MLP Coursework 2: Investigating the Optimization of Convolutional Networks

s2110911

Abstract

This report investigates optimization issues related with vanishing gradients in deep neural networks. Our investigations are primarily concerned with identifying the causes of vanishing gradients and exploring existing methods in literature to combat it. We present a comparison of a shallow and deep network and show that the deep network completely fails to learn due to near zero gradients throughout the training period. Repeated multiplications and convolutions, in a deep network, driving the error signal to a diminutive value was identified as the probable cause. Batch normalization was chosen to be implemented in our deep network and demonstrated immediate positive results. Gradient flow plots show that weights are being updated and the deep network is training normally. We additionally implement some minor hyperparameter tuning to demonstrate potential for model improvement. Our final deep model overcomes the vanishing gradient issue and outperforms the shallow network. It achieves an accuracy of 50.1% on a test set of the CIFAR-100 dataset.

1. Introduction

The challenge of image classification has led researchers to explore neural networks that are growing in depth. These deeper networks are thought of as having greater abstraction power. In the case of image classification the stacked layers represent different levels of features within the image. The idea is that with more layers the levels of these features can be augmented (He et al., 2016). These multiple hidden layers of abstractions intuitively gives an advantage in solving complex pattern recognition problems (Nielsen, 2018).

However, as one tries to continually grow a network in the hopes of boosting their model accuracy, an opposite effect begins to take place. There will be a point during the progressive stacking of layers where the performance of the model begins to decline. This performance dip is observed in the training data as well as the validation, thereby ruling out overfitting.

If more investigation is done, then we can observe that the issue lies within the optimization of the network. With deeper networks some of the layers weights will struggle to converge or update. A common source of this issue is the unstable gradient problem, where the weights of some layers update at very slow rates or fail to learn completely. As we later investigate, this instability is a fundamental problem for gradient-based learning networks.

In this report, we will compare two convolutional networks in an image classification tasks. The two networks will be of a similar architecture (based off the VGG network) with the only difference being the depth of each network. An investigation is conducted to compare the baseline performance of each network with an analysis into the how the optimization performance varies between the two.

We then proceed to research different methods in literature to overcome this problem. From this, we will observe that even with the use of non-saturating non-linearities such as ReLU activation, a deep network may still suffer optimization related issues. In the following section, we implement one of the suggested methods, batch normalization, within our deep model and a subsequent investigation is carried out to observe any remedying effects. We discover that the vanishing gradient problem is resolved and show that performance improvement is possible through hyperparameter tuning.

2. Identifying training problems of a deep CNN

To illustrate the issue of unstable gradients with deep networks, we conduct an experiment with two networks, one shallow with 7 convolutional layers and 1 fully connected layer and another deeper network with 37 convolutional layers and 1 fully connected layer. Both these networks are as mentioned based on the VGG architecture and will be run on the CIFAR-100 image classification dataset. More details on the model configuration and input data structure will be explained in the Experiments section.

The accuracy results on the training and validation data are presented for both models in figure 1. The numbers are reported at each training epoch. The shallow network will be referred to as VGG-08 while the deeper network, VGG-38. From the comparison we can observe that the shallow VGG-08 recorded a normal score for its accuracy and demonstrated a reasonable improvement while training. The VGG-38 network however, recorded a horizontal line almost immediately after training initiated. The accuracy of the model did not improve at all and remained near 0. This indicates that the network did no learning throughout

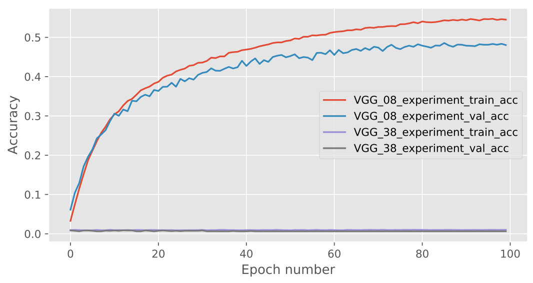


Figure 1. Deep vs shallow network accuracy across training epochs

the 100 epochs. Adding to this even the training accuracy of the deep model does not increase, ruling out overfitting as the source of the issue. These are classic symptoms of an optimization problem and could result from the unstable gradients. To further investigate this, we visualise the average gradient 'flow' of the network at each of its layers. The gradient flow plots are shown in figure 2. The lines are shaded according to their training epoch, i.e. earlier epochs are more faded.

Its important to note that we take the average of the absolute value of the gradients at each layer, since we are more concerned with the magnitude of the gradient rather than its direction. Also, we only visualize layers with learnable parameters since these are the ones that are updated during gradient descent.

The 'healthy' VGG-08 gradient plot illustrates a normal scenario where each layer shows non-zero gradients indicating that some learning is done by the network. The broken VGG-38 network clearly shows major issues with its optimization. The gradients of all layers barring the final linear layer is zero throughout training. This demonstrates why the accuracy and loss plots did not improve at all as the weights of network barely changed from their initialisation. More importantly this figure identifies the issue to be the vanishing gradient problem. These problems are synonymous with deep networks due to the repeated multiplication of the original error signal backwards across the many layers (assuming the gradients are below 1). To further investigate this problem and existing solutions for it we refer to the following literature.

3. Background Literature

3.1. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

The main problem the authors focus on is the changing of the layer input distributions as training progresses. This is especially true for deeper layers in the network where inputs are dependent on the parameters of all the layers before it. Even if a small change in the parameter in one of the earlier layers will amplify the change in input distribution in the deeper layers. This is a problem since with any machine learning model, we assume the input distribution to remain constant during training. The authors argue that is true for the sub-network as well and claim it causes difficulty in optimization. They refer to this problem as an internal covariate shift and their solution is to normalize the data between each layer in the network using batch statistics or batch normalization. By doing so, the deeper layers would not need to readjust to compensate for the change in their input distribution. The reason in using batch statistics instead of global statistics is that gradient descent optimization would need to consider the change in not only the current training example but also the entire training set when a parameter is changed at each layer. And this would be too expensive to perform using the entire training set.

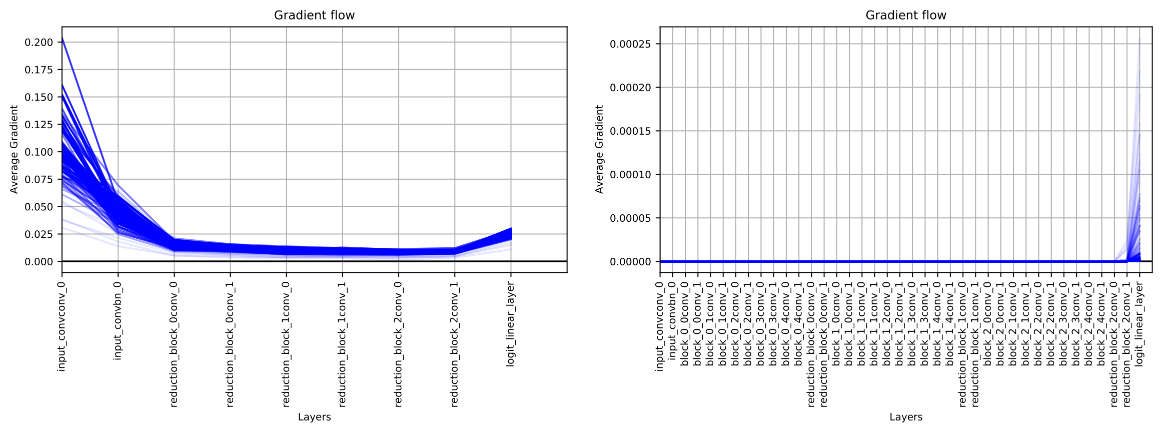
The authors also included shift and scale learnable parameters to ensure the representational power can be restored (if its beneficial for a particular layer to do so). Training with gradient descent is made possible since the BN transforms are differentiable. Issues with vanishing or exploding gradient are also claimed to be resolved as the normalization prevents small changes in parameters to be amplified across deep networks. Their main goal was to improve the convergence especially for models were prone such optimization issues, and from the empirical results they demonstrated this with state of the art results with only a fraction of the training time.

3.2. Deep Residual Learning for Image Recognition

(He et al., 2016) presents a novel solution for deep network optimization issues by reformulating the layers to fit a residual mapping rather than the original desired underlying mapping. They first establish an argument where the performance of a deep layer should be at least as good as its shallower counterpart if the shallow layers are copied over to the deep network and every other layer performs an identity mapping. However, this turns out not to be the case and the deeper network train performance degrades. The authors conjecture that it is actually easier to optimize the residual mapping than the original and that if such deeper layers were to act as identity mappings, then driving the residual layers to have zero weights is easier than actually fitting the identity mapping on the stacked layers.

To perform these residual mappings, the authors create shortcut connections, where output activations skip layers and essentially perform identity mapping when they are added back. In order for the dimensions to agree during the addition at all possible locations in the network, the authors suggest different techniques including, zero padding and projections.

They experimented on two networks, a shallow (18 layer) and a deep (34 layer) network with and without residual architecture. As previously observed the plain deep network experiences a performance degradation relative to the plain shallow network. With skipped connections on the deep network, performance improves and surpasses its shallower counterparts. With this degradation seemingly eliminated, the authors extend the depth of their network and achieved state of the art performance.



(a) VGG-08 Gradient flow plot

(b) VGG-38 Gradient flow plot

Figure 2. Gradient flow plot comparison

3.3. Densely Connected Convolutional Networks

The final paper very much takes inspiration from ResNets and other similar networks where the theme is to create shortcut paths connections. This is line with the objective of eliminating issues related to vanishing gradients in deep networks. Densely Connected Neural Networks or DenseNets takes this to the extreme where every layer takes as input, the feature of maps of all the preceding layers and distributes its output feature map to all subsequent layers. This introduces many more connections that a standard feed forward network.

A major difference between DenseNets and ResNet, is that in implementing these shortcuts, DenseNets use concatenation while ResNets as described previously uses addition. Because of these extended connections, the authors argue that DenseNet accommodates better information flow making them easy to train since each layer has ‘direct access’ to the gradients of the loss function at the very end of the network. The authors also emphasise the compactness of DenseNet, stating that because of this ‘feature reuse’ behaviour, each layer can be made narrow by producing a relatively small number of features. This makes the architecture very parameter efficient. In the results section, the authors demonstrate this by equalling state of the art results with much less parameters.

4. Solution Overview

In this section we will be looking more in depth at the chosen solution, Batch Normalization (BN). This method is chosen since it addresses the problem of vanishing gradient quite well in literature. For example in the paper, it manages to remedy the effects of vanishing gradients even when tested on a network with sigmoid activations, a type of saturating nonlinearity which is known to be prone to vanishing gradients in deep networks. Since this is the main problem we identified in section 2, Batch Normalization will be the tool of choice to address our issue and to aid in convergence time.

As described in section 3, the technique involves normalizing the input to each layer with respect to the other samples in the mini-batch. This normalized value is then adjusted using scaling and shifting parameters that are learned during training which allows the network to control these parameters to allow for faster convergence. Note that without BN, these parameters are allowed to have arbitrary values too, however these parameters are not managed but instead is affected by the cascade of parameters of the preceding layer. The BN transformation operation is presented below:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{M} x_i \qquad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{M}(x_i-\mu_B)^2$$

$$\hat{x}_i = \frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\epsilon}} \qquad y_i = \gamma\hat{x}_i + \beta$$

The parameters are calculated for each training sample i in the minibatch B where i ranges from 1 to M. The learnable shift and scale parameters are denoted by γ and β respectively. The output of the transformation y_i replaces x_i as the input in the subsequent layer. We refer to this transformation as $transformation_{BN}$.The pseudocode algorithm for implementing batch norm in our networks are presented in Algorithm 1.

Its important to note that the transformation is slightly different during the evaluation phase. Instead of using the minibatch statistics for normalization we keep track of the mini-batch mean and variance and take the expectation of them for the population statistics. This is the original implementation in the paper and is the default implementation in Batch normalization module in Pytorch (our solver). The normalization procedure only takes place before a nonlinearity layer (after an affine or convolution layer). This is also in concurrence with the original paper, and is designed this way as the normalization after a non-linearity is unlikely to produce a stable distribution (Ioffe & Szegedy, 2015).

Algorithm 1 Batch Normalization

```
**Training Phase**
**Input:** The activations, $x^k$, at each layer k in the network where $k = 1..K$.
**for** $k = 1$ **to** $K$ **do**
  **if** $k + 1$ is a non-linearity **then**
    Swap output activation $x^k$ with $y^k$ using $transformation_{BN}$
    Store $\mu_B$ and $\sigma_B^2$
  **end if**
**end for**
**Testing Phase**
$E[x] = E_B[\mu_B]$
$Var[x] = \frac{m}{m-1} E_B[\sigma_B^2]$
**for** $k = 1$ **to** $K$ **do**
  **if** $k + 1$ is a non-linearity **then**
    Swap output activation $x^k$ with $y^k$ using $transformation_{BN}$ but:
    Replace $\mu_B$ with $E[x]$
    Replace $\sigma_B^2$ with $Var[x]$
  **end if**
**end for**
```

5. Experiments

5.1. The Input Data

In our image classification experiments, we use the CIFAR-100 dataset. This is a dataset of 100 classes, with 600 32x32 images for each class. 500 of the images are part of the training set (for training and validation), while the remaining 100 is used for testing.

Augmentation transforms of the training data is performed using random crops and horizontal flips, to improve robustness against overfitting. Both the training and test data are normalized as a preprocessing step.

5.2. General Network Architecture

This section will discuss in detail the general network architecture for all following experiments. As mentioned these networks are based of the VGG architecture used in (Simonyan & Zisserman, 2014).

We break the overall network into different blocks. There are **two main block types**, a processing block and a dimensionality reduction block. The processing block is composed of 2 convolutional layers each with kernel size of 3x3 and produces 32 filters. A padding of 1 is applied to each input feature map to ensure the preservation of spatial resolution. Each convolutional layer is followed by the Leaky ReLU nonlinearity.

The second, dimensionality reduction block is mostly the same architecture with a pooling layer in between the convolutional + non-linearity layers. A reduction factor of 2 is set, halving the spatial dimension of the feature maps.

Blocks are grouped together in different stages, where each stage is composed of a specified number of processing blocks and is capped with a single dimensionality reduction block. The number of stages will be defined for each experiment. Each network starts with a special entry block which consist of a convolutional layer followed by a Batch Norm layer and Leaky ReLU nonlinearity. Each network also ends with a global average pooling layer which compresses each feature map into a single pixel before feeding this as inputs into a fully connected linear layer which outputs the 100 classes. A cross-entropy loss function is used and the Adam optimizer is employed (Kingma & Ba, 2014). On top of this we use a cosine annealing learning rate scheduler with the initial learning rate set at 0.001 and a cycle period of 100 epochs. The training period is 100 epochs. This is the general network architecture that will be implemented across all out experiments. Any modifications will be mentioned explicitly. The result metrics (seed 0) for all experiments is displayed in table 1. Note that, each experiment is run under 3 different seeds (0,1,2) with validation accuracy results in figure 3 to show variability under different initialisation.

5.3. Problem Identifying Experiments

The two networks explored here, VGG-08 and VGG-38, are those that were mentioned in section 2. In our VGG-08 network, the number of stages is set to 3 while the number of blocks per stage is set to 0 i.e. each stage only consists of the dimensionality reduction block. For our deeper VGG-38 network, the number of stages is maintained at 3, but now there are 5 processing blocks per stage (plus the dimensionality reduction block).

5.4. Batch Normalization Implementation

As an attempt to fix the broken VGG-38 network, we implement the chosen solution, Batch Normalization on the this deep network. The implementation procedure is as described in Algorithm 1, where a batch norm layer is inserted prior to nonlinearity layers in the network. This batch normalized deep model is referred to as BN-VGG-38. The results are compared with the 2 original networks in table 1.

We can observe that the modified deep model (BN-VGG-38) actually learns now and slightly outperforms the shallower VGG-08. We discuss more about this result in the following section. However, from a general observation, our maximum training accuracy thus far is only at around

Model	TRAIN_ACC	TRAIN_LOSS	VAL_ACC	VAL_LOSS
Base Models				
VGG-08	0.545	1.617	0.480	1.935
VGG-38	0.010	4.605	0.006	4.607
BN-VGG-38	0.553	1.577	0.480	1.976
Fine-tuned Models				
HIGHLR-BN-38	0.510	1.768	0.448	2.026
WR50-HIGHLR-BN-38	0.575	1.480	0.502	1.843
MBS50-BN-38	0.511	1.763	0.480	1.940

Table 1. Comparison of performance metrics at 100 epochs

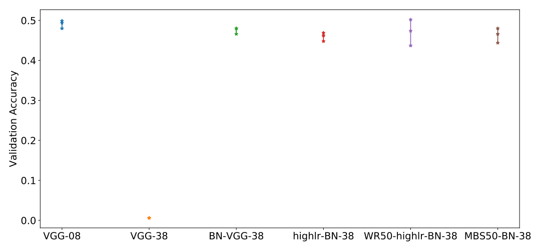


Figure 3. Validation Accuracy for 3 seeded runs for each model

50% which seems too suggest that our models are underfitting. At this point, we try to further modify this BN model by implementing some of the recommended settings from the original paper (Ioffe & Szegedy, 2015). The first of which is the afforded larger learning rate. The original authors experimented with a high learning rate of 0.0075. We increase our learning rate from 0.001 to this value. A comparison of model accuracy between this high learning rate model (highlr-BN-38) and BN-VGG-38 is illustrated in figure 4.

Initial observations show that the high learning rate model is more unstable with accuracy spikes/dips (an expected feature of the higher learning rate). The model also appears to plateau earlier with both training/validation curves flattening near the 60th epoch.

This could represent a case where the optimization is stuck in a local minimum, as a result of the boosted learning rate. To fix this, we implement the warm restart method as described in (Loshchilov & Hutter, 2016). The method was chosen since it has a potential to aid the optimization in escaping local minima. We replace the vanilla cosine annealing learning schedule with its warm restart variant and set the restart at epoch 50. The experiment is denoted as WR50-highlr-BN-38 and its performance is displayed in figure 5. The effects of the warm restart is clearly apparent with the sudden dip at epoch 50. However, following this, the network improves rapidly and eventually obtains a higher accuracy than BN-VGG-38 at the end of the training epochs.

In our final experiment, we attempt to utilise batch norm's regularization properties to reduce the visible overfitting in WR50-highlr-BN-38. This was one of the side contributions of batch norms as expressed in (Ioffe & Szegedy, 2015). To further induce this property we reduce the minibatch size from 100 to 50. All other settings from WR50-highlr-BN-38 are maintained, and this new model is denoted as MBS50-BN-38 (shortened name for clarity). The accuracy/loss metrics between this model and the WR50-highlr-BN-38 model are compared in table 1. A comparison between the train and validation accuracy of this model shows that the generalization gap has improved from above 7% to now about 3%. However in doing so, the both the train and validation accuracy are smaller than in WR50-highlr-BN-38. Therefore, we take this WR50-highlr-BN-38 as our final model and perform a test run to achieve an accuracy of 50.1% and a loss of 1.859.

6. Discussion

Our sets of experiment begin with the comparison of the baseline shallow, VGG-08, and deep, VGG-38, models. From our analysis in section 2, we identified the issue with broken deep network being the result of a optimization issue relating to a vanishing gradient problem. This was primarily based on our observations in figure 2. This problem is prone to occur in deep networks with gradient descent based optimizations since the original error signal has a to backpropagate across more layers where repeated multiplications and convolutions can drive the signal down to a diminutive value such that any updates on the weight is rendered insignificant (Kolbusz et al., 2017). The weights remain stationary at their original initialisation and as a result the model accuracy does not improve with training.

To remedy this issue, we implemented batch normalization in our deep network. The authors of the paper assert that by normalizing the inputs of each layer, the presence of internal covariate shift in deep networks is minimized. Each layer now has a more stable input distribution during training meaning the latter layers would not have to compensate for repeated changes in distribution caused by parameter updates of the preceding layers. One can think of batch normalization as a form of whitening of the layer inputs. This consequently results in a more 'circular' loss landscape making it easier for gradient descent optimization to traverse to the minimum rather than get stuck in a local minimum.

All of this could potentially alleviate the issues resulting in the vanishing gradient descent problem. To test this, we implement batch normalization on our deep model. We immediately investigate gradient flow plot for the batch normalized deep model, BN-VGG-38, in figure 6. Observe that the plot is very much different that the original deep

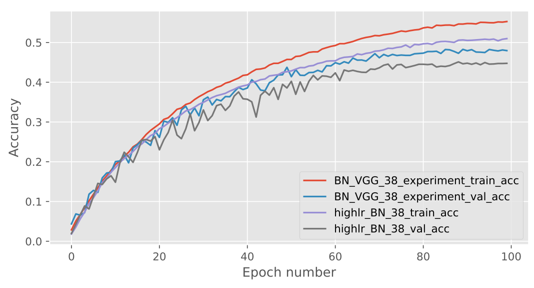


Figure 4. Model accuracy with learning rate 0.0075 vs 0.001

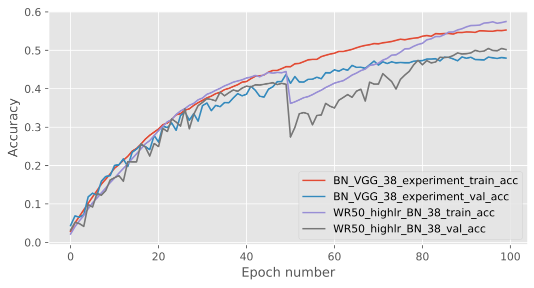


Figure 5. Accuracy of WR50-highlr-BN-38 vs BN-VGG-38

network, in that the gradients are non-zero for all the layers, indicating that the weights are getting updated during training. This plot no longer resembles that of a network with vanishing gradient descent issues. Another interesting observation is the presence of spikes which only occur at the batch normalization layers. If we refer to table 1, we can also see that the network's accuracy is no longer stagnant near zero and actually surpasses (slightly) the performance of the shallow VGG-08 model.

At this point, our objective of overcoming the vanishing gradient problem has been satisfied. However, based on our prior assumptions, a deeper network, rid of optimization issues, should outperform a shallower network. We investigate whether minor hyperparameter tuning can upgrade the deep network's relative performance.

Our first attempt involves increasing the learning rate to 0.0075. This was inspired by the argument from the original paper (Ioffe & Szegedy, 2015), where batch normalization makes a network less susceptible to optimization issues and tolerate a higher learning rate to reach convergence faster (avoiding the need to extend training beyond 100 epochs). However, in doing so we observe a performance degradation exhibited in figure 4. The model plateaus noticeably earlier at a lower accuracy. We conjecture that this is due to the optimization being stuck in a local minimum. Its also important to note that (Ioffe & Szegedy, 2015) implemented the higher learning rate with a similarly aggressive exponential learning rate decay. In order to replicate this higher decay and attempt to escape the local minima, we employ a warm restart feature in our learning schedule. The sudden spikes in learning exhibited during a restart event, may aid the optimization escape the local minima and end up in a more stable configuration in the loss landscape. By introducing restarts we are also increasing the learning rate decay of the cosine annealing function. The restart epoch was set at 50 to allow enough training time for the optimization to recover.From figure 5 we can observe that restart achieved the intended outcome. The sudden dip represented the point of the restart and after training has finished the model achieved a much improved result compared to highlr-BN-38.

Finally, an attempt was made to reduce the generalization error of the network by utilizing batch normalization's regularization properties. Batch norm's main function is not regularization, and in truth there are probably better regularization methods. However, since these experiments are focused on batch normalization, we attempt to make full use of its benefits and try to introduce some regularization by decreasing the minibatch size. The intuition, based on the works of (Ioffe & Szegedy, 2015) and (Luo et al., 2018), is that batch norm helps to introduce noise when using the batch statistics (which depends on the batch) to normalize the data (instead of global statistics). And by using smaller batches we deviate further away from the global statistics and thus introduce more noise during normalization. To test this, we reduce our batch size in half to 50. Table 1 shows the result of doing so with the model MBS50-BN-38. As expected, the gap between the train and validation data decreased, indicating that the model is now better at generalization. However, in comparison with the larger batch sized model, both training and validation metrics are lower. A possible reason for this is that the original model, WR50-highlr-BN-38, was not severely overfitted and by introducing a regularization feature, the effect was to strong in that it resulted with the model underfitting instead. We conclude this investigation by taking WR50-highlr-BN-38 as our final model on the basis that it has overcome this issue of vanishing gradients and through some brief fine-tuning, outperformed the shallow VGG-08 by a more noticeable margin. A test classification measurement is performed and the model records an accuracy of 50.1%.

7. Conclusions

In this report we investigated optimization issues with deep networks related to the vanishing gradient phenomenon. We presented an example of a broken deep network that showed strong indications of vanishing gradient problems and compared to it a normally functioning shallow network. A brief literature review on methods to overcome optimization issues with deep networks was conducted. Batch normalization was chosen to be implemented on our broken deep network and immediately demonstrated positive results. Gradient flow plots indicated that the vanishing gradient problems were eliminated and the model was learning during training. Our investigation was continued to a brief phase of hyper-parameter tuning, where the learning rate, learning schedule and batch size were modified to tune model performance. Our final model consisted of batch normalized deep network, with a high initial learning rate and a warm restart during training. Future works could involve the exploration of other methods mentioned in the background literature section. For instance, combining batch norm with residual learning has been shown to deliver great results (He et al., 2016). Integration of mainstream regularization methods, such as weight penalties and dropout could also be explored to achieve better generalization results and allow for deeper models that can further utilise the optimization improvements.

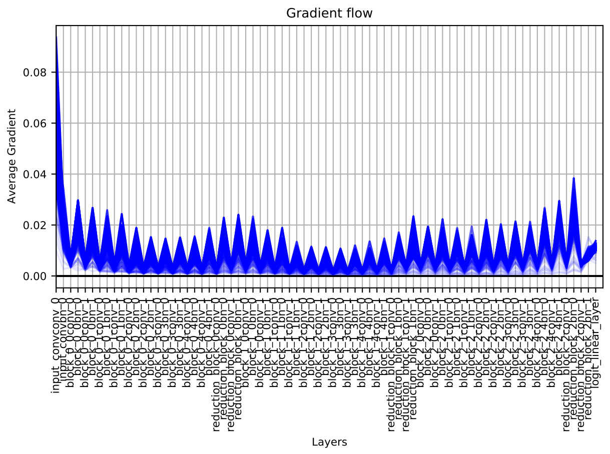


Figure 6. Gradiet flow plot for BN-VGG-38

References

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Kingma, Diederik P and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kolbusz, Janusz, Rozycki, Pawel, and Wilamowski, Bogdan M. The study of architecture mlp with linear neurons in order to eliminate the "vanishing gradient" problem. In *International Conference on Artificial Intelligence and Soft Computing*, pp. 97–106. Springer, 2017.

Loshchilov, Ilya and Hutter, Frank. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Luo, Ping, Wang, Xinjiang, Shao, Wenqi, and Peng, Zhanglin. Towards understanding regularization in batch normalization. *arXiv preprint arXiv:1809.00846*, 2018.

Nielsen, Michael A. Neural networks and deep learning, 2018. URL http://neuralnetworksanddeeplearning.com/.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.