Accelerated Deep Neural network training using lightweight scaling activations

Abdul Hannan Kanji
University of Massachusetts, Amherst
akanji@cs.umass.edu

Abstract

We know from [4] that the internal covariate shift(ICS) is a serious problem for learning deep neural network efficiently. As a solution, [4] introduced a separate layer called the Batch Normalization layer to overcome ***ICS****. This project tries to overcome this problem using an alternative approach from Batch Normalization where there is a single activation function which also handles normalization with an equivalent or better accuracy and at the same time, reducing the number of parameters in the model.*

1. Introduction

In recent times, the amount of computation power at hand has been increasing at an exponential rate. With this scale of computational abilities, we are able to train much deeper neural networks and unearthed fascinating results which were not able to be easily materialized earlier. But with the increase in computation, we started training much deeper neural networks with double digit number of layers. And with more depth, came much more problems like vanishing or exploding gradients.

One of the biggest problems comes from the basic fact that the input data itself is so unpredictable that the internal distribution is stochastic. That is, the distribution of the unprocessed inputs does not follow any statistic and can change arbitrarily as training proceeds. This is known as *Internal Covariate Shift*. With the changing statistics of the input, the model parameters also try to change accordingly. So [4] introduced another layer after the activation where it fixes the mean and the variance of the layer inputs. What this project proposes is that though [4] uses 2 learned parameters to normalize the layers, an equivalent, if not better, performance can be achieved by just not learning any extra parameters but just using one scaling parameter *before* an activation.

We know that removing *ICS* is ideal for attaining high rates of learning and optimal convergence during training. This is done using a *Batch Normalization* layer after every activation which performs the computation:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}.$$

Hence, it basically learns 2 parameters: $\beta^{(k)}$ and $\gamma^{(k)}$ for every input activation. This means it would be performing gradient calculations for each of these parameters including the mean and variance. This is very expensive and would also be an overhead in the training phase, if there was a way to learn just one parameter to do the same. Plus, as [13] posits, we still don't know what the real effectiveness of Batch Normalization. [13] also questions whether Batch Normalization reduce *ICS at all*?

Based on [13], Batch Normalization does indeed smoothens the landscape of the optimization objective **significantly**. In this project, we also explore main question posed in [13]: *Is the smoothening effect a unique feature of Batch Normalization, or there other less computationally intensive alternatives?*.
Though there are other variations of the same [1] [14] [12], they still require a mini batch for calculation of their moments and also learn 2 parameters.

In the solution for normalization specified here, we use the concept of a running standard deviation parameter based on a momentum variable. Lets call this as the scaling parameter α. If the input is always normalized with 0 mean and unit variance, Batch Normalization can be simplified further to just scale instead of even shifting. This project gives empirical evidence that this is the case and for bigger datasets, it even, at times, trains faster and gives the same, if not better precision.

2. Background/Related Work

Since Batch Normalization was introduced in [4], it has now become a de-facto standard for speeding up training and it does indeed do so. Alongside this, a higher learning rate can also be used. But its also very computationally intensive and also does not work well with small batches

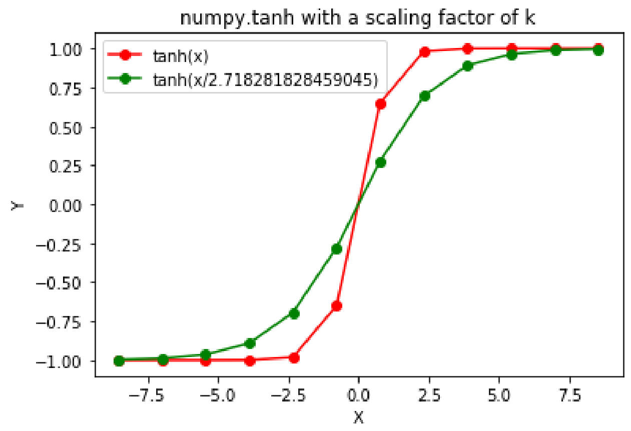

Figure 1. tanh activation function scaling. Here $tanh(x)$ is compared against $tanh(x/e)$

or individual training examples (*on-line learning*). This is expected as Batch normalization tries to estimate the mean and variance using mini-batches of inputs [4].

There are many variants of Batch Normalization that overcome certain limitations like Layer Normalization [1], Weight Normalization [12] and Group Normalization [14]. These are better in ways that they do not require a mini-batch to work on. Moreover, these normalization techniques work well with on-line learning, but fail to perform as well as Batch Normalization. Also, all these techniques do the scaling as well as the shift of the variables. There has been enough evidence and work on normalization techniques after Batch Normalization was proposed [1] [14] [12]. Indeed, this is an ongoing research topic that is actively worked on.

For the exploration, we checked some existing implementations of just having a self normalizing activation layer. There is a some work done on it by [6]. We see that in [6], the inputs to the activation layer itself is normalized using a Scaled Exponential Linear Unit (SELUs). If it could be done for ReLUs, this can definitely be extended to other activations as well. Hence, we set out to explore if this could be done as part of this project. For the implementation, we took the 2 main types of activations, namely, ReLU and Tanh.

3. Approach

For the initial approach, the tanh activation is taken and seen if a scaling could be done on this to avoid saturation. Figure 1 is an example of the effect of scaling on tanh.

Based on the standard deviation, we can scale such that the inputs do not saturate the activation. Moreover, this standard deviation is calculated using a momentum to avoid drastic changes in the scaling parameter with outliers in the data.

Previously, as specified in the project milestone, we did not include this variable in the computation graph and hence got some very unexpected spikes during training. This was because of the way the parameter α was treated in the Scale layer. After fixing this, we saw a similar curve as in the case with batch norm layers.

In another tangent, this idea also draws certain analogy from the concept of neural plasticity [9] where neurons learn using Hebbian plastic connections [11] where the new activation also depends on its past values. Though this used in a different setting altogether, the concept of a momentum seems to be a good mathematical analogy of inducing plasticity.

So as part of this experiment, a scaled $tanh$ function was used:

$$f(x) = \tanh \frac{x}{\alpha_t}$$

$$\alpha_t = \gamma\alpha_{t-1} + (1-\gamma)std(x_t)$$

Here, γ is the momentum which we set as 0.9.Also, α is initialized by default at 1.0. Grid search to find the best γ is left for future work. As of now, we are just exploring if taking only the standard deviation as part of the scaling will give sufficiently good results.

For this, a PyTorch custom Module for the Scaled layer was implemented. This layer was used in a moderate as well as a dense network. The moderate network had 2 Conv2d layers followed by 4 Linear layers. The baseline model had just tanh non-linearity with Batch Normalization using learned parameters.

Another note on the Scale Layer, we have made this layer generic to take any other activation function as a parameter and all that the layer would do is scale based on the running standard deviation.

The idea of a running standard deviation, though unconventional, works here because we have assumed that the input has been normalized. The greatest use of this running scale is that if there is one training example that with high variance, it does not This was compared to the exact same architecture with the Scaled tanh function instead of tanh + Batch Normalization layers.

3.1. Initialization

The model parameter initialization turns out to be crucial to improve the efficacy of model training. The linear layers were initialized using a normal distribution with $0, \frac{1}{\sqrt{n}}$ moments and the convolution layers were initialized using the same distribution with a standard deviation of $2/\sqrt{N}$. This is similar to the initialization technique specified in [2]. This also ensures to a certain level that the initial variance in one layer does not have an adverse effect on the activations in the proceeding layers.

4. Experiment

For the experiment, let us call the network with batch normalization layers as the *control* and the network with the Scale Layer as the *test*. The experimentation was done in 2 phases mainly to evaluate the Scale Layer at multiple magnitudes of the network density with different types of layers.

The depth of the networks for the 2 phases was selected such that it tests the efficacy of the control and test judiciously. The next 2 subsections elucidate the 2 phases of the experimentation.

The metric used in evaluation of the first phase is test error percentage whereas for the second phase, we use precision. K-factor accuracy measurement is scoped for future work.

4.1. Phase 1

In this phase, a small neural network was used with 2 Convolution layers, each followed by the activation along with a Fractional Maxpool layer [3]. This is followed by 4 Linear layers, each followed by the activation. We used a standard mini-batch size of 64 and training was done using Adam optimizer [5] with a learning rate of $3e-4$ using the Cross Entropy loss as the optimization criterion. The control and test models were trained for 20 epochs each.

Intentionally, not a lot of thought was put into tuning and finding the model configuration that would best work for CIFAR-10 [7] at this scale. This is because all we wanted to do was a relative comparison of the 2 models. Nonetheless, we did choose a network that performed sufficiently well to be considered as a good candidate for the experiment.
We further break down this section into the 2 parts for each activation that was evaluated: ReLU and $\tanh$.

4.1.1 Tanh activation

Tanh has emerged as one of the most widely used activation functions in neural networks [8]. It is mostly in place of a sigmoidal function, where stronger gradients are required, as well as in recurrent networks like RNNs and LSTMs which are usually highly susceptible to exploding/ vanishing gradients.

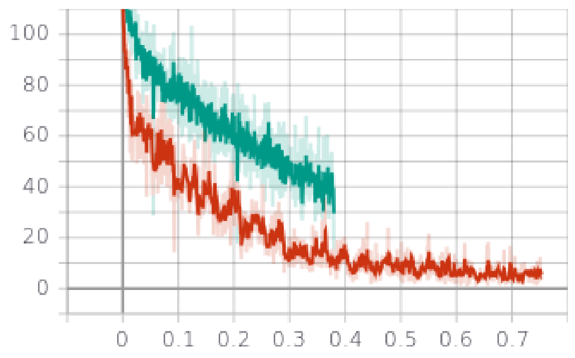

Figure 2. Loss for the small network as training progresses using ReLU. Legend: **control - test**

Using the *Tanh* activation, the control network had a Batch Normalization layer before every *Tanh* activation whereas the test network had a Scaling Layer with a $\tanh$ activation. The results are shown in Figure 3. As you can see, the Scaling is not that helpful for a small network like the one used here.

Also, the α values are adjusted drastically to the initial standard deviation and then it progresses smoothly. A similar behaviour is seen for almost all cases which indicates that if the number of training iterations is large enough, the initial setting of this parameter does not matter as much [4] [6].

You can see from the graph in Figure 3, the loss progression for the test model diverges after around 10 epochs(0.4 on the x-axis). But this is due to the changes in scaling. When both the models were evaluated on the test, these are the results -

Model	Test Error Rate
Control	30.59%
Test	32.58%

4.1.2 ReLU activation

ReLU stands for Rectified Linear Unit and is used as an activation function [10] to induce non-linearity in the model. This activation function is extensively used in numerous models that give really good results [10]. This almost universal acceptance is partly because it is not computationally heavy to perform [10]. The activation function basically

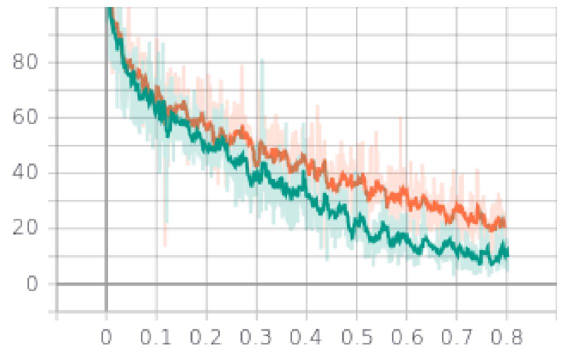

Figure 3. Loss for the small network as training progresses using tanh. Legend: **control** - **test**

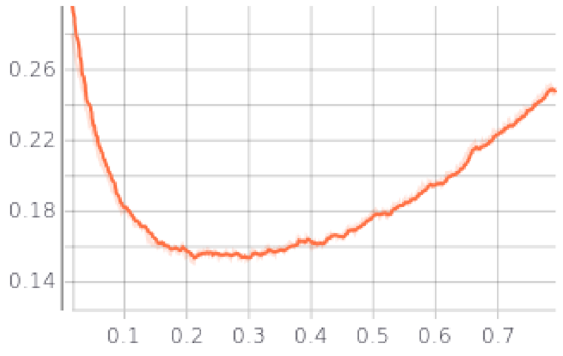

Figure 4. α for the first layer of the small network using the Scale Layer as training progresses using tanh and Scale Layer.

performs this computation on the inputs:

$$f(x) = max(0, x)$$

For ReLU, we used the same setting as in for the Tanh activation section, and replaced tanh layers with ReLU and also used ReLU in the Scale Layer.

The results are shown in Figure 2. Figure 6 shows the progression of the value of α.

The α progression is similar to what we saw with the tanh figure, but it is a little more noisy, it is still unknown as to why it is the case. But what's more interesting is in Figure 2. Recall that both the control and the test are run for 20 epochs only. Figure 2 shows that the training for the test model happens much faster that the control. Though the loss values are not completely indicative of the quality of the learned model after training, the test accuracy can give a more clear picture. Here is the table with the results:

Model	Test Error
Control	29.1%
Test	31.2%

4.1.3 Conclusion for the small models

As you can see from this, though the loss value progression does not give a good idea of the efficacy of the learned model, the test error seems to indicate that both the models have comparable accuracy.

Considering the fact that the test model trains much faster than the control, we can say that these two have the same capacity. Also, for large training epochs, this difference is much more noticeable.

An advantage of the scale layer compared to Batch Normalization layer is that the scale layer does not requires as many learned parameters as Batch Norm. This means, with the correct setting, faster convergence is very likely compared to Batch Normalization.

4.2. Phase 2

Now that we ran experiments on a small network, the next step is to test and see how the Scale Layer performs on a much bigger and popular model. Hence, we chose the VGG-16 model [15] and train it on the CIFAR-10 [7] dataset.

The training was again performed using an Adam optimizer [5] with a learning rate of $3e-4$ and a weight decay parameter of $5e-3$. This time, a batch size of 128 was used and the training data was again normalized the same way, with the cross entropy loss as the optimization criterion.

The control model here is the vanilla VGG-16 model with Batch normalization layer before every activation. The test model had the Scale Layer instead of the Batch Norm layer. Everything else was set to be the same for the test and control. We again tested this on the tanh and ReLU layers.

4.2.1 Tanh activation

Using the *Tanh* activation, the control network had a Batch Normalization layer before every *Tanh* activation whereas the test network had a Scaling Layer with a tanh activation. The results are shown in Figure 10.

As you can see from the graph, the loss progresses pretty much similarly for both the test and control. One observation was that the test model took slightly lesser time to finish 20 epochs of training than the control. You can also see from Figure 7 the progression of the value of α is

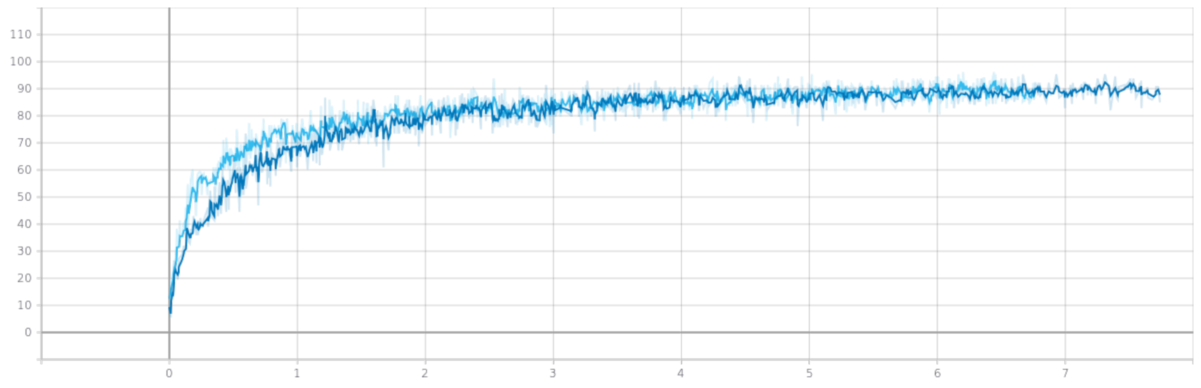

Figure 5. Top-1 precision for the VGG network as training progresses using ReLU. Legend: **control - test**

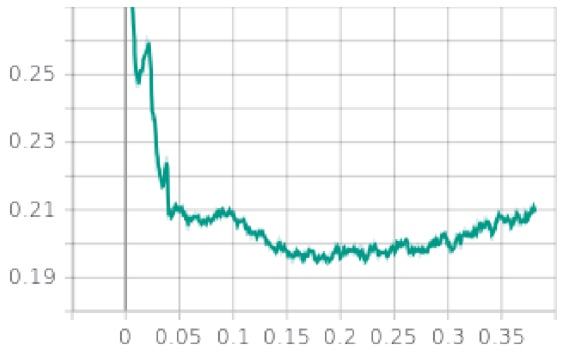

Figure 6. α for the first layer of the small network using the Scale Layer as training progresses using ReLU and Scale Layer.

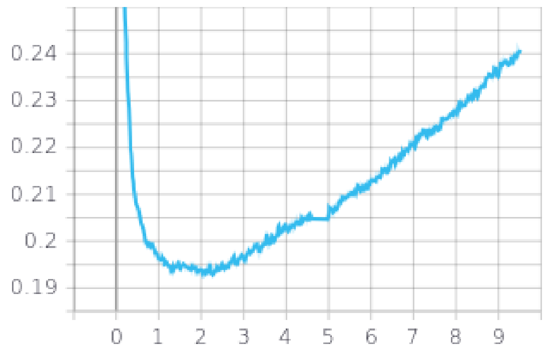

Figure 7. α for the first layer of the VGG network as training progresses using $\tanh$ and Scale Layer.

similar to what we have seen for the model in Phase 1.

The performance of the models is also almost similar as shown here:

Model	Precision	Training Time
Control	84.72%	9h 40m
Test	85.03%	9h 30m

From the data above, we can see that the models are almost equivalent. Without any degradation in test accuracy, we can effectively learn a model with lesser model parameters and slightly faster, too.

4.2.2 ReLU activation

Similar to the $\tanh$ activation procedure, we instead use ReLU layers. Figure 11 shows the loss as training progresses. Though the figure here is not very clear, but the test model here took more time, this behavior has to be explored more and is interesting, but nevertheless, scoped beyond this project due to time constraints.

Also, Figure 9 shows how the value of α changes. One common pattern we see is that sudden decrease in the beginning of training and increases slowly. Checking the behaviour of α for higher epochs is scoped for the future as it would need significantly more time and compute resources.

The models evaluates almost equivalently here too-

Model	Precision	Training Time
Control	86.80%	6h 45m
Test	87.83%	7h 02m

4.2.3 Conclusion for Phase 2

After training the VGG model on CIFAR-10, it is clear that Batch Normalization is an overhead in many cases (atleast in this case mentioned here) and simple tricks like scaling the activation would suffice for accelerating training and convergence [4]. While training with such layers, it is evident that in the early epochs, learning using batch norm layers is more effective than any other normalization methods like LayerNorm, WeightNorm or GroupNorm, but as

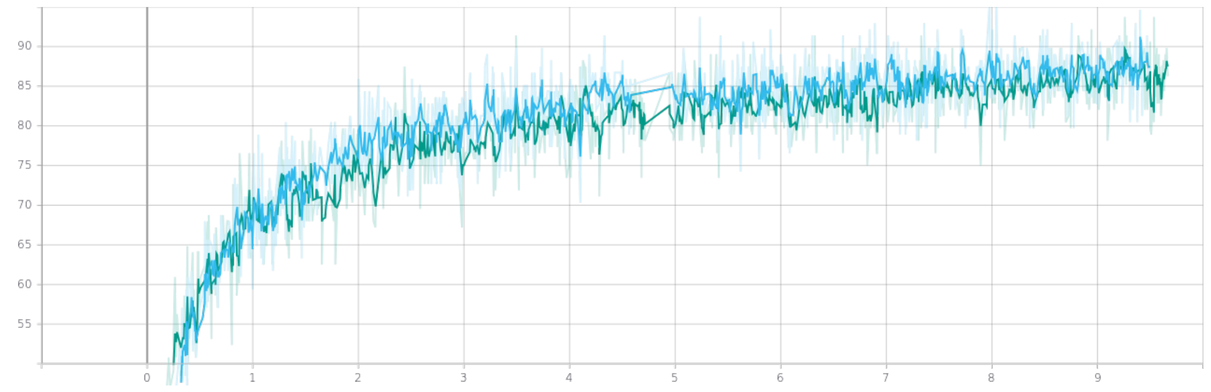

Figure 8. Top-1 precision for the VGG network as training progresses using Tanh. Legend: **control** - **test**

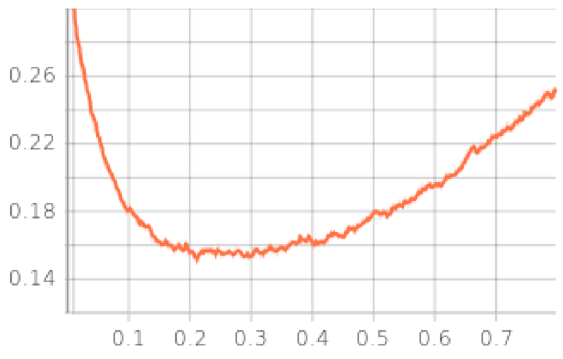

Figure 9. α for the first layer of the VGG network as training progresses using ReLU and Scale Layer.

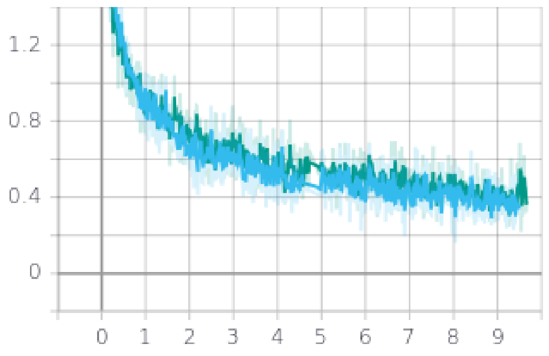

Figure 10. Loss for the VGG network as training progresses using $\tanh$. Legend: **control** - **test**

learning proceeds, and when the scale increases, we see the difference between the various normalization methods.

5. Future work

There is a lot of scope for this work ahead. Here are some thoughts:-

- Explore other configurations of the momentum parameter. This was intentionally omitted after doing a basic grid-search on a small model.
- Try other model configurations which are deeper than VGG and also try different initial input layer normalization configurations and see which works the best.
- Explore the effect of the Scale Layer in Recurrent networks.

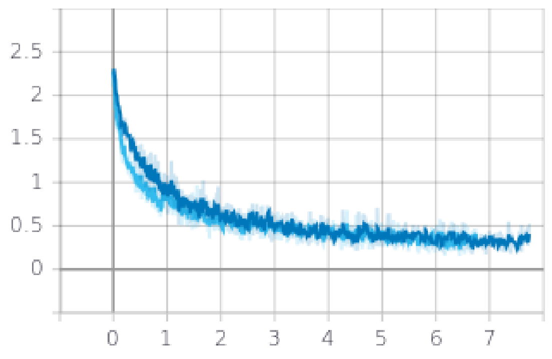

Figure 11. Loss for the VGG network as training progresses using *ReLU*. Legend: **control** - **test**

6. Conclusion

The complexity involved in training neural networks increases exponentially as the number of model parameters and/or the number of epochs increase. This was encountered many times in the second phase of the project. The amount of time required to run a single pass was a lot, but alongside, we also learnt to stop early in case we see a pattern that is unlikely to learn or indicates that something is wrong. For example, when the variance in the loss values as training progresses changes (an increase or decrease) drastically, we know that something is wrong so we should stop early and check what is wrong.

Another key observation was that one particular concept may not necessarily work perfectly at every scale or configuration of the model. For example, in this case, the Scaled Layer did not work very well for small networks and epochs with relatively shallow configuration of the model.

Agreed that in many cases, this would be a terrible idea and other techniques like Layer Norm [1], Weight Norm [12] and Group Norm [14] would work better and they also do not have the requirement of working in mini-batches. But the selection of normalization technique is highly subjective to the underlying model selection that you do. This project was originally intended to just explore a lightweight alternative to Batch Norm while also being able to work in an *on-line* training setting and I hope this method, along with the empirical evidence cited, put forth a feasible alternative based on the use case for normalization.

References

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016.

[2] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[3] B. Graham. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014.

[4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[5] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks, 2017.

[7] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`, 2009.

[8] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[9] T. Miconi, J. Clune, and K. O. Stanley. Differentiable plasticity: training plastic neural networks with backpropagation. *arXiv preprint arXiv:1804.02464*, 2018.

[10] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[11] T. A. Polk and M. J. Farah. Brain localization for arbitrary stimulus categories: A simple account based on hebbian learning. *Proceedings of the National Academy of Sciences*, 92(26):12370–12373, 1995.

[12] T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

[13] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization?, 2018.

[14] Y. Wu and K. He. Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

[15] X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):1943–1955, 2015.