An Exploration of Neural Network Parameter Initializations

John McKeown

***Abstract*—A lot of work has been put into every step of modern deep learning pipelines. Convolutional nets have revolutionized image processing. Adaptive moment estimation (ADAM) represents significant advancements into efficient gradient based training. Rectified linear units (ReLU), residual networks (ResNet), and batch normalization represent large improvements in effective backpropagation of gradients. Despite this, little research has been conducted concerning weight initialization. Xavier et al. and He et al. are the de-facto standard references for weight initializations. They teach us to keep variances of activations and gradients consistent across forward and backward passes respectively, but reveal nothing about the distribution from which our weight vectors should be drawn.**

In this paper, we explore several promising weight initialization schemes and evaluate them by training models on the MNIST handwritten digits dataset. In all of our experiments, we make sure to keep the average length of a weight vector equal to the average length of He initialized weight vectors for fair comparison. After conducting our experiments, we assess the statistical significance of our results. None of our proposed schemes were statistically significantly better than He et al initialization. We examine their theoretical advantages and practical limitations in order to analyze our results and propose a few directions for future research in this area.

***Index Terms*—He, Xavier, initialization, theory**

I. Introduction and History

When first learning about neural networks, one is usually told that they should initialize weights and biases to "small random values centered around 0." This leaves one with questions like, and "Why small values?", "Why not all initialize all weights as zeros?", "How small?", and "From what distribution?" All of these answers except that last one have good answers. We choose small values because large values cause neurons to saturate and therefore take longer to train with sigmoidal/tanh activation (and cause "dead neurons" with ReLU.) Setting weights as zeros initially causes neurons to learn the exact same thing as each other due to identical gradients. The question of "How small?" is discussed later in this introduction and has an acceptable answer. The most pressing unanswered question is still, "From which distribution?." The uniform and normal distributions seem like good choice in the absence of any other evidence, but more exploration in this area is required to say that either of these is truly optimal.

The most common initialization today is Xavier initialization for linear activation and He initialization for ReLU activations. Both of these are (usually normal or uniform) distributions centered around zero with different standard deviations suitably chosen for their specific activation. Xavier initialization works well for linear activation because those activations functions have an expected value of 0 assuming uniformly distributed input. Since ReLU is biased toward higher activations, the standard deviations need to be adjusted.

To see how this works, consider the inputs of a fully connected linear neural network layer to be a list of n_{in} random variables, x_i. If we have weights, w_i, then our output, $out = \sum_{i=1}^{n_{in}} w_i x_i$ (ignoring biases for now) can be considered another random variable which is just the dot product of our x's and our w's. The question now is, "What is the standard deviation of a neuron's output, out?" If you multiply two random variables, w_1 and x_1, both with mean 0, then that product will have variance $var(w_1 x_1) = var(w_1) \times var(x_1)$. Now if you add two such products, this sum will have variance $var(w_1 x_1 + w_2 x_2) = var(w_1) \times var(x_1) + var(w_2) \times var(x_2)$. So as long as our w's and x's have mean 0, otherwise complicated variances of polynomials in our w's

and x's can be calculated as a simple function of the variance of our w's and x's. If we assume that the inputs all have variance V and they are independent variables, then our output, $out = \sum_{i=1}^{n_{in}} w_i x_i$ has the following variance:

$$var(out) = var(v_1 x_1) + \cdots + var(v_{n_{in}} x_{n_{in}})$$
$$= var(v_1)var(x_1) + \cdots + var(v_{n_{in}})var(x_{n_{in}})$$

If we choose our weights to have variance $1/n_{in}$, then this equals $\underbrace{V/n_{in} + V/n_{in} + \cdots + V/n_{in}}_{n_{in} \text{ terms}} = V$, the same variance as our inputs!

A similar derivation exists for keeping the variance of the gradients the same as well. In that case, the prescribed variance is $1/n_{out}$ where n_{out} is the number of neurons in that layer instead of $1/n_{in}$ (the number of inputs). Also, the above derivation was limited to linear activations, but the derivation in He et al. was similar, but made the necessary modifications for accommodating ReLU. Their resulting variance is surprisingly simply twice that derived above ($2/n_{in}$ instead of $1/n_{in}$). An in-depth discussion of this can be found in [2].

These weight initializations are the current state of the art. Without a similar scaling of weight variances, deep networks can have intermediate and later layers saturated from the beginning of training. It is worth noting that the theoretical basis for these initializations hold only if the dimensions of our data are independent. Also, the properties these initializations guarantee are most helpful if our data is also normalized as a preprocessing step. Despite the successes of these initializations, we suspect that they can still be improved upon, since we can change anything about the distribution other than the mean and variance without affecting their analysis. In practice, deep neural networks often have many redundant units which learn highly correlated features of the inputs. This is a waste of computational resources and indicates that perhaps weight initialization could be improved so that weight vectors are less correlated. This could also conceivably reduce overfitting.

One idea for how this could be achieved is by making sure that our weight vectors are initialized far away from each other. This idea is motivated by the fact that weight vectors which are very close to each other correspond to neurons with similar outputs which will lead them to having similar gradients and learning similar things. Good questions for future research are, "How far apart is enough?", "What is the best metric for measuring distance in this context?", and "Are there any downsides to initializing them as far apart as possible?"

Saxe et al.[1] seem to suggest that making weight vectors orthonormal is a good idea. If orthonormal is up for consideration, one might also consider orthogonal initializations which do not all have the same length vectors (but, of course, still have length centered around that proposed by He et al.). Therefore we make an "orthogonal" initialization which is a version of our orthonormal initialization where we randomly scale up and down weight vectors afterward. In our future experiments, our "orthonormal" initialization is not truly orthonormal, since truly orthonormal vectors must have length equal to one. In our case, we want the length of a vector to be $\sqrt{2}$ such that the average component of the vector has variance $2/n$ as in He et al. initialization.

Sarkar[4] suggests an initialization he calls "orthoordent" which forces weight vectors into unique quadrants using Hadamard matrices. A Hadamard matrix is a matrix of 1's and -1's whose rows are mutually orthogonal. The Hadamard product of two matrices multiplies elements component-wise. (Perhaps what you thought matrix multiplication "should" have been when you first learned it.) Therefore the Hadamard product of a matrix of all positive entries with a Hadamard matrix produces a new matrix which has one row in each quadrant. (Note that, although rows of the Hadamard matrix are orthogonal, rows of this product matrix will not be.)

A similar initialization to Sarkar's which we will call "quadrant subset initialization" can be found by simply selecting random quadrants (without replacement) and initializing a weight vector within that

quadrant. This bypasses the need to find a Hadamard matrix of a particular order. If quadrant subset ends up performing better, then this would be a major advantage of this method, since Hadamard matrices do not exist for orders other than 1,2, and multiples of 4. In fact, it is not even known if there exists a Hadamard matrix of order $4n$ for every positive integer, n. The Hadamard Conjecture states that there is, but it has not been proven.

II. Overall Strategy

We evaluate 5 different weight initialization schemes for a relatively shallow network on the MNIST handwritten digits dataset[3]. The weight initialization schemes we consider are the ones we discussed briefly above. He et al., orthonormal, orthogonal, orthoordent, and "quadrant subset."

A. Model Architecture

The architecture for our experiments will be the following:

5x5 Convolutional layer with 5 filters.
– 2x2 max-pooling –
3x3 Convolutional layer with 5 filters.
– 2x2 max-pooling –
40 Unit Fully-Connected Hidden Layer
10 Unit Fully-Connected Output Layer (with Softmax)

After the first convolutional layer, we will have activation shape (24,24,5) and after the max-pooling this will be (12,12,5). After the second convolutional layer, the shape will be (10,10,5) and after the max-pooling this will be (5,5,5). The number of parameters for each of these layers is the following:

5x5x1 = 25
3x3x5 = 45
(5x5x5)x40 = 5000
40x10 = 400

This gives a total of 5470 parameters. This is very few parameters, but it is ideal for a baseline since MNIST is task which requires very few parameters in order to get decent accuracy. We were also wary of introducing much more complexity since we were exploring weight initializations and not parameter tweaking. Given more time and computational power, a grid search over more architectures would be conducted and an in-depth analysis of the average or expected error across all architectures would be performed.

B. Experiments

Fundamentally, our method consisted of training 50 neural networks using PyTorch for each of our 5 initialization schemes for a total of 250 networks. (In these networks, the convolutional layers were the only ones initialized with novel initializations. The fully-connected layers were kept with a standard He et al. initialization for consistency.) We used PyTorch since its dynamic graph and simple API made it quicker for us to prototype than TensorFlow. See the Open Neural Network Exchange[5] for details on how to convert deep learning models between frameworks.

All networks were trained with the same learning rate of 0.0001 and batch size of 32. All networks were also trained using the Adam optimizer for simplicity and quicker training. Normally, we would have considered opting for the simpler stochastic gradient descent in order to keep our complexity to a minimum for our experiments, but the long training time and large number of networks trained made this prohibitively slow to reach any sort of good accuracy.

Each of the 50 networks for a given initialization scheme was trained using a randomly sampled 95% of the data, using the other 5% as validation data for that particular network. This is a standard form of cross-validation (known as "repeated random subsampling cross-validation" or sometimes "Monte Carlo cross-validation"), but it is less common than k-folds cross-validation. (Perhaps because some data points can be in the testing set for multiple networks.)

One potential advantage of this method is that the proportion of the training/validation split is independent of the number of iterations. This allowed us to train 50 networks each with 95%

of the data only instead of 50 networks each with 98% of the data like you would need to do with k-folds cross validation.

During training of these networks, for each network, we stored the loss and accuracy of the network on every twentieth training batch as well as the loss and accuracy of the network on the validation data (also only stored every twentieth batch.) The reason why we only save training stats every twentieth batch is because it takes up a lot of space in memory and that level of granularity was unnecessary anyway. We use these values later to compare the loss/accuracy over time between different initialization schemes.

In particular, we use a program called TensorBoardX to write TensorBoard logs from PyTorch. We store the following different types of graphs to analyze later:

1) Average training loss over time.
2) Average training accuracy over time.
3) Average validation loss over time.
4) Average validation accuracy over time.
5) Average difference between the validation accuracy of an initialization and validation accuracy of He et al. over time.
6) Standard Deviation of the difference between the validation accuracy of an initialization and validation accuracy of He et al. over time.
7) z-scores for the difference between the validation accuracy of an initialization and validation accuracy of He et al. over time.

When we say "average validation accuracy over time" we mean that we took the 50 different graphs of validation accuracy over time for each initialization, and averaged them. Therefore, this line graph has one line per initialization scheme and the ith position on the x-axis has the y-value which is the average of the validation accuracies at this point in time during training over all 50 cross-validation trials.

The last 3 graphs will perhaps be the most important for our analysis since they put the accuracies over time for each initialization in relationship with the accuracies over time for He initialization and ignore absolute performance by simply focusing on the "advantage" one initialization has over another.

In particular, the final graph not only shows advantages, but measures the statistical significance of the advantages for each time step by displaying the advantages as a z-score!

III. Compromises

If a layer of our network transforms m units of its input, x, into n units in its output, y, then the weight matrix for this would have shape n by m if $y = xW + b$ with x,y, and b all row vectors. Note that our weight matrix, W is not square.

Therefore, for fully-connected layers in our neural network which have more units than the previous layer, the corresponding weight matrix will have more weight vectors than the dimension of weight-space. In this case, a truly orthogonal, orthonormal, orthoordent, or quadrant subset initialization is impossible! In order to address this problem, we simply initialize as many desired weight vectors as possible and then fill in the rest of the weight vectors as He et al. initialized random weight vectors as a default. For further details of our methods, see our PyTorch code in the appendix.

For fully-connected layers in our neural network which have fewer units than the previous layer, the corresponding weight matrix will have fewer vectors than the dimension of weight-space. In this case, we simply select a subset of the weight vectors created by the corresponding full-rank initialization.

We considered altering our architecture in order to mitigate the effects of these compromises, but decided against it since any discussion of a general scheme for weight initialization should not restrict itself to a type of initialization which forces researchers into a very limited space of architectures.

IV. Methods for Analysis

One of the hardest problems of this entire project was the slow time for testing ideas. Every time

we had a new idea, it would take at least 6 hours to train our collection of networks to get a good idea of how things compared. The experimental results presented here truly represent only a small portion of the experiments we ran. To the best of our ability, these results represent an fair and accurate comparison of the best versions of each initialization.

For the first half of this project, we were inadvertantly using Xavier et al. initialization instead of He et al. initialization. We also had lurking problems in making each initialization scheme the best version of itself. For instance, at various times in our experiments, we were multiplying the length of our orthonormal weight vectors by 2 or $\sqrt{2/n_{in}}$ (where n_{in} is the length of the weight vector) instead of $\sqrt{2}$. This is just one example of the many small mistakes which significantly added to the length of this project due to the unavoidably long prototyping time.

Due to my ignorance of basic statistics, I was running these experiments for a long time before considering how to assess statistical significance. After a bit of statistical priming from the internet, we implemented p-values for the difference in validation accuracy of each initialization with He et al.

For each timestep, t, we have 50 accuracy difference values for a given initialization. We can think of this as a sample of size 50 from a distribution. We don't know what this distribution is, but the central limit theorem tells us that as we increase the sample size, the mean of a sample approaches a normal distribution.

This is called the sampling distribution of the sample mean. The sampling distribution of the sample mean has a mean equal to the original distribution in the limit. The standard deviation of the sampling distribution of the sample mean can be estimated by $\sigma/\sqrt{k}$, where σ is the standard deviation of our sample $k = 50$ is our sample size.

The null-hypothesis here would be that each initialization has mean validation accuracy equal to the mean validation accuracy for He et al. This means that their difference should be equal to zero on average. If we assume the null-hypothesis, then we use 0 as the mean of the sampling distribution of the sample mean. We have an estimate of the standard deviations of the sampling distribution of the sample mean. With these two components, we can calculate the probability of experiencing a mean difference of a sample of size k by calculating a z-score.

In order to make this more rigorous, one should truly conduct an analysis of how closely our distribution of sample means matches a normal distribution. This would involve measurements of skew and kurtosis. We did not conduct these, however, since a sample size of 50 seemed plenty adequate given our data.

V. Results

We conducted the same experiment 3 times in order to make sure any results were consistent even after the averaging 50 trials. The lines of "z-score of advantage over He" generally stay between 2 and -2 and vary across time. This variance across time is interesting as it seems to indicate that even when one of these initialization gets a temporary advantage over He initialization, it eventually goes away. If you care to see the graphs, you can use tensorboard to visualize the logs which we will upload to github here: https://github.com/jackeown/nnInitializations

P-values give you the probability that you would see results at least as extreme as you did given some null-hypothesis. In our case, our p-value tells us the probability that we would see the advantages we saw given that the initialization was just as bad or worse than He initialization.

A p-value of 5% is usually considered significant, but since we're plotting a line with many samples which could cross that threshold at any point, this may be a form of "p-hacking" to consider such a low threshold significant. This threshold of 5% would be acheived at a z-score of around ± 1.65. A z-score of 2.0 would give a p-value of about 2.3% and a z-score of 5.0 would give a p-value of well

below 0.001%!

Earlier on in my experimenting with these initializations, I found that I had an error with my initializations which was making all of them worse except for one. This made that one initialization seem significantly better than the others and it had a z-score of over 5 consistently! This is the type of behavior I'd expect from an initialization which is truly better than another. Once I fixed the major bugs in my implementation, the significance of the advantages of one initialization over another faded greatly.

VI. Conclusion

Our results are a bit disheartening, but there is some hope. Much like He initialization is a modification of Xavier initialization so that it will be optimal for ReLU, perhaps there is a modification for orthogonal initialization which makes them optimal for ReLU.

Another large factor was that we only enforced these properties of weight vectors during initialization and forgot about them during training. It would be interesting to see what happens if you actively try to put distance between weight vectors during training maybe with a custom loss function.

Perhaps the idea of iteratively pushing He initialized weight vectors further apart as a preprocessing step has merit. The testing code I've written at least gives me a framework for evaluating the statistical significance of future experiments with other architectural changes.

References

[1] Surya Ganguli Andrew M. Saxe, James L. McClelland. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. 2014.

[2] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015.

[3] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[4] Dilip Sarkar. Ortho-ordent initialization of feedforward artificial neural networks (ffanns) to improve their generalization ability. 1995.

[5] Onnx Development Team. Open neural network exchange, 2018.