Understanding a deep Neural network

Srijita Das

ABSTRACT

Deep neural networks find it's application in a wide variety of domain and they have been performing consistently well in these domains ranging from medicine to speech recognition tasks. However, most of the time, these networks are used as a black box giving extremely good classification results. Until recently, little effort has been put in understanding the various intricacies of a deep neural network. In this paper, we explore the various learning patterns of a deep neural network. We try visualizing the gradient patterns of each layer with different initialization scheme and try understanding what these initialization schemes are actually doing in the solution space. We also try studying the impact of momentum on learning in such networks. Lastly, we try exploring the activation functions for each layer to answer the observed gradient patterns in different layers of a deep Neural network.

INTRODUCTION

Deep architectures like multi-layer neural network, autoencoders and convolutional neural networks have been very popular in recent times in the machine learning community. Much of the attention received by these deep architectural networks are due to their immense success in classification tasks in a wide variety of domains including Computer vision, Medical Imaging and Natural Language Processing. All of these architectures consists of several layers of feature abstraction by introducing some non-linearity. These non-linear functions in such networks leads to useful representation of the features which finally results in improved classification accuracy. In this project, we focus our attention on one such deep architecture which is the multi-layer neural network. These neural networks have many hidden layers and are therefore called deep neural networks.

A few years back, it was believed that deep neural networks were hard to train using simple Stochastic Gradient descent as the objective function was non-convex and SGD on such deep network leads to poor solutions as the function would get trapped in local basins and continue oscillating within such regions. In 2006, Bengio et al. [1] introduced the idea of greedy layer wise pre-training of the layers of a neural network. This unsupervised pre-training seemed to initialize the parameters of the network in a much better solution space from where fine tuning the network using SGD worked reasonably well. This method had been used for a long time for training deep neural networks. This process was effective and worked well for a large number of tasks in Computer vision([5],[6]) and Speech recognition([8]).Hinton and Shalakutdinov[4] got superior results in training a deep autoencoder with this pre training and fine tuning procedure which was a bench mark result at that time.

However, Martens [7] came up with a second order optimization method called Hessian free optimization which along with a clever initialization technique called sparse initialization was able to achieve the same results as that of Hinton and Shalakutdinov. They used a Hessian approximation in their procedure and thus included curvature information to better guide the gradient descent procedure. Second order curvature information allows the gradient descent method to descent more in directions of low curvature where the rate of change of gradient is low as compared to directions of high curvature. Apart from that, they used an initialization technique called sparse initialization in which they limit the number of non-zero weight components of each weight vector associated with each neuron to 15. Doing this helped them to overcome the problem of saturation while doing back propagation in neural networks.

Following that, a lot of first order optimization techniques with some well defined initialization schemes have gained prominence in training deep neural networks performing almost as well as a pretrained neural network or second order gradient descent techniques. The use of momentum to accelerate neural network training was studied before as in [10] and was not found to be that effective in training deep neural network. Later on, Sustskever et al. [11] studied the effect of momentum with a well defined initialization scheme and they were able to get almost as good result as that of Marten's. They applied both the classical and the Nesterov's accelerated momentum on deep autoencoders along with sparse initialization technique of Marten's and got superior results. They also found out that Nesterov's accelerated gradient descent technique performs better than the classical momentum when applied with a well defined momentum schedule and initialization.

A problem which is commonly associated with deep neural network, especially recurrent neural network is the problem of vanishing gradient descent.[2] This is the problem of the gradient vanishing in the lower layers while the error propagates backwards from the top layers during the training of a deep neural network. The intermediate neurons in such a network becomes saturated in the beginning of the training thus preventing the error from propagating backwards. As a result, the top layers learn quickly whereas the lower layers of such a network learns very slowly as the effect of change of parameters on the objective function is negligible for these layers. In such a scenario, the lower layers have to wait for the units of the topmost layers to escape saturation. When the neurons become unsaturated, then the bottom layers of such networks start learning. In some networks of certain depth (depth five network with sigmoid activation), the units of the topmost layer never escaped saturation as seen by Glorot et all [3].

From all the related work, we get an idea of the optimization techniques and the initialization schemes that work or does not work in a multi-layer neural network. While a lot of work has been done in the optimization side of deep networks, little attention has been focused on study of things like activation functions and gradient flow across a deep neural network. It has been shown that few clever initialization schemes do work well on a neural network, but why do they work well has not that clearly been answered. The most closest paper to my knowledge answering this question has been by Glorot et al.[3] in which the author tried understanding the gradient flow and activation function saturation inside a multi layer neural network. They compared various kinds of activation functions like sigmoid, tanh and rectified linear units and showed how each of these activation functions affect the gradient back propagation inside a deep neural network as well as visualised the activation function inside a deep network. They showed how the different layers got saturated in a deep neural network using different activation functions. Finally, they also came up with an initialization scheme which helps the network out of saturation so that the gradient back propagates well and the different layers are able to learn more quickly than waiting for the top layers to come out of saturation.

The above mentioned paper has been our main motivation in pursuing this project. In this project, we try to explore the various aspects of a multi layer neural network. We study the propagation of gradients across different layers in a neural network for random and sparse initialization techniques to see if we can see the vanishing gradient problem persisting in deep neural networks. We also try to analyze which initialization scheme performs better in a deep network and what are its implications. Next, we analyze the activation functions in various layers of a neural network to see if we can relate these activation functions to the gradient patterns across different layers. We also try seeing if applying momentum accelerates learning in such deep networks and eliminates the vanishing gradient problem that has been persistent in deep neural networks.

BACKGROUND

A neural network consists of several layers of neurons stacked one after the other as shown in figure 1. It can be viewed as stacks of generalized linear models.

The first layer of a neural network consists of the features of a single data point. Each layer consists of several number of units called neurons. This draws it's motivation from human brain where synaptic neurons are

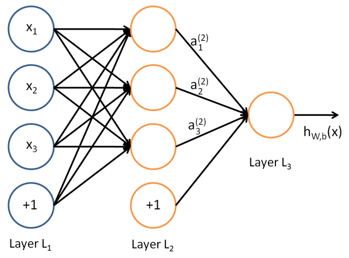

Figure 1: A single hidden layered neural network

present and each neuron is fired by some activation function.Every unit in a layer is connected to all other units in the next layer by some unknown weight vectors. Every unit in a layer of a neural network is activated by linear or non-linear functions called activation function.

In our experiments, we have considered the quadratic loss which is

$$C = \frac{1}{2n} \sum_x ||y(x) - h_{W,b}(x)|| \tag{1}$$

y(x) in the above equation is the true output and $h_{W,b}(x)$ is the activation function of the last layer of the network which can be seen as the predicted output.

We implemented the most common implementation of gradient descent called back propagation to learn the weight parameters and the bias of each layer.Each layer except the first has a weight parameter and a bias parameter which the learning algorithm learns during training phase.The activation associated with a layer l can be defined as below:

$$sa^l = \sigma(z^l) \tag{2}$$

Each component of the matrix z^l is further written as

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{3}$$

In the above equation, w_{jk}^l is the weight associated in layer l with a neuron j from all k neurons in the previous layer (l-1). b_j^l is the bias of neuron j associated with layer l.

There are two gradients associated with a neural network. One is the rate of change of Objective function with respect to the associated weights in each layer and the other is with respect to the respective biases of the neurons associated with each layer. The equation for the two gradient updates are as below:

$$\frac{\partial C}{\partial b} = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{4}$$
$$\frac{\partial C}{\partial W} = a_{in} \odot \delta_{out} \tag{5}$$

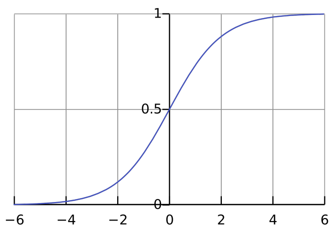

Figure 2: Sigmoid function

a_{in} in the above equation is the activation matrix of the neurons in the previous layer. δ_{out} in the above equation is the error that is propagated backwards as a result of any change in activation in the intermediate layers. The expression for δ is calculated as per the below mentioned equation for the last layer l and this error back propagates to the lower layers of the network.

$$\delta = (a^l - y(x)) \odot \sigma^{'}(z^l) \tag{6}$$

The gradient of a neural network can become extremely small due to two reasons. It can be small if the incoming activation to the current layer from the previous layer is extremely small. This can be seen from equation 5. Another reason why a gradient can be extremely small is if the quantity δ_{out} from equation 5 becomes extremely small. This can happen if the derivative of the sigmoid function is extremely small. Derivative of a sigmoid function becomes negligible when the sigmoid function approaches 0 or 1 as seen in the figure 2. In Neural networks, the gradient starts becoming very small when the activation of the neurons of a layer either becomes too small(almost 0) or becomes too large(almost 1). This is when we say that the neurons have become saturated and the network layer stops learning and the gradient more or less remains constant. The vanishing gradient problem also occurs due to this very small derivative of sigmoid function when the neurons get saturated. The maximum value of $\sigma^{'}(z)$ is 0.25 when z is 0. For the bottom layers, the multiplication of this derivative term repeatedly, once for every layer finally leads to almost a vanishing gradient value for the bottom most layer of the network.

EXPERIMENT

All our experiments were carried out on the publicly available MNIST dataset which has 784 features. There are 50000 data points in the training set and 10000 data points in the test set. The neural network on which we carried out our experiments had 4 hidden layers. This is a 5 layered neural network with 5 gradients, one for each layer. The network has sigmoid activation function applied at each of it's layers. We considered quadratic loss for our experiments. We used the publicly available Neural network code base of Michael Neilsen [9] as the base for our experiments on which several changes were done.

Back propagated gradient observation with Random Initialization

We studied the behavior of the back propagated gradient in a 5 layered neural network. We used a learning rate of 0.3 and a mini batch update of 10. This means that the noisy gradient across all the layers of a Neural network was updated with the gradient estimate of every 10 data points. We plotted the back propagated gradients with respect to the weight parameters as well as the bias parameter as seen in the Figures 3 and 5 respectively.We plotted the L2 norm of the gradients in each of the two cases after every mini batch update for 10 epochs during training. We initialized the biases with 0 and the weights with an uniform distribution of $[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$ where n is the size of the previous layer. This is one of the common initialization scheme which is used to train a neural network.

From the graph of the back propagated gradients plotted against 10 epochs of training, we can see quite a few interesting observations. We see that for random initialization, the vanishing gradient problem can be seen during the first two epochs of training. During these two epochs, the top layer learns very less as can be seen from the very small gradient of the top layer in Figure3. Towards the beginning of the third epoch, the top layer reaches it's learning peak, that is the gradient of the top layer reaches it's maximum value. After this, the gradient again decreases and reaches a consistent value which remains the same for the rest of the training epochs.As soon as the top layer reaches it's learning peak, the other bottom layers start learning as is evident from the figure. We can also see from the figure that all the 5 layers of the network has different learning speed throughout the training as they have different patterns of gradient during the 10 epochs.

Another observation from the graph of the back propagated gradient is that the top most layer shows the minimal variance of the gradients.This variance increases as we go towards the bottom layers with the bottom most layer showing the highest variation in gradient. The gradient of the bottom layer is very large as compared to other four layers. This is because of the fact that the 100 neurons in the first hidden layer tries to learn the 784 features of the input dataset as opposed to other layers where 100 neurons learns 100 abstract features of the previous layer.

We also plotted the gradient with respect to the bias parameter for 10 epochs to see if any significant difference is found between the two gradients-one with respect to bias and the other with respect to weight parameters. The graph for this change in gradient can be seen in Figure 5. No significant difference in learning was found between these two gradients. However, the difference in learning rates between the different layers are less for the gradient with respect to bias than the gradient with respect to the weight parameter. From the graph, we can see that the top 2 layers(4 and 5) learn almost at the same rate and the bottom two layers(layer 1 and 2) learns at almost the same rate.

We also carried out the same experiment to observe the gradients in a 2 layered neural network to see if the vanishing gradient problem persists for shallow neural networks. The graph which has back propagated gradient magnitude in the y-axix and intervals of mini batch update for 1 epoch in the x axis is shown in Figure 4. For such a shallow network, we could not see the vanishing gradient problem and right from the beginning of the epoch, the lower layer learns at a greater speed than the topmost layer. This makes us conclude that the vanishing gradient problem is not an issue with shallow neural networks.

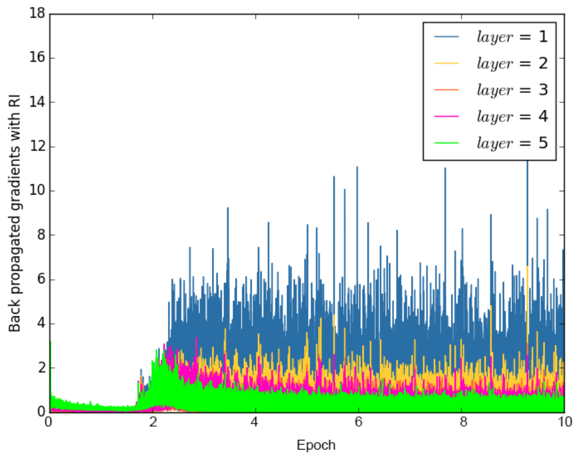

Figure 3: Plot of the back propagated gradient magnitude with respect to weight along y-axis and the epochs along x-axis for the 5 layers of Neural network with Random Initialization.

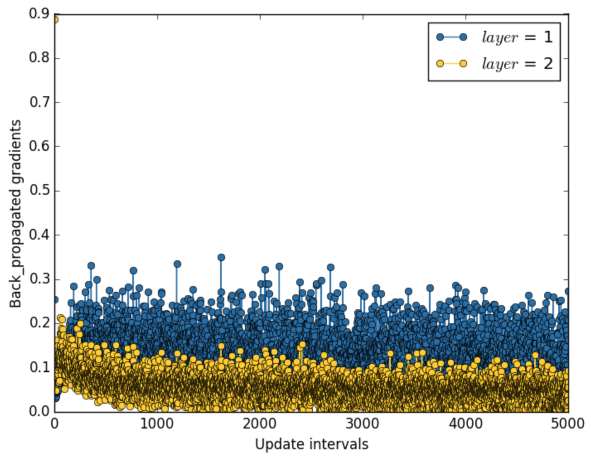

Figure 4: Plot of the back propagated gradient magnitude with respect to weight along y-axis and the update intervals of 10 minibatch along x-axis for a single hidden layered Neural network for 1 epoch.

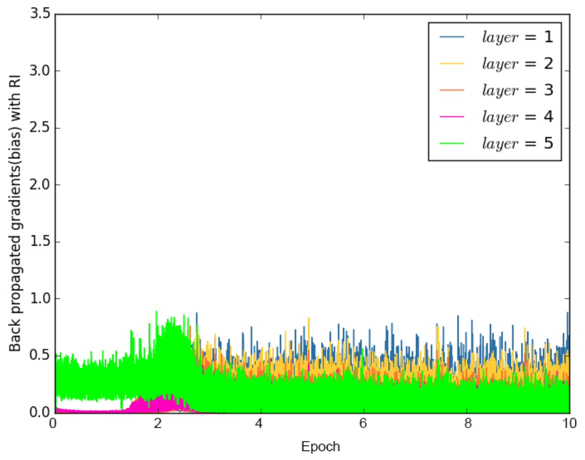

Figure 5: Plot of the back propagated gradient magnitude with respect to bias along y-axis and the epochs along x-axis for the 5 layers of Neural network with Random Initialization.

Test accuracy with Random initialization

From the previous section, we saw that the top layer reaches it's learning peak at third epoch and then the other layers starts learning. In order to see if learning of the top layer is sufficient to learn a good model, we calculated the test accuracy for 10 epochs. Test accuracy for each epoch has been averaged over 10 runs with 3000 random test points from the test dataset. The test accuracy with Random initialization is shown in the Table 1.

From Table 1, we can see that the test accuracy takes a big leap from 40% to 89% when the top layer reaches it's learning peak in the third epoch. After this, the test accuracy increases slowly and reaches a high of 95.9%. These results makes us conclude that it is the top layer of a deep network that drives the learning. More specifically, the learning of the top layer places the parameters in a good solution space close to a good local minimum. After the top layer finishes it's learning, the other layers including the bottom layer essentially fine tunes the solution and hence, the accuracy increases slowly from that point. The learning of the bottom layer takes the parameters placed in a good solution space by the top layer slowly towards a favorable local minima.

Activation Function analysis for Random Initialization

In order to hypothesize why the gradients were vanishing in the beginning epochs of the training phase, we tried plotting the activation function of the various hidden layers of a neural network. We plotted the mean activation function values for each layer taken at an interval of every 10 mini batch updates.We plotted data for 8 epochs. The activation functions were calculated on a

Epoch	Accuracy(%)
1	12.343
2	40.036
3	89.456
4	93.466
5	94.273
6	95.06
7	95.606
8	95.766
9	95.556
10	95.91

Table 1: Test accuracy with Random Initialization over 10 epochs

fixed test set of 300 data points after every mini batch update during the training.

The activation function plot for a 5-layered neural network is shown in Figure 6. From this graph, we see that the top most layer of the neural network maintains a very low activation of 0.1 (almost saturation) all throughout the training phase and the activation for this layer never increases during the entire training phase. This behavior was also observed by Glorot et al.[3] in their paper. The penultimate layer starts at a high mean activation value but drops to lower activation very quickly and continue maintaining this low activation(a little greater than the top layer) during the training. The other layers maintain a somewhat high mean activation of around 0.5 which slowly starts diminishing with time. This plot somewhat explains the gradient behavior as seen in Figure 3. From equation 5, we see that the gradient with respect to the weight parameter depends on the incoming activation of the previous layer as well as the error output from that layer which in turn depends on the activation of that layer. A very high or very low activation value of the topmost layer would lead the gradient to be small by making the value of the derivative of the sigmoid function small. From the graph, we somewhat understand the low gradient of the topmost layer is due to the low activation value of the topmost layer. The somewhat consistent gradient all through the remaining training phase after reaching the learning peak can be attributed to the activation of the previous layer which remains a little more than the activation of the top layer all throughout the epochs.It is interesting to see that the bottom layer of the network, in spite of having a high mean activation value have vanishing gradients during the early epochs of the training. This vanishing gradient problem may be attributed to the low mean activation of the top layers (layer 4 and 5) in the early epochs, though more deep understanding of the activation functions is required to reach to the root cause of this issue.

Another observation that we notice from these graphs is that the activation function has variance in the points where learning takes place. For the top layer, we see variance at the end of the second epoch and beginning of the third epoch. This is the point when the learning of the top layer reaches it's peak. The weight parameters change the most at this point and hence, it makes sense that the mean activation value varies at this point. Also, in accordance with learning, we can see that the activation values of other layers change from the end of the second epoch when the top layer does it's maximum learning.

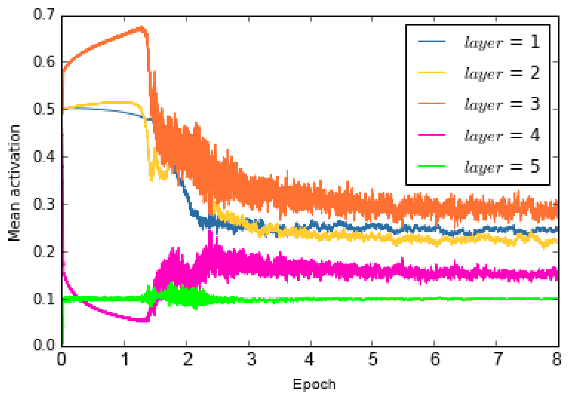

Figure 6: Plot of mean activation function in the y-axis and epochs in the x-axis for a 5 layered neural network with Random Initialization

Back propagated gradient observation with sparse initialization

Sparse initialization is an initialization scheme that was introduced by Martens et al. [7]. This initialization technique restricts the total number of non-zero weight component of each weight vector in every layer. The other components of weight vector are all set to 0. The non-zero components of weight vector are chosen randomly for each neuron. Martens et al applied this technique and got superior results on deep autoencoders. The intuition behind this technique to work well is that a mixture of random non-zero and zero values for each weight vector makes the saturation of the neurons neither too high nor too low. This prevents the neurons from getting saturated. The mixture of zero and non-zero values also promotes greater diversity in activation values for all the neurons.

We used the Sparse initialization technique to train the 5 layered neural network with 5 random components of each weight vector being set to a non-zero value from a standard normal distribution. Rest all components of weight vector and biases were set to 0. We used the same setting as below to observe the back-propagated gradients with Sparse Initialization as in Figure 7.

As seen from Figure 7, we cannot see the vanishing gradient problem in the Neural network gradient with Sparse Initialization. From the figure, we see that the top layer starts learning right from the first epoch as seen by the gradient magnitudes. The learning pattern for the top layer is still the same with it reaching a learning peak in the first epoch. The gradient value for the top layer then falls slowly and reaches a consistent value which is maintained for the rest of the training. As the top layer reaches the learning peak in epoch 1, the other layers including the first layer starts learning right from the first epoch of training in contrast to Random initialization. The bias for bottom layer is still more and decreases as we go to the top layer.

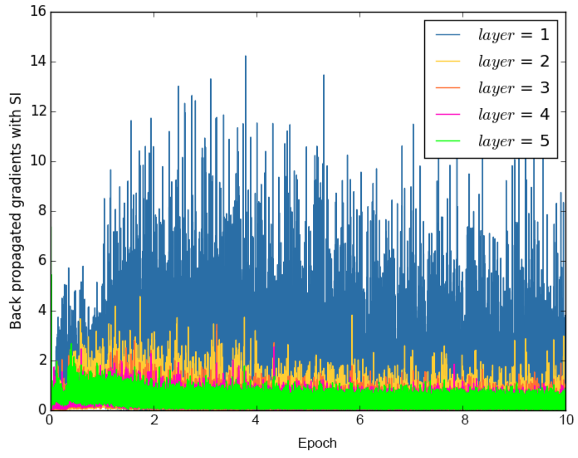

Figure 7: Plot of the back propagated gradient magnitude with respect to weight along y-axis and the update intervals of 10 minibatch along x-axis for the 5 layers of neural network with Sparse Initialization.

Test Accuracy with Sparse Initialization

As with Random Initialization, we also looked at the test accuracy over 10 epochs averaged over 10 runs for the model trained with Sparse Initialization as seen in Table 2. We can see that a good accuracy is reached at the end of first epoch when the top layer reaches it's learning peak. This makes us conclude that Sparse Initialization also places the parameters in a good solution space. After the first epoch, the accuracy however increases slowly and reaches a value of 87% at the end of 10 epochs. This means that the other layers including the bottom layer in this case also tries to fine tune the solution and starts approaching the nearest local minima. However, Random initialization places the parameters in a solution space with better local minima than Sparse Initialization as can be seen from the high test accuracy of Random Initialization at the end of 10 epochs as compared to Sparse Initialization. We also tried setting the limit of Sparse Initialization to 15 instead of 5 and got results similar to Random Initialization with the persisting vanishing gradient problem.

Activation Function analysis for Sparse Initialization

We also tried analyzing the mean activation function of the neurons trained with Sparse Initialization as seen in Figure 8 to see if we can see any significant difference in the activation value patterns for the different layers. The

Epoch	Accuracy(%)
1	78.843
2	84.289
3	85.226
4	86.423
5	86.503
6	87.586
7	87.479
8	87.593
9	87.473
10	87.293

Table 2: Test accuracy with Sparse initialization over 10 epochs

hypothesis was that since the vanishing gradient problem is not seen in Sparse Initialization method, so the top layer will not be saturated in this case. However, we could not see that and the top layer continued to be at the mean activation of 0.1 all throughout the training. Another difference with Random Initialization was that the penultimate layer behaved almost similarly as that of the top layer. Also, the bottom layer showed the maximum variance in Sparse Initialization as opposed to Random Initialization in which Layer 3 showed the maximum variance. This means that in Sparse Initialization, the bottom layer does learning throughout and the weight parameters keeps changing considerably throughout the training. This behavior can be attributed to the fact that in Sparse Initialization as discussed before, the parameters end up in an inferior solution space than Random Initialization and this continuous change of weight can be due to the oscillation of the weights in this basin in order to find a good local minimum.

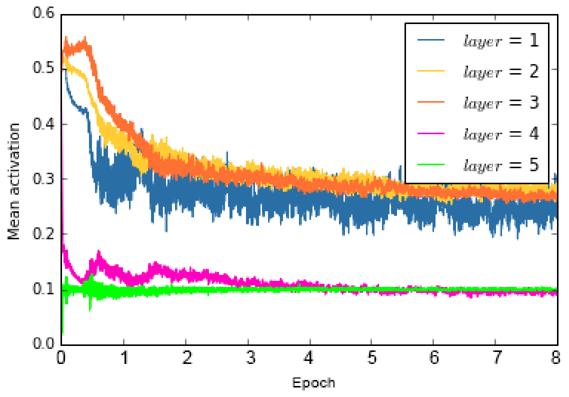

Figure 8: Plot of mean activation function in the y-axis and epochs in the x-axis for a 5 layered neural network with Sparse Initialization

Back propagated gradients with momentum

It is known in the optimization community that momentum accelerates the learning by taking useful steps across directions of low curvature as well as converging in directions of highly turbulent high curvature. We applied the momentum technique to both Random and Sparse Initialization setting to study it's effect on learning in deep Neural network.

For Random Initialization, we applied a momentum coefficient $\mu = 0.5$ and it paced up learning. We can see the back-propagated gradient magnitudes plotted against y-axis and epochs in x-axis in Figure 9. In contrast to Random initialization without momentum, applying momentum helped us to get rid of the vanishing gradient problem by one epoch. Without applying momentum, the vanishing gradient problem was persistent during the first 2 epochs of training. Applying momentum leads to the vanishing gradient problem persisting for the first epochs. As opposed to the learning peak of the top layer at epoch 3 with Random initialization, applying momentum helps the top layer to reach a learning peak at the beginning of second epoch thus accelerating learning. The other gradient patterns remains the same as that of Random Initialization.

We next applied momentum to Sparse Initialization but could not see any significant improvements in learning, so we do not show the plot for the back propagated gradient in this paper. One interesting observation was that a high momentum coefficient as is mostly preferred degraded the performance of Sparse Initialization and made it perform as poorly as Random Initialization. A low momentum coefficient of 0.1 for weight parameter and 0.5 for bias parameter was found to be ideal and accelerated learning in the first epoch as compared to Sparse Initialization without momentum. However just like Sparse Initialization, it did not reach a very high accuracy. From this behavior, we can have an insight that a low momentum coefficient makes progress in high curvature direction in the beginning as suggested by the learning acceleration in the early epochs. However from these high curvature directions, it finds it difficult to make useful progress in low curvature direction and keeps oscillating in these high curvature directions. This is the reason the test accuracy keeps oscillating and does not reach a very high optimal value. This observation is in accordance with the observation by Sutskever et al [11] in their paper.

Comparison of various methods

We did a comparison of the Test accuracy for the four methods we considered above in our experiments. These methods are Random Initialization, Sparse Initialization, Random Initialization with momentum and Sparse Initialization with momentum. All these accuracy have been averaged over 10 runs with 3000 random test data. As seen from the graph in Figure 10, Random Initialization starts with a poor accuracy of around 19% at the beginning. It however ends with the highest final accuracy of around 95% at the end of 10 epoch. Sparse Initialization starts with an initial accuracy of around 79% doing much better than Random initialization in the early phase of training. It however ends at an average accuracy of around 85% which is not as high as that of Random Initialization. Random Initialization with momentum applied gives an initial boost to learning and reaches an average initial accuracy of around 45 % which is better than that without momentum.It also reaches a high test accuracy like that of Random Initialization. Sparse Initialization with momentum applied does not show any significant changes in test accuracy than Sparse Initialization without momentum and both these techniques show the same behavior. From the graph, it seems that Random Initialization with momentum applied does a good balance between initial learning and final high test accuracy.

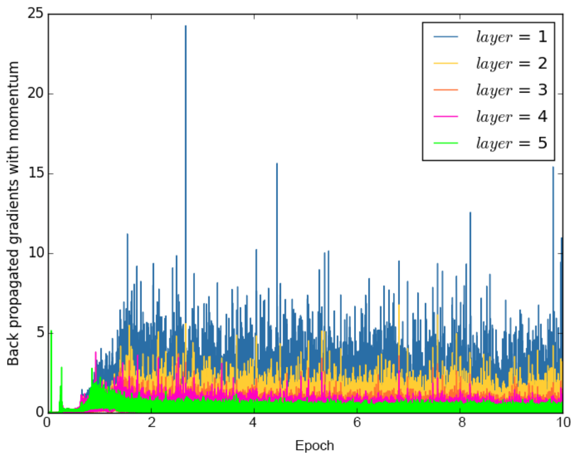

Figure 9: Plot of the back propagated gradient magnitude with respect to weight along y-axis and the update intervals of 10 minibatch along x-axis for the 5 layers of neural network with Random Initialization and momentum applied.

CONCLUSION AND FUTURE WORK

In this paper, we study the gradient and activation function properties of a deep neural network. We conclude that the vanishing gradient problem exists in a deep neural network with Random Initialization during the early training phase and gradually this problem fades away during the later half of the training. We also conclude that it is the top layer which drives the learning and the learning of the bottom layers fine tunes the solution to converge to the nearest minimum. We also saw that Sparse Initialization technique can be used to get rid of the vanishing gradient problem but at the cost of lower accuracy. We also conclude that momentum has a positive effect on learning with Random Initialization and accelerates the learning while reducing the persistence of vanishing gradient problem by an epoch. Overall, we looked deeply into the layer wise learning patterns of a deep neural network.

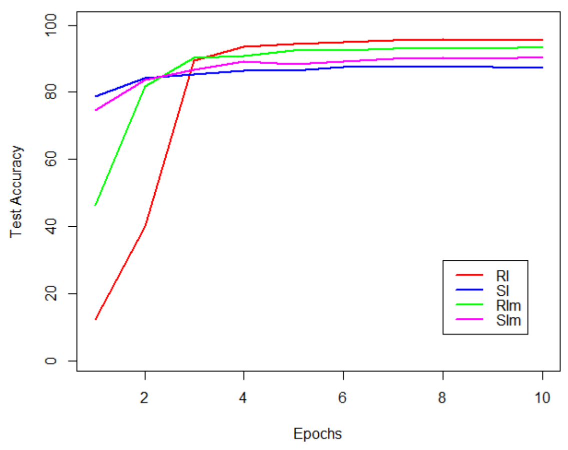

Figure 10: Test Accuracy plotted on the y-axis against number of epochs in x-axis for various methods. RI: Random Initialization, SI: Sparse Initialization, RIm: Random Initialization with momentum, SIM: Sparse Initialization with momentum.

A lot of future work remains in order to make these claims stronger. Our conclusions are based on results from a single dataset. We will have to run our experiments on several other commonly used datasets to make sure that our claims are robust to the choice of data. A stronger analysis of the activation function needs to be done in order to answer the behavior of the gradient patterns seen in the layers. Apart from that, it would be interesting to see how application of different learning rates in the different layers would behave in such a network. Since, the bottom layer fine tunes the solution, so it would make sense to keep the learning rate low for these layers so that they do not surpass the minimum. On the other hand, increasing the learning rate by a reasonable amount for the top layer might accelerate the learning further in the initial training phase and help in getting rid of the vanishing gradient problem. These insights would further increase our understanding of the working of a deep neural network, which would help us to come up with better and efficient optimization technique for leveraging the power of a deep neural network.

REFERENCES

1. Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. 2007. Greedy Layer-Wise Training of Deep Networks. 153–160. `http://www.iro.umontreal.ca/~lisa/pointeurs/BengioNips2006All.pdf`
2. Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on* 5, 2 (1994), 157–166.
3. Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*. 249–256.
4. Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (2006), 504–507.
5. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
6. Yann LeCun, Koray Kavukcuoglu, Clément Farabet, and others. 2010. Convolutional networks and applications in vision.. In *ISCAS*. 253–256.
7. James Martens. 2010. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 735–742.
8. Abdel-rahman Mohamed, George E Dahl, and Geoffrey Hinton. 2012. Acoustic modeling using deep belief networks. *Audio, Speech, and Language Processing, IEEE Transactions on* 20, 1 (2012), 14–22.
9. Michael Nielsen. 2012. Code base for Back propagation. (Aug 2012). `https://github.com/mnielsen/neural-networks-and-deep-learning.git`.
10. Genevieve Beth Orr. 1995. Dynamics and algorithms for stochastic search. (1995).
11. Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*. 1139–1147.