Word Count: 3073
Session: May 2018

**International Baccalaureate**
**Mathematics Extended Essay**

# Addressing the Difficulty of Training Deep Neural Networks Using Gradient-Based Optimization Methods

## Contents

# 1. Introduction

Artificial Neural Networks (ANNs) are complex modelling techniques that can be used to find the relation between the output of a complex multivariable function and its arguments, effectively approximating it.

Nowadays, the technique is being used successfully for a variety of tasks ranging all the way from predicting liver cancer[1] and approximating physics of fluids[2] to autonomous vehicles and colorizing black and white videos[3]. Neural networks require many layers to be stacked for such complex classification and prediction problems as they need to develop complex feature detectors for the data used for these tasks.

Apart from the fact that computation power and memory required increases as layers get stacked up, there is also architectural issues that make training deep neural networks (DNNs) very hard, such as vanishing/exploding gradients and numerical instability.

As a way to address the vanishing gradient problem, researchers working in the field of machine learning have been using ReLU[4] $(\max(x, 0))$ activators.

Although ReLU activator is a great solution to the vanishing gradient problem, it is not the only solution and almost definitely not a perfect solution.

ReLU activator has many cons of its own which will be touched upon in more detail later in this thesis, such as non-derivable parts or dying units.

[1] (Ye, et al., 2003)

[2] (Tompson, Schlachter, Sprechmann, & Perlin, 2016)

The primary concern of this thesis is to accelerate deep neural networks' training phase further and address the numerical problems with gradients at the same time. We propose two new approaches which when combined lets us achieve this goal.

We propose an activation function that is much more stable and flexible and a weight initialization technique that lets us stack up layers and have "gradient-equalized" layers no matter how deep the network is when combined with the proposed activation function.

## 1.1 Structure of a Deep Neural Network

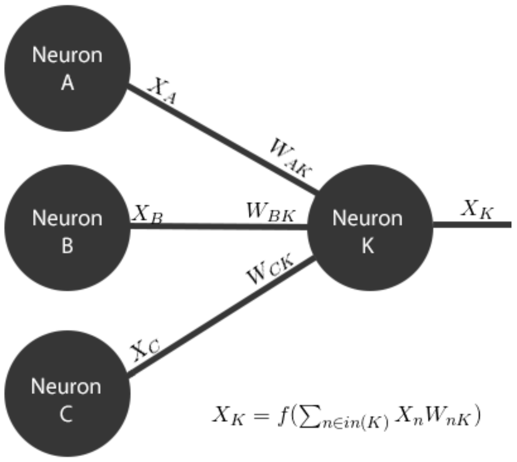

Figure 1: Structure of a simple neural network

Although the way neurons are connected may vary (recurrent, convolutional, fully-connected etc.) neural networks and likewise DNNs generally follow a single model. Values from pervious layer are multiplied by their respective weights, summed up and then are fed to an activation function as in Figure 1. Some of the common choices for activation functions are sigmoid $\left(\frac{1}{1+e^{-t}}\right)$, hyperbolic tangent $\left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)$, ReLU $(\max(x, 0))$ and its variations.

[3] (Iizuka, Simo-Serra, & Ishikawa, 2016)

[4] (Nair & Hinton, 2010)

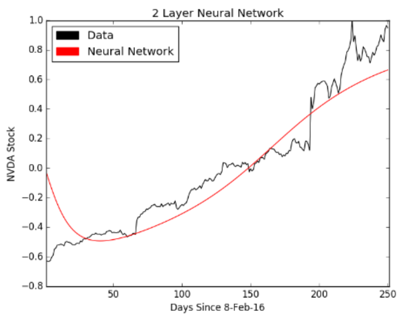

Figure 3: Output of a 2-layer ANN trained on stock value of NVDA

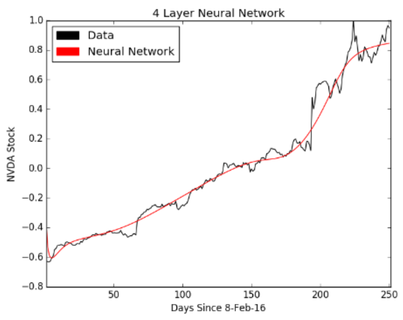

Figure 2: Output of a 4-layer ANN trained on stock value of NVDA

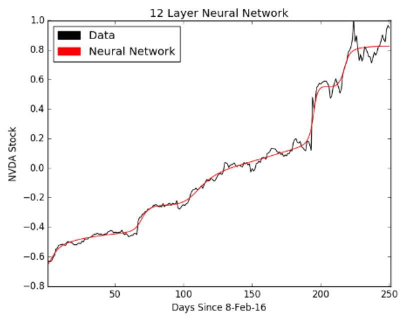

Figure 4: Output of a 12-layer ANN trained on stock value of NVDA

After the forward propagation is complete in order to train the neural network, we use back propagation. In this phase we define an error function, and using our optimization method we change all the parameters (such as $W_i$ in Figure 1) in order to minimize the error function.

In order to keep this example simple, we are going to use stochastic gradient descent (SGD) as our optimization method and mean squared error (MSE) as in Equation 1 for our error function.

$$E(\hat{y}) = \frac{1}{2}(\hat{y} - y)^2 \qquad (1)$$

The principle of SGD is very simple. We first calculate how much impact each parameter had on the error function using chain rule. After that is done, we simply subtract that gradient after multiplying it with a learning rate (hyper-parameter chosen when constructing model, mostly between 0.1 and 0.0001).

Let $\varphi$ be an optimizable parameter and $\mu$ be the learning rate. We would update every parameter like shown in Equation 2.

$$\hat{\varphi} := \varphi - \mu \frac{\partial E}{\partial \varphi} \qquad (2)$$

When we try simulating a weight update, the reason behind gradient instabilities becomes really clear. Let's update $W_{BK}$ from Figure 1, the rate of change for the parameter is:

$$\Delta \widehat{W}_{BK} = -\mu \frac{\partial E}{\partial W_{BK}} \qquad (3)$$

Which is equal to:

$$-\mu X_b (X_k - y) f'\left(\sum_{n \in in(K)} X_n W_{nK}\right) \qquad (3.6)$$

Refer to Appendix 1 for the proof.

The culprit behind all numerical instabilities is $f'(\dots)$. With an identity activation function the weights will eventually blow up and $X_i$ will become unrepresentable using 32-bit floating numbers, and with an activation function that have an exponentially decreasing rate of change such as tanh or sigmoid, the derivative of the function will get very small, eventually making no updates at all to the parameters; one obvious solution would be to decrease the number of layers but the depth of a network has significant effect on its ability to represent functions, as demonstrated in Figures 2, 3 and 4; which will get investigated further in Section 1.2.

Shallow neural networks sacrifice accuracy and complexity for ease of training, but they are not very suitable for everyday use due to their inaccuracy. With fields that use neural networks for critical decisions such as autonomous driving or cancer prediction, sacrificing accuracy is simply intolerable.

## 1.2 Examining the Effect of Network's Depth to Feature Complexity

One easy way to measure how complex the features developed by a neural network is to compute the numerical derivative for every point on the predicted line and calculate the array's variance like in Equation 4.

$$\vartheta = \mathrm{Var}(\{y'(0), y'(0.1), \dots\}) \qquad (4)$$

| Number of Layers | $\vartheta$ ( Complexity ) | $\frac{\vartheta}{\min \vartheta}$ |
|---|---|---|
| 2 | 0.000063 | x1.000 |
| 4 | 0.000977 | x15.460 |
| 12 | 0.002224 | x35.206 |

Table 1: Computed $\vartheta$ values of the graphs from Figures 2, 3 and 4

Referring to Table 1, the 12-layer network developed features that are ≈35 times more complex compared to the 2-layer network in 125k iterations.

## 2. Pre-Examination of the Problem

There are multiple issues with training DNNs using SGD.

One of the biggest issues are unstable gradients; gradients that blow up due to unrestricted activation functions (ReLU, Identity, ...), and gradients that vanish due to being too close to the activator's horizontal asymptotes (tanh, sigmoid, ...).

Another problem would be the unstable learning rate. Learning rates for each layer are different due to the nature of gradient-based optimization methods such as SGD, despite the fact that naturally, one would want every layer to learn at similarly distributed speeds to accelerate the learning process. [5]

[5] (Singh, De, Zhangy, Goldstein, & Taylorz, 2015)

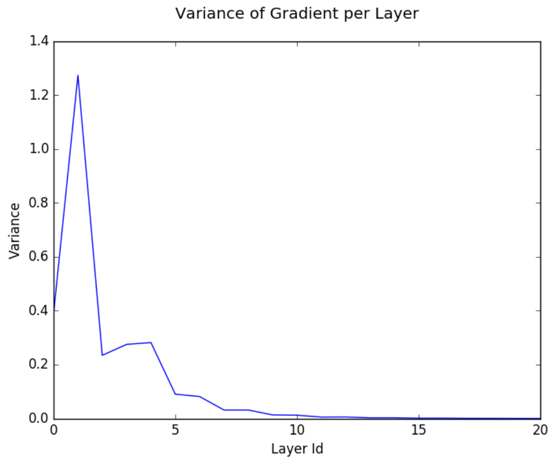

Figure 5: Variance of gradients per layer in a DNN

Figure 5 illustrates both problems perfectly. As it can be seen from the figure, after 15th layer is passed, the variance –and the numerical value– of the gradients becomes effectively zero, causing the layers afterwards to not update. We can also see the problem of changing distributions very easily.

In order to fix these issues, this thesis proposes a new activation function which is much more flexible when it comes to relatively big numbers and yet can still reach [-1, +1] easily and has zero mean.

This new activation function will also fix the problems with ReLU activations that are being used currently by many researchers for training deep neural networks in order to combat with the vanishing gradient problem.

ReLU layer is inherently a bad choice but works well due to its ability to pass gradients linearly. The problems with ReLU layers include; being non-derivable at zero, non-zero mean nature and the probability of disabling up to 50% of the network.

## 3. Approaching the Problem

$$Z\,\mathrm{Var}\big(\delta f'(X)\big) \le Var(\delta) \tag{5}$$

$$\sigma^2 = \frac{Z}{n_{in}} \tag{6}$$

In order to make gradients stay similar across layers and make them stable, we are going to solve the Equation 5 for the constant Z where $\delta$ is the gradient from the next layer and $f$ is the activation function and mix it with Xavier initialization[6] $\left(\sigma^2 = \frac{1}{n_{in}}\right)$ like in Equation 6.

As for the activation function, we are going to pick a function which increases quadratically instead of exponentially unlike most activation functions used (sigmoid, tanh, etc.) and squash it to the range we need.

### 3.1 Proposed Activation Function

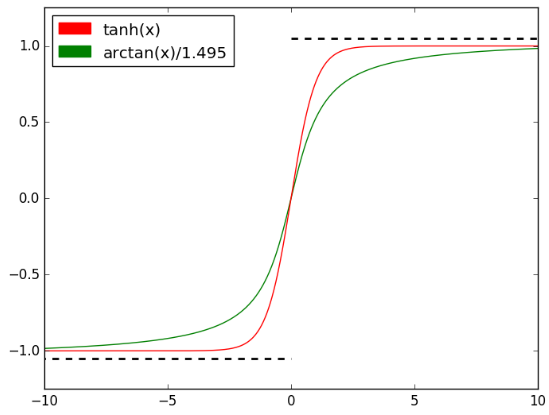

Figure 6: Comparison of tanh(x) and arctan(x)/1.495

This thesis proposes the activation function:

$$f(x) = \frac{\arctan(x)}{\gamma} \tag{7}$$

The function $\arctan(x)/\gamma$ meets the requirements of universal approximation theorem[7].

It is a non-constant, bounded and continuous function which is monotonically increasing.
The activation function reaches its maximum and minimum respectively at $+\infty$ and $-\infty$ which correspond to $\frac{\pi}{2\gamma}$ and $-\frac{\pi}{2\gamma}$.

To understand the most important feature of the proposed activation function we first must derive it.

$$\frac{df}{dx} = \frac{1}{\gamma(1+x^2)} \tag{8.7}$$

Refer to Appendix 2 for the proof.

The proposed activation function increases quadratically which is a big advantage compared to tanh which increases exponentially as seen in Equation 10.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{9}$$

$$\frac{d}{dx}\tanh(x) = \frac{4e^{2x}}{(e^{2x}+1)^2} \tag{10}$$

| Value | $\mathbf{\frac{1}{1.495(1+x^2)}}$ | $\mathbf{\frac{4e^{2x}}{(e^{2x}+1)^2}}$ |
|---|---|---|
| -15 | 0.00296 | 3.74304e-13 |
| -3 | 0.06689 | 0.00986 |
| 3 | 0.06689 | 0.00986 |
| 15 | 0.00296 | 3.74304e-13 |

Table 2: Comparsion Between the Derivatives of tanh and arctan ($\gamma$=1.495) Activators

As seen in Table 2, learning rate became $\approx$ 300000 times slower with tanh when the activated value became 15 instead of 3 which is a minimal change. Tanh and likewise all exponential activation functions suffer from being too sensitive to input range.

[6] (Glorot & Yoshua, 2010)

[7] (Barron, 1993)

The proposed activator's range is close to [-1.0507, +1.0507] for $\gamma = 1.495$. Although this has the possibility to increase the entropy of the network, it is a minimal cost paid for its advantages. For $x \cong 13.168$, activator's output reaches 1.0. This makes a 1:1 relation much easier to represent. The fact that the activator is much more flexible to big numbers also makes us able to equalize gradients of all layers over the network no matter the depth which is a big advantage as it will be seen in Section 3.2.

Although arctan's main advantage becomes clearer with a DNN used for generalization tasks such as image recognition, we tried the activator on a regression task. We fed 200 days history of NVIDIA's stock value to neural networks using tanh activators and arctan activators with same topology and hyper-parameters.

| Activator | Mean Squared Error | Percentage Error |
|---|---|---|
| arctan | 0.000734 | 3.83% |
| tanh | 0.001117 | 4.72% |

Table 3: Comparison of tanh and arctan activators' performances on a regression task.

Although we expected tanh activator to perform much better in this task due to its very quickly increasing nature and the fact that we only let each network run for 40k iterations, arctan activator ended up performing better in this task. This was a solid proof that arctan activators could replace activators used for logical units (tanh, sigmoid, etc.) in DNNs with success.

The next test case was 64x64 image classification with classes: car crash, car, gun, police car, person and miscellaneous using a custom dataset. This is where arctan activator performed significantly better.

Figure 7: Some examples from the used dataset

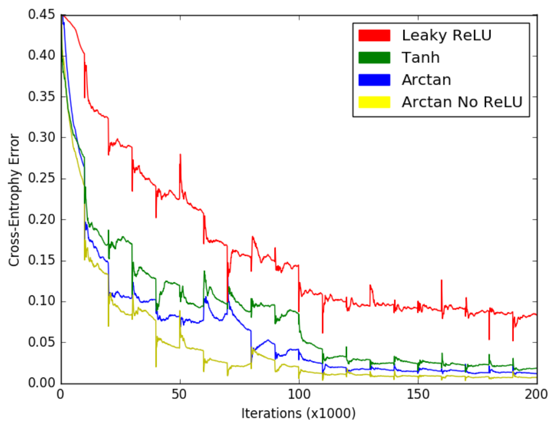

Figure 8: Error history of DNNs with different activators over 200k iterations using the dataset

This test proved us that the proposed activator could also replace ReLU activators with relatively great success.

Network with only Leaky ReLU as activators achieved a Cross-Entropy Error of 0.081523, and the network with arctan replacing the activator functions used for last fully connected layers achieved 0.012521 while the one with arctan replacing all activator functions achieved 0.008023 Cross-Entropy Error.

We can conclude from our experiments that, due to arctan activator's zero-mean, between ReLU (linear) and tanh (exponential) nature, arctan is very flexible and can be applied to most regression, classification or prediction problems.

## 3.2 Achieving Gradient Stability

Solving the gradient instability issue is a big factor to success as it can be seen from ReLU example.

ReLU is being used by many researchers in the field of machine learning for deep neural networks simply because of their one single property, which is non-vanishing gradients.

As it was shown in Figure 5 previously, neural networks with many layers suffer from vanishing gradients problem when they are activated by an input-squashing function repeatedly.

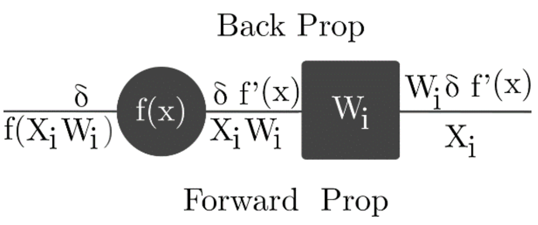

Figure 9: A simple model demonstrating how a gradient passes an activation function and a weight layer pair

As it can be seen from Figure 9, when the gradient $\delta$ passes the activator and weight layers, it gets multiplied by both the activators derivative and the value of the corresponding weight $W_i$.

This lets us manipulate the gradients distribution simply by changing the distribution of weights which is initialized by us.

In order to use arctan activations without worrying about killing the gradient and have same gradient distribution every layer, we want to manipulate the gradient in such a way using the weight distribution that, f'(x) will not have an effect on $\delta$ after it is multiplied by $W_i$.

We can achieve this by making the assumption $E(\delta) = 0$. What we need to solve after this is the following equation:

$$Z\,\mathrm{Var}\big(\delta f'(X)\big) \le Var(\delta) \qquad (5)$$

We can use the assumptions that $E\big(f'(x)\big) = 0$ and $E(\delta) = 0$ to simplify Equation 5 to:

$$Z = \frac{1}{\max\limits_{X\in\mathbb{R}^n} \mathrm{Var}\big(f'(X)\big)} \qquad (5.8)$$

Refer to Appendix 3 for the proof.

If a tensor transformed by function $f$ has the maximum possible variation, distribution-wise we can simplify the tensor to a set of:

$$\Big[\min_{x\in\mathbb{R}} f(x),\ \max_{x\in\mathbb{R}} f(x)\Big] \qquad (11)$$

Hence, in order to find the maximum variance of a set transformed by $f'(x)$ we should calculate the minimum and the maximum values of $f'(x)$.

We are going to use the derivative of arctan(x) / $\gamma$ in our case, the function $\frac{1}{\gamma(1+x^2)}$ reaches

- its minimum at x=∞, y=0
- its maximum at x=0, y= 1/γ

We use $\gamma = 1.495$ so the set becomes $[0, \frac{1}{1.495}]$ which has a variance of ≈0.1119. This gives us a Z value of ≈8.940. Mixing this technique with Xavier initialization we are going to initialize our weight-tensors like in Equation 12.

$$W_i \sim \mathcal{N}\left(0, \frac{Z}{n_{in}}\right) \qquad (12)$$

Using the same DNN from Figure 5, we can demonstrate how this weight initialization technique effected the gradient distribution.

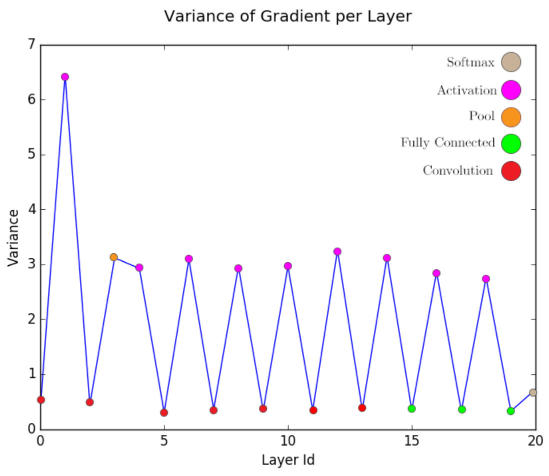

Figure 10: Variance of gradients per layer in the DNN from Figure 5 when our method is used

As it can be seen from Figure 10, weight layers (marked with red and green) have a stable gradient distribution that no longer leads to vanishing gradient problem after getting initialized with our method.

In order to test for this technique's effectiveness, we are going to be training the DNNs from Figure 5 and Figure 10 on our dataset and compare the speed of convergence.

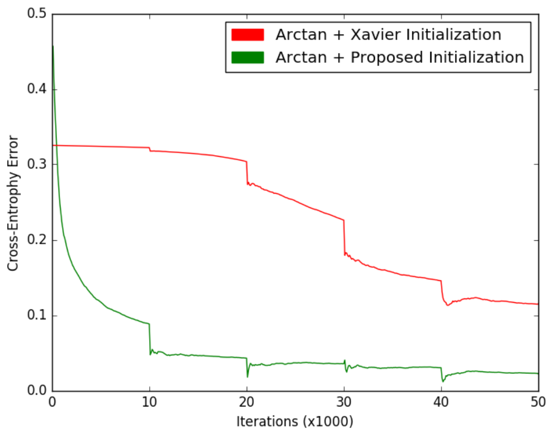

Figure 11: Error history of 2 DNNs initialized using different weight initialization techniques

As it can be seen from Figure 11, the DNN trained using the technique we proposed (marked by green) was not affected by vanishing gradient problem at all and reached the error other DNN reached after 50k iterations only in 5k iterations.

DNN which used Xavier initialization on the other hand (marked by red) struggled to start and wasted many iterations simply trying to get the gradients to bigger numbers.

There are as expected cons to this technique. This technique cannot be used with exponentially increasing activation functions and requires a soft function like arctan due to the fact that with increased variance the values get higher and exponential activation functions simply cannot handle such high initial values.

Although this may simply not be an issue at all as we saw in Section 3.1 that arctan activators performed as good as if not better when compared with the traditional activation functions.

## 4. Experimentation

An easy way to test how the technique performs when many and many layers stacked together without waiting for hours to get a result is to train the network on XOR dataset.

We built a network with 24 fully-connected layers with 16 neurons each, stacked on top of each other.

We then, trained the network once with Xavier initialization and various activators (including ReLU) and once with the initialization technique we proposed and arctan.

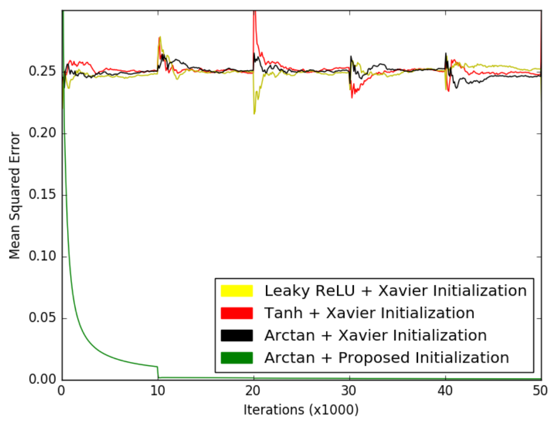

Figure 12: Error history of various methods on a very deep network trained on XOR

It can be seen from Figure 12, Although none of the networks seem to be learning when the network is that deep, arctan network had MSE of less than 0.0004 by 50k iterations. Similar results were obtained using 48 layer structures within 50k iterations as well.

We can easily conclude that the proposed technique lets us train deep neural networks in a much faster manner compared to its alternatives including the activators (ReLU and variants) that does not have the problem of vanishing gradient.

In order to test for our hypothesis that gradient-normalized arctan networks will perform better than their ReLU equivalents in a real-life problem, we are going to use the custom dataset we mentioned in Section 3.1.

We are going to use a 12-layer deep neural network with 9 convolutional layers and 3 fully connected layers and see how the training concludes after 125k iterations.

Before starting let's examine each networks gradient distribution.

From Figure 13 it can be seen that the variance of gradients is quite similar for each weight layer unlike Figure 14.

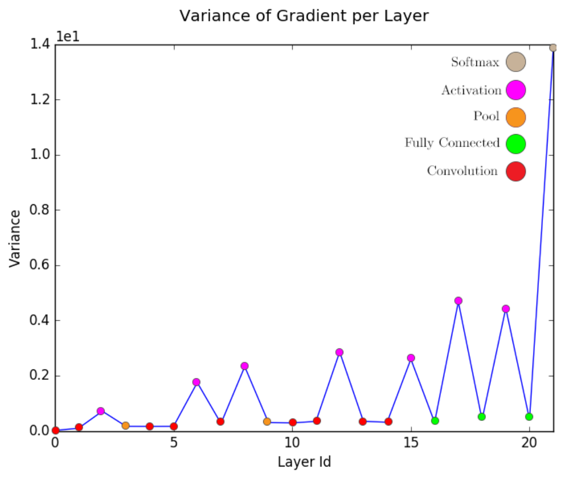

Figure 13: Variance distribution of the demo DNN initialized using our method

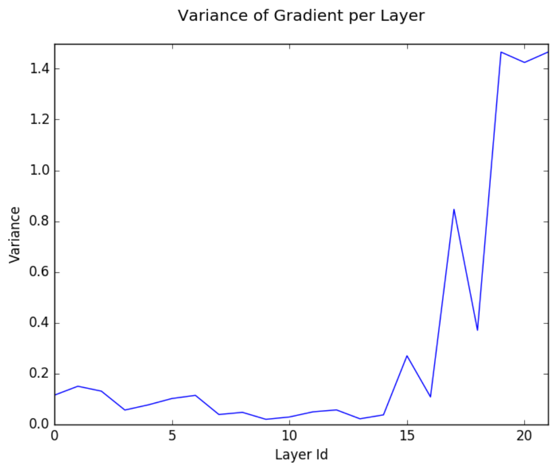

Figure 14: Variance distribution of the demo DNN initialized using Xavier initialization

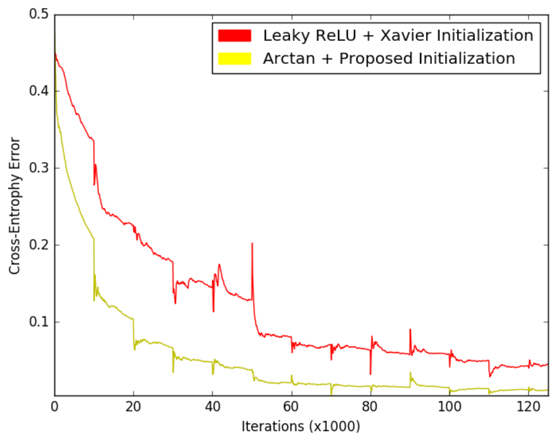

Figure 15: Error history of 2 DNNs initialized using different weight initialization techniques for the demo task

It is also very clear from Figure 15 that the network initialized using our technique performs much better in the real-life test.

|  | Arctan + Proposed | ReLU + Xavier |
|---|---|---|
| Test Error | 1.794% | 7.289% |
| Cross Entropy Error | 0.005337 | 0.028781 |
| Computation Time | 1588 seconds | 1605 seconds |

Table 4: Comparison of two techniques

Using Table 4, we can see that not only our technique got a much better error rate both on testing and training it also ended up being as computationally efficient as ReLU when computed on GPU where most state-of-art computation is done nowadays.

## 5. Conclusion

Our technique makes gradient-stabilization possible for even the deepest networks and lets us train networks of any deepness with simple standard stochastic gradient descent (accelerated with momentum), performing better and better compared to alternatives as the networks get deeper.

This is a very important factor as state-of-art machine learning applications almost always use very deep neural networks with more than 20 layers as it can be seen in GoogLeNet[8].

The field of machine learning has been dominated by deeper and deeper neural networks ever since linear activator functions such as ReLU started to be used instead of their non-linear bounded counterparts such as sigmoid or hyperbolic tangent.

ReLU activators sacrifice performance by having dead units and non-zero mean output in order to achieve numerically stable gradients but we could see from our experiments that similar/better results can be achieved using a different approach which did not require these sacrifices to be made.

Contrary to popular-belief, deep neural networks can be trained using bounded activation functions without the worry of vanishing gradients using a different approach to the problem with relatively good success.

We can also see that “same learning rate at every layer” ideology of optimizers using adaptive learning rates can be achieved using vanilla stochastic gradient descent with this approach.

In the end, from both synthetic and real-life experiments we have conducted, we reached to the conclusion that gradient-equalized layers activated using bounded functions such as arctan can be used to reach lower error percentages considerably faster and keep up the performance throughout entire learning phase without experiencing any of the problems seen in other activators such as vanishing or exploding gradients.

### 5.1 Open Problems

1. Inability to apply this technique with tanh or sigmoid activators due to their sensitive nature remains.
2. Although softsign $\left(\frac{x}{1+|x|}\right)$ function was not tested, its output is very similar to arctan(x) /$\gamma$ bounded to [-1, +1] and may be a faster alternative.

[8] (Szegedy, et al., 2015)

## Bibliography

Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory.*

Glorot, X., & Yoshua, B. (2010). Understanding the difficulty of training deep feedforward neural networks. *Aistats. Vol. 9*, 249-256.

Iizuka, S., Simo-Serra, E., & Ishikawa, H. (2016). Let there be color!: joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification. *ACM Transactions on Graphics (TOG) 35*, 110.

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

Singh, B., De, S., Zhangy, Y., Goldstein, T., & Taylorz, G. (2015). Layer-Specific Adaptive Learning Rates. *IEEE 14th International Conference on Machine Learning and Applications (ICMLA).*

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 1929-1958.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Rabinovich, A. (2015). Going deeper with convolutions. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1-9.

Tompson, J., Schlachter, K., Sprechmann, P., & Perlin, K. (2016). Accelerating Eulerian Fluid Simulation With Convolutional Networks. *arXiv preprint*, 1607.03597.

Ye, Q.-H., Qin, L.-X., Forgues, M., He, P., Kim, J. W., Peng, A. C., & Simon, R. (2003). Predicting hepatitis B virus–positive metastatic hepatocellular carcinomas using gene expression profiling and supervised machine learning. *Nature medicine 9*, 416-423.

## Appendices

### Appendix 1

$$
\begin{aligned}
\Delta\widehat{\mathrm{W}}_{\mathrm{BK}} &= -\,\mu\frac{\partial \mathrm{E}}{\partial \widehat{\mathrm{W}}_{\mathrm{BK}}} && (3)\\
&= -\,\mu\frac{\partial \mathrm{X}_{\mathrm{K}}}{\partial \widehat{\mathrm{W}}_{\mathrm{BK}}}\frac{\partial}{\partial \mathrm{X}_{\mathrm{K}}}\left[\frac{1}{2}(X_k-\mathrm{y})^2\right] && (3.1)\\
&= -\frac{\mu}{2}\frac{\partial \mathrm{X}_{\mathrm{K}}}{\partial \widehat{\mathrm{W}}_{\mathrm{BK}}}\,2(X_k-\mathrm{y})\frac{\partial}{\partial \mathrm{X}_{\mathrm{K}}}[X_k-\mathrm{y}] && (3.2)\\
&= -\mu(X_k-\mathrm{y})\frac{\partial \mathrm{X}_{\mathrm{K}}}{\partial \widehat{\mathrm{W}}_{\mathrm{BK}}} && (3.3)\\
&= -\mu(X_k-\mathrm{y})\frac{\partial}{\partial \widehat{\mathrm{W}}_{\mathrm{BK}}}\left[f\left(\sum_{n\in in(K)} X_n W_{nK}\right)\right] && (3.4)\\
&= -\mu(X_k-\mathrm{y})f'\left(\sum_{n\in in(K)} X_n W_{nK}\right)\frac{\partial}{\partial \widehat{\mathrm{W}}_{\mathrm{BK}}}\left[\sum_{n\in in(K)} X_n W_{nK}\right] && (3.5)\\
&= -\mu\mathrm{X}_{\mathrm{b}}(X_k-\mathrm{y})f'\left(\sum_{n\in in(K)} X_n W_{nK}\right) && (3.6)
\end{aligned}
$$

## Appendix 2

$$f(x) = \frac{\arctan(x)}{\gamma} \tag{7}$$

$$\frac{df}{dx} = \frac{1}{\gamma}\frac{d}{dx}\arctan(x) \tag{8}$$

$$= \frac{1}{\gamma}\frac{dy}{dx} \tag{8.1}$$

Solving $\frac{dx}{dy}$,

$$x = \tan(y) \tag{8.2}$$

$$\frac{dx}{dy} = \frac{d}{dy}\tan(y) \tag{8.3}$$

$$= 1 + \tan^2(y) \tag{8.4}$$

$$= 1 + x^2 \tag{8.5}$$

Thus,

$$\frac{df}{dx} = \frac{1}{\gamma}\left(\frac{dx}{dy}\right)^{-1} \tag{8.6}$$

$$= \frac{1}{\gamma(1+x^2)} \tag{8.7}$$

## Appendix 3

$$Var(\delta) \geq \mathrm{Z}\,\mathrm{Var}\big(\delta \mathrm{f}'(\mathrm{X})\big) \tag{5}$$

$$Var(\delta) = \max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Z}\,\mathrm{Var}\big(\delta \mathrm{f}'(\mathrm{X})\big)\Big) \tag{5.1}$$

$$= \mathrm{Z}\max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Var}\big(\delta \mathrm{f}'(\mathrm{X})\big)\Big) \tag{5.2}$$

$$= \mathrm{Z}\max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Var}\big(\mathrm{f}'(\mathrm{X})\big)\mathrm{Var}(\delta) + Var\big(f'(x)\big)E(\delta)^2 + Var(\delta)E\big(f'(\mathrm{X})\big)^2\Big) \tag{5.3}$$

If $\mathrm{E}\big(\mathrm{f}'(\mathrm{X})\big) = 0$ and $\mathrm{E}(\delta) = 0$ then,

$$Var(\delta) = \mathrm{Z}\max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Var}\big(\mathrm{f}'(\mathrm{X})\big)\mathrm{Var}(\delta) + Var\big(f'(\mathrm{X})\big)(0) + Var(\delta)(0)\Big) \tag{5.4}$$

$$= \mathrm{Z}\max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Var}\big(\mathrm{f}'(\mathrm{X})\big)\mathrm{Var}(\delta)\Big) \tag{5.5}$$

$$= \mathrm{Z}\,\mathrm{Var}(\delta)\max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Var}\big(\mathrm{f}'(\mathrm{X})\big)\Big) \tag{5.6}$$

$$1 = \mathrm{Z}\max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Var}\big(\mathrm{f}'(\mathrm{X})\big)\Big) \tag{5.7}$$

$$\mathrm{Z} = \frac{1}{\max_{\mathrm{X}\in\mathbb{R}^n}\Big(\mathrm{Var}\big(\mathrm{f}'(\mathrm{X})\big)\Big)} \tag{5.8}$$