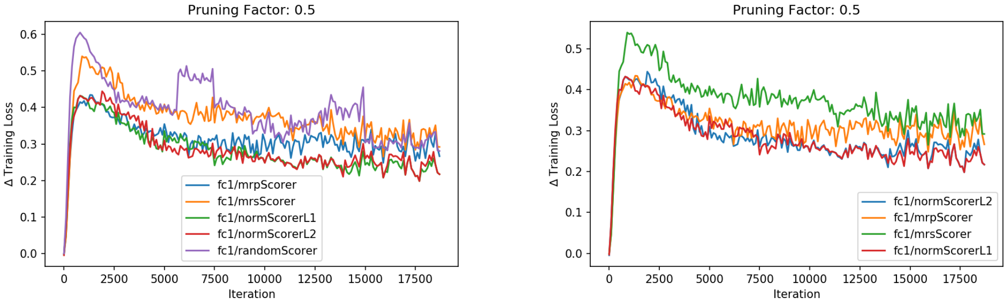
Figure 4.4 shows how our saliency measures perform at pruning the first fully connected layer(256 units). For pruning factors of 0.1 and 0.2 MRP and MRS seem like working better. However, for 0.4 and 0.5 norm-based pruning seems like inducing less loss penalty. The picture is slightly different in the last two plot, where the MRP gets the first place and MRS being the last.

More important than the rankings, the main observation here is that we get up to 7x more loss penalty (the positive change in the loss after pruning) compare to Figure 4.3(ReLU activation). This result is very striking since the final training and test loss values are pretty similar after 10 epochs for both networks. These results suggest that maybe the mean replacement is not the best removal method for all kind of activations. Maybe we shouldn't replace a mostly saturated Tanh unit that generates zeros or ones consistently with a value in the middle.

4.1.5 Units with Tanh activations are more tolerating

Figure 4.5 shows how our saliency measures perform at pruning the first fully connected layer(256 units) with a pruning factor of 0.5. Compare to Figure 4.2, our baseline(*randomScorer*) induces much smaller loss penalty (around 3x less) when we use Tanh. Our network with Tanh tolerates the absence of a random half of the units much better than the network with ReLu.

We also see that our saliency measures somehow have higher loss penalties over the course of the training. Our saliency measures do not perform significantly better than the baseline (random selection). This observation further supports our previous claim that the removal after bias propagation might not be the best method of pruning for networks with Tanh.



(a) randomScorer included

(b) randomScorer not included

Figure 4.5: Change in the loss after replacing half of the units with their mean values in the first fully connected layer according to various saliency measures. Measurements are made over the course of the training at every 100 iteration(step) using a randomly selected subset of the training data(size 1000).

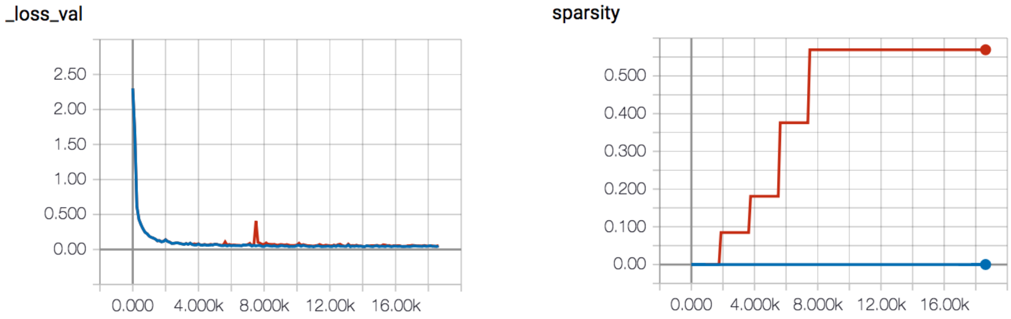
4.2 Pruning a small network with MRS

In this section, we perform unit pruning using the MRS saliency measure. We use the same experimental setup in Section 2.2 (same learning rate, model,random seed,etc.). We slightly changed our pruning schedule reducing the target pruning factors(see Table 4.2). We perform bias propagation before removing units at every pruning round. Removal of a unit is simulated through masking the parameters of the units. We calculate the MRS score using a randomly sampled subset of the training data.

Epoch	2	3	4	5
Fraction f	0.1	0.2	0.4	0.6

Table 4.2: Pruning schedule for the unit pruning experiment on MNIST

Figure 4.6 shows the training loss and pruning fractions for 2 different runs of the same experiment: one with pruning(red) and another without(blue). Right after the last two pruning round (fractions 0.4 & 0.6), we observe small peaks in the loss curve (Figure 4.6a). However, both runs approach to zero after 10 epochs and get 98.42% accuracy on the test set. The pruned network has only 4019 parameters, where the original network has 20522 (5x reduction!). Note that, since we remove entire units, pruned parameter tensors can be copied into smaller ones. Therefore pruning units don't require sparse representations and therefore we expect to see greater speed ups and smaller memory footprint.



(a) loss

(b) fraction of parameters pruned

Figure 4.6: Loss and fraction of pruned parameters over time. The x-axis represents time and parameterized by the iteration (step) number. The red curve is the run with scheduled pruning, where the blue curve is the original run without any pruning done. We are using the same seed for random number generator to ensure consistency and reproducibility.

4.2.1 Magnitude Colormaps of Parameters

As we did in Section 2.2, we plot the magnitude colormaps of our layers. Since we use the same exact random seed, the original colormaps should match with the ones in Chapter 2. The reader is encouraged to compare the two pruning strategies and enjoy similarities between the two experiment.

Colormaps in Figure 4.7,4.8,4.9 and 4.10 depict magnitudes after the training for the the first four layers of our network. We observe that the magnitudes of the parameters in the surviving units seem unaffected by the pruning.

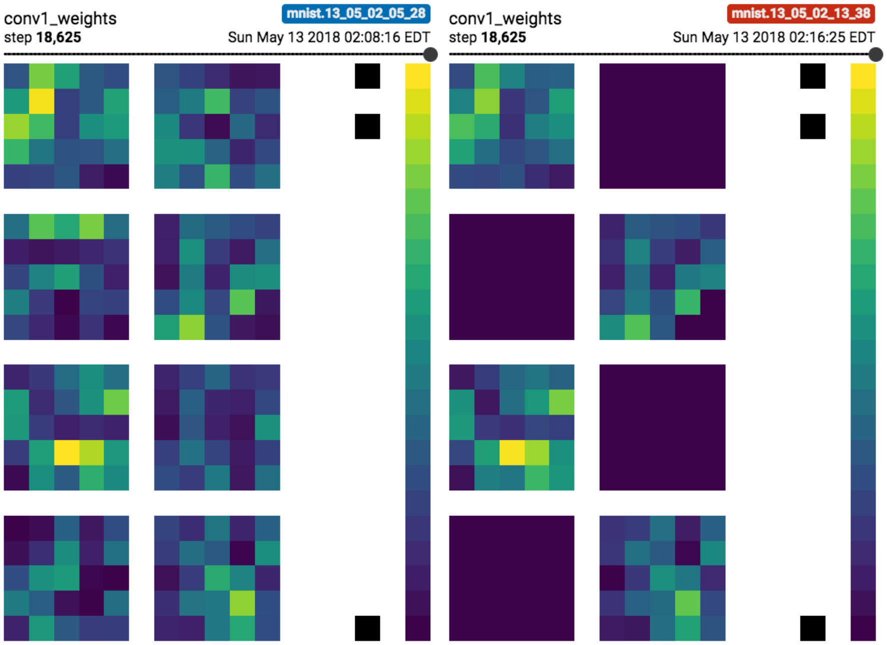


Figure 4.7: Magnitude colormaps of the weight tensor of the conv1 layer with(right) and withoutleft pruning. Colormaps are generated after the training and all values are normalized such that yellow represents the max-value and blue represents the minimum value.

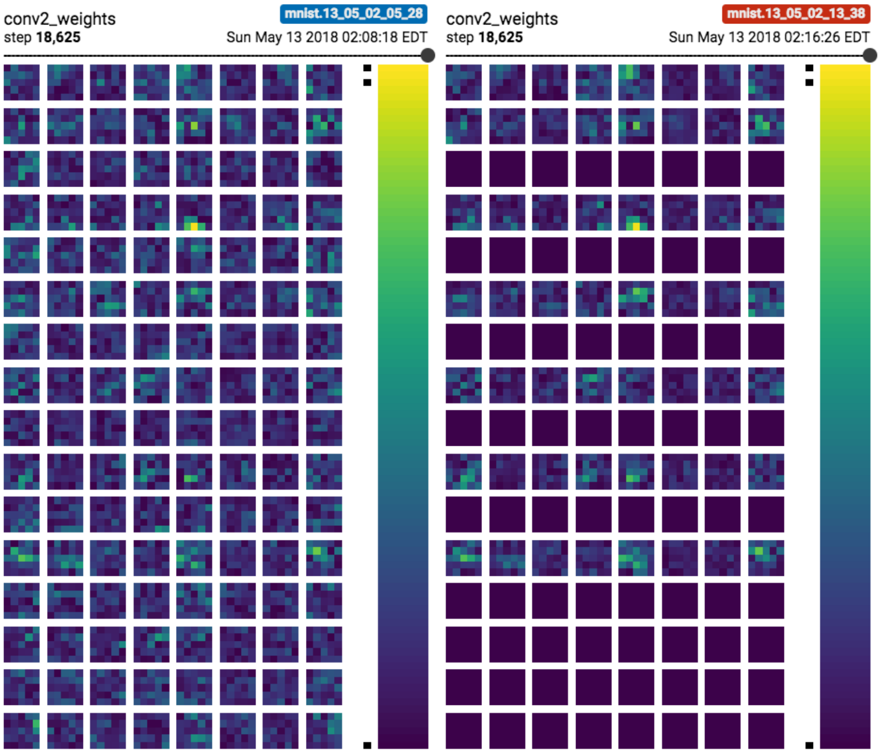


Figure 4.8: Magnitude colormaps of the weight tensor of the conv2 layer with(right) and withoutleft pruning. Colormaps are generated after the training and all values are normalized such that yellow represents the max-value and blue represents the minimum value.

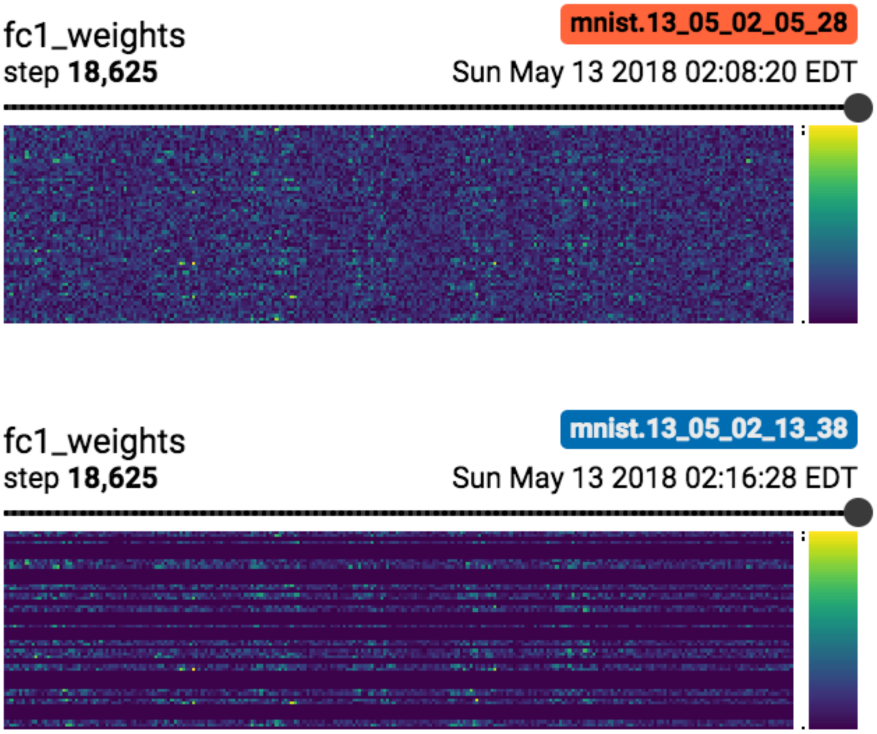


Figure 4.9: Magnitude colormaps of the weight tensor of the fc1 layer with(right) and withoutleft pruning. Colormaps are generated after the training and all values are normalized such that yellow represents the max-value and blue represents the minimum value.

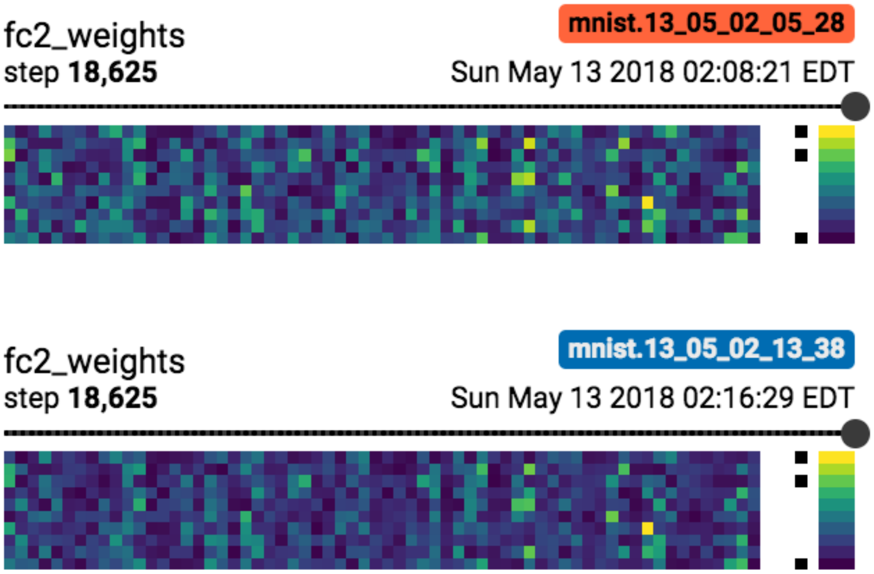


Figure 4.10: Magnitude colormaps of the weight tensor of the fc2 layer with(right) and withoutleft pruning. Colormaps are generated after the training and all values are normalized such that yellow represents the max-value and blue represents the minimum value.

Appendix A

a Pytorch Pruner Library: pytorchpruner

`pytorchpruner` is a pytorch package for pruning neural networks. It is intended for research and its main objective is not to provide fastest pruning framework, however, it is relatively efficient and fast. It uses masking idea to simulate pruning and supports two main pruning strategies. It also implements various second-order functions to calculate Hessian and Hessian-vector products.

`https://github.com/evcu/pytorchpruner`

There are 5 main parts of this library.

1. **Parameter Pruning (pytorchpruner.scorers)**: has saliency measures that return a same-sized-tensor of scores for each parameter in the provided parameter tensor.
2. **Unit Pruning (pytorchpruner.unitscorers)**: has saliency measures that return a vector of scores for each unit of the provided parameter tensor.

3. **Pruners (pytorchpruner.pruners)**: has two different pruner engine for the two different pruning strategies (parameter vs unit). `remove_empty_filters` function in this file reduces the size of the network by copying the parameters into smaller tensors if possible.

4. **Auxiliary Modules (pytorchpruner.modules)**: implements `meanOutputReplacer` and `maskedModule`, the two important wrappers for `torch.nn.Module` instances. The first one replaces its output with the mean value if enabled. And the second one simulates the pruning layers.

5. **Various first/second-order functionality (pytorchpruner.utils)**: implements hessian calculation, Hessian-vector product, some search functionality and some other utility functions.

Appendix B

Experiment Bootstrapper Library

This package includes the bootstrap code for generating experiments and includes helper functions/scripts/examples for pytorch training and slurm job scheduling. The release can be cloned from the following public GitHub repo.

`https://github.com/evcu/exp.bootstrp`

The very basic/main idea is:

- Create a python script(experiment) which accepts command line arguments
- Provide some argument lists and generate SLURM jobs using the cross product of the given argument lists.
- Make all above with ease and allow easy reproducibility.

A quick walkthrough of library can be found in the repository above.

B.0.1 Features

- **read_yaml_args** reads conf.yaml and creates a type-checked argParser out of the definition. Write the conf, read and overwrite with CLI args.

- **Customizable eval-prefixes** inside yaml file, which enables defining programatic eval-able arguments. i.e. the string '+range(5)' would be evaluated and read as a list.
- Current configuration of the experiment **is copied to the experiment folder** such that you can always change experiments default_args after submission
- **ClassificationTrainer/ClassificationTester** which wraps the main training/testing functionalities and provides hooks for loggers.
- tensorboardX **logging utils** and examples.
- **convNetGeneric** implementation
- **Multiple experiment definitions** through yaml lists.

Appendix C

Model Definitions

C.1 small CNN for MNIST

Following model is used for experiments in Chapter 2 and Chapter 4. Note that the first layer is wrapped with *meanOutputReplacer* wrapper to simulate mean-replacement during forward pass and/or calculate MRS score.

```
ConvNet_generic(
  (conv1): meanOutputReplacer(
        module=Conv2d (1, 8, kernel_size=(5, 5), stride=(1, 1))
        ,is_mean_replace=False
        ,enabled=False)
  (conv2): Conv2d (8, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=256, out_features=32)
  (fc2): Linear(in_features=32, out_features=10))
```

C.2 small CNN for CIFAR

Following model is used for experiments in Chapter 3

```
ConvNet_generic(
  (conv1): Conv2d(3, 8, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(8, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

Appendix D

Loss curves of attacked units

In this section we provide the loss curves of the experiments shared in Chapter 3. Each plot has a caption telling which figure they match in Chapter 3.

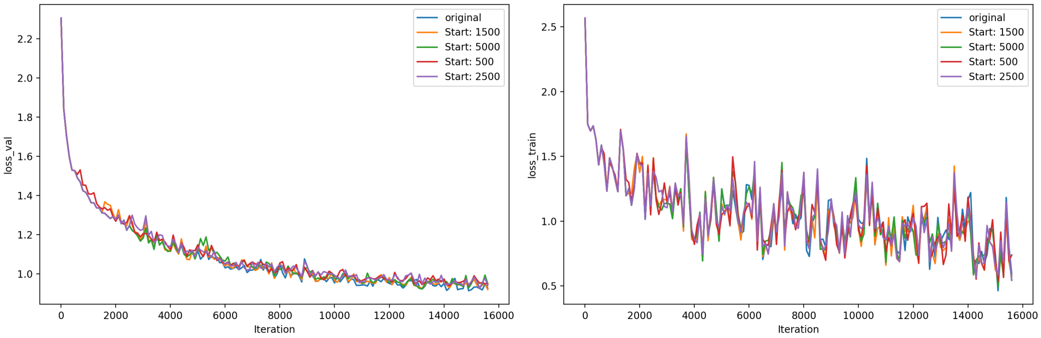


Figure D.1: Loss curves of the Figure 3.3a

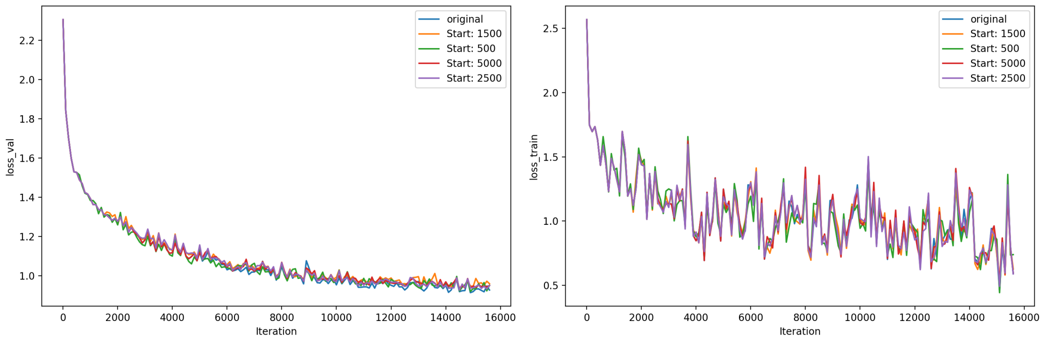


Figure D.2: Loss curves of the Figure 3.3b

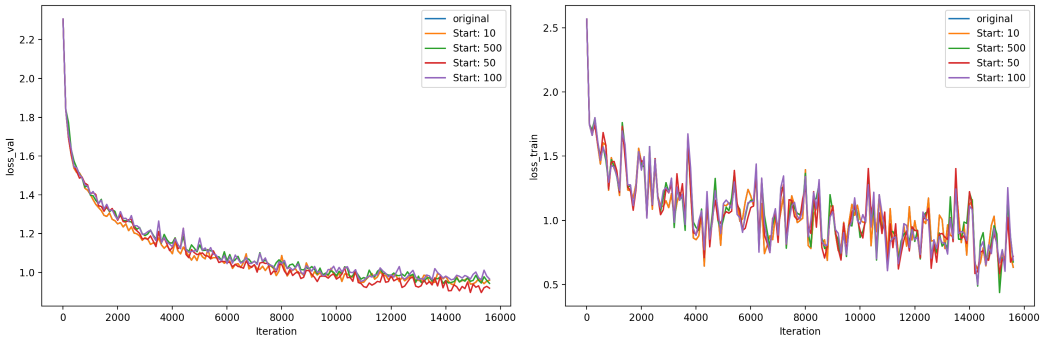


Figure D.3: Loss curves of the Figure 3.4a

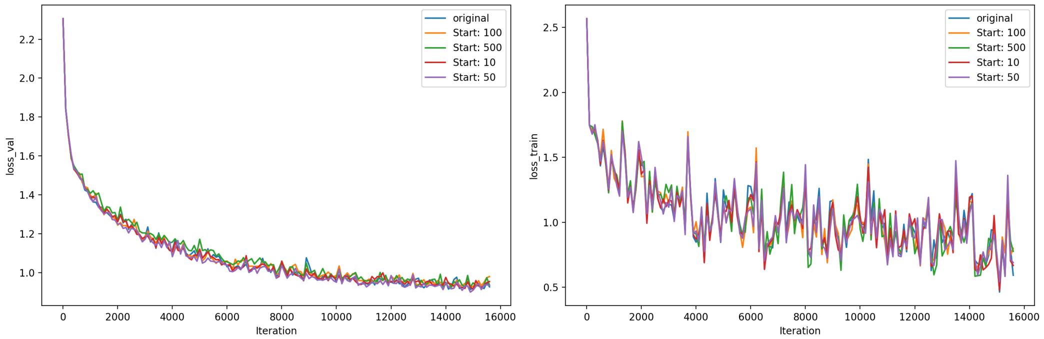


Figure D.4: Loss curves of the Figure 3.4b

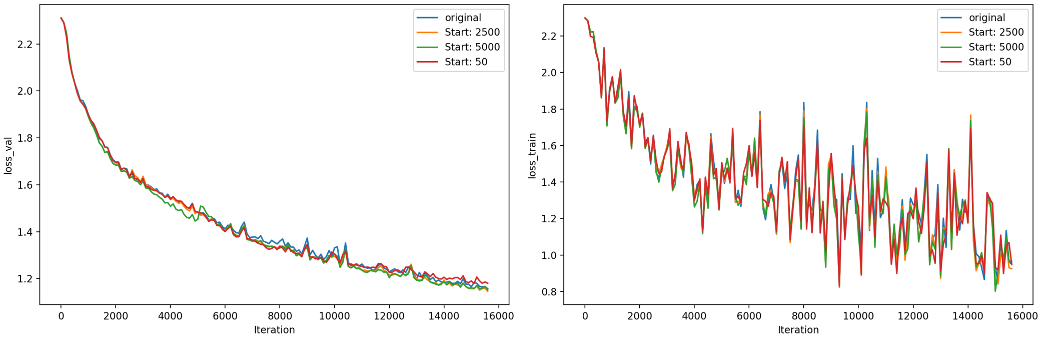


Figure D.5: Loss curves of the Figure 3.7a

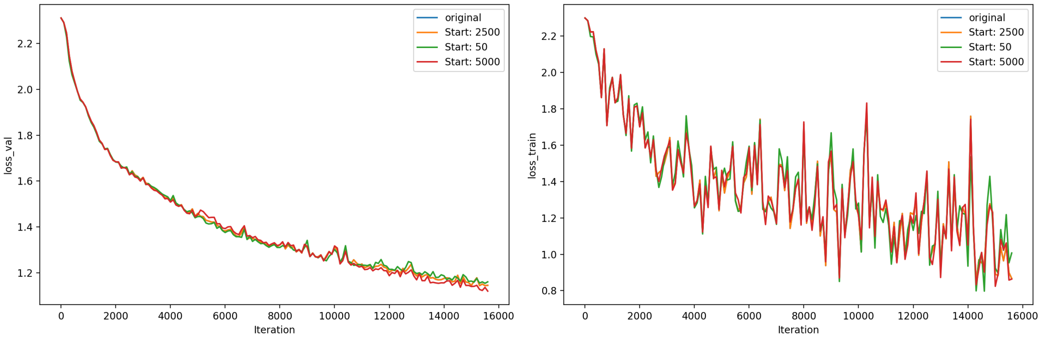


Figure D.6: Loss curves of the Figure 3.7b

Appendix E

Unit Saliency Measure Comparison: Additional Plots

In this section we provide additional results for Section 4.1. We

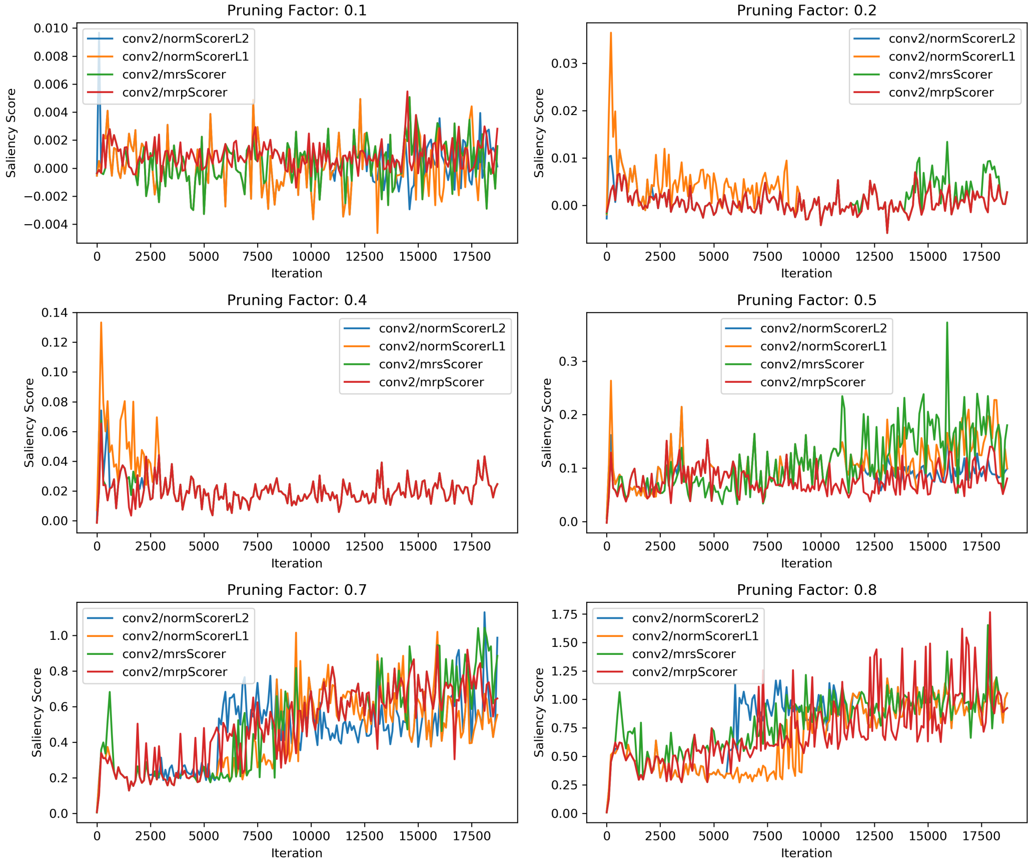


Figure E.1: We calculate the change in the loss value after pruning the second convolutional layer using various saliency measures. Each plot employs a different pruning factor, i.e. fraction of parameters in that layer to be pruned. Δ Training Loss is calculated on a randomly sampled training set of size 1000.

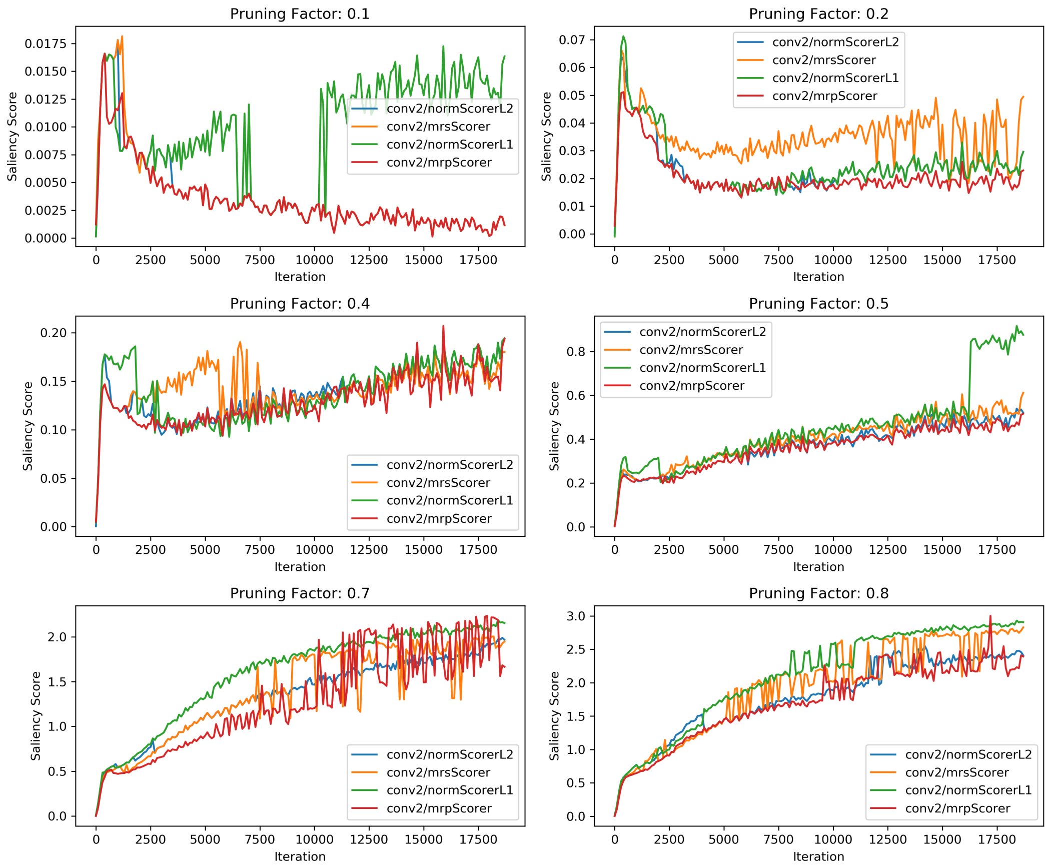


Figure E.2: Same setup as the Figure E.1 except units have Tanh activation.

Bibliography

[1] Achterhold, J., Koehler, J. M., Schmeink, A., and Genewein, T. Variational Network Quantization. In *International Conference on Learning Representations* (2018).

[2] Cheng, Y., Wang, D., Zhou, P., and Zhang, T. A Survey of Model Compression and Acceleration for Deep Neural Networks. 1–10.

[3] Choi, Y., El-Khamy, M., and Lee, J. Towards the Limit of Network Quantization. 1–14.

[4] Courbariaux, M., Bengio, Y., and David, J.-P. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. 1–9.

[5] Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. Predicting Parameters in Deep Learning. 1–9.

[6] Denton, E., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. 1–9.

[7] Foroosh, H., Tappen, M., and Penksy, M. Sparse Convolutional Neural Networks. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 806–814.

[8] Frosst, N., and Hinton, G. Distilling a Neural Network Into a Soft Decision Tree. *arXiv 1711*, 09784 (nov 2017), 8.

[9] Galloway, A., Taylor, G. W., and Moussa, M. Attacking Binarized Neural Networks. In *International Conference on Learning Representations* (2018).

[10] Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. Gene Selection for Cancer Classification using Support Vector Machines. *Machine Learning 46*, 1 (jan 2002), 389–422.

[11] Han, S., Mao, H., and Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. 1–14.

[12] Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both Weights and Connections for Efficient Neural Networks. 1–9.

[13] Hanson, S. J., and Pratt, L. Comparing Biases for Minimal Network Construction with Back-Propagation. *Advances in neural information processing systems 1*, May (1989), 177–185.

[14] Hardt, M., and Vinyals, O. Understanding Deep Learning Requires Re-Thinking Generalization. *International Conference on Learning Representations 2017* (2017), 1–15.

[15] Hassibi, ., and Stork, D. Second order derivaties for network prunning: Optimal brain surgeon. *Advances in NIPS5* (1993), 164–171.

[16] Hinton, G., Vinyals, O., and Dean, J. Distilling the Knowledge in a Neural Network.

[17] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors. 1–18.

[18] Kohavi, R., and John, G. H. Wrappers for feature subset selection. *Artificial Intelligence 97*, 1-2 (1997), 273–324.

[19] Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. *... Science Department, University of Toronto, Tech. ...* (2009), 1–60.

[20] Kumar, S. K. On weight initialization in deep neural networks. 1–9.

[21] LeCun, Y. *Modeles connexionnistes de l'apprentissage (connectionist learning models)*. PhD thesis, Université P. et M. Curie (Paris 6), jun 1987.

[22] LeCun, Y., and Cortes, C. {MNIST} handwritten digit database.

[23] LeCun, Y., Denker, J. S., and Solla, S. A. Optimal Brain Damage. *Advances in Neural Information Processing Systems 2*, 1 (1990), 598–605.

[24] LeCun, Y., Denker, J. S., and Solla, S. A. Optimal Brain Damage. *Advances in Neural Information Processing Systems 2*, 1 (1990), 598–605.

[25] Lin, Z., Courbariaux, M., Memisevic, R., and Bengio, Y. Neural Networks with Few Multiplications. 1–9.

[26] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning Convolutional Neural Networks for Resource Efficient Inference. 1–17.

[27] Morcos, A. S., Barrett, D. G. T., Rabinowitz, N. C., and Botvinick, M. On the importance of single directions for generalization. In *International Conference on Learning Representations* (2018).

[28] Neyshabur, B., Bhojanapalli, S., McAllester, D., and Srebro, N. Exploring Generalization in Deep Learning.

[29] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[30] Pearlmutter, B. A. Fast Exact Multiplication by the Hessian. *Neural Computation 6*, 1 (1994), 147–160.

[31] Polino, A., Pascanu, R., and Alistarh, D. Model compression via distillation and quantization. In *International Conference on Learning Representations* (2018).

[32] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. 1–17.

[33] Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* (1958), 65–386.

[34] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors, 1986.

[35] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural Networks 61* (2015), 85–117. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[36] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. 1–19.

[37] Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning Structured Sparsity in Deep Neural Networks.

[38] Wu, S., Zhong, S., and Liu, Y. Deep residual learning for image steganalysis. *Multimedia Tools and Applications* (2017), 1–17.

[39] Ye, J., Lu, X., Lin, Z., and Wang, J. Z. Rethinking the Smaller-Norm-Less-Informative Assumption in Channel Pruning of Convolution Layers. 1–11.

[40] Zhu, M., and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression.