Adaptive Learning Rate Methods for Stochastic Gradient Descent and Vanishing Gradient Problems from Backpropagation

Tom Blain

Data Science Portfolio B

Abstract

In this paper, we provide an introduction to neural networks and explain the ease of gradient computation using backpropagation, which enables the utilisation of optimization tools such as Stochastic Gradient Descent (SGD) for adjusting weights and biases. We examine the challenges associated with SGD, including the careful choice of learning rate and the noise induced by stochastic evaluations of the gradient. To address these issues, we explore adaptive learning rate optimizers, such as Adam and AdaGrad, demonstrating their improvements in convergence and variance reduction. For adaptive learning rate optimisers, we primarily follow [1]. We then look at the vanishing gradient problem. This can arise from the activation functions when the network trains through backpropagation. This paper is written as an extension based on a paper for Variational Inference [2], which looks at stochastic optimisation for efficient and scalable algorithms in variational inference. This extension has gone into further detail of the adaptive learning rate algorithms and contextualised the methods for data science and neural network models, as well as introducing the problem of vanishing gradients which can be a result of backpropagation in neural networks.

Contents

1 Introduction

Gradient descent (GD) is one of the most popular algorithms to perform optimisation and by far the most common way to minimise loss functions in neural networks [3]. Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters.

Stochastic gradient descent (SGD) can be considered a stochastic approximation of gradient descent optimisation, since we aim to approximate the true gradient of the objective function using a noisy estimate of the gradient. This has the benefit of being computationally cheaper than gradient descent which finds the true gradient by using the entire dataset. Stochastic gradient descent provides unbiased estimates of the true gradient, meaning the expectation of our gradient estimate will converge to the true gradient over multiple batches and runs. Work on Stochastic gradient descent is extensive, and the ideas can be traced back to robbins-monro in the 1950s [4].

However, by implementing stochastic mini batches, we introduce noise into the gradient descent process. It is therefore of interest to study stochastic optimisation and variance reduction methods which may be able to lead to faster convergence and improved performance in many optimisation problems. Work on optimisation for deep learning models such as deep neural networks is essential since we have a large amount of weights and biases to train, where they may be a lot of local minima. Convergence to suboptimal points may limit the predictive power of our model. Another issue is noisy estimates of the gradient, which cause large loss function fluctuations in turn slowing down convergence speed.

Furthermore, when training neural networks we must select activation functions which help the network capture the often complex relationship between the data and the target. There is often no clear cut function selection, with many famous network architectures using a variety of functions. One issue however can be vanishing gradients which appear whilst training and affect the networks ability to learn. We are interested in making sure that the network is learning optimally and avoids such issues.

2 Context

2.1 Introduction to Neural Networks

In this paper, we study stochastic gradient descent with a focus on neural networks. Neural networks are a class of machine learning models that are loosely inspired by the biological neural networks that make up the human brain. These models consist of interconnected layers of artificial neurons or nodes, where each connection has a weight associated with it. The neurons receive input signals, process them, and produce output signals that are passed on to the next layer. The process of learning in a neural network involves adjusting the weights of these connections to minimize the error between the network's predictions and the ground truth.

2.1.1 Perceptrons

Neural networks can be in their simplist form, perceptron algorithms. A perceptron consists of an input layer, a processing unit, and an output layer. The input layer receives a feature vector $\boldsymbol{x} \in \mathbb{R}^d$, where d is the number of features. Each feature in the input vector is multiplied by a corresponding weight w_i, and the sum of these weighted inputs is computed:

$$z = \sum_{i=1}^{d} w_i x_i + b, \tag{1}$$

where b is the bias term. The sum z is then passed through an activation function $f(\cdot)$ to produce the output $\hat{y}$:

$$\hat{y} = f(z). \tag{2}$$

In neural networks, the objective function $J(\theta)$ is typically a loss function that quantifies the difference between the predicted outputs and the actual target values. The goal is to minimize this loss function by adjusting the model's the weights w_i and biases b of the connections between the neurons.

2.1.2 Activation Functions

Activation functions are typically mathematical functions which transform the input received by a neuron into an output signal. This allows us to

introduce non-linearity into our networks which enables the nerual network to learn more complex relationships between the inputs and outputs. There are many possible popular choices for activation function, and this is usually selected based on computational complexity and the specific problem.

- Sigmoid function: The sigmoid function maps input values to the range (0, 1). It is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{3}$$

- Tanh function: The tanh function maps input values to the range (-1, 1). It is defined as:

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{4}$$

- Rectified Linear Unit (ReLU) function: The ReLU function is a simple non-linear function that outputs the input if it is positive, and 0 otherwise. It is defined as:

$$f(x) = \max(0, x). \tag{5}$$

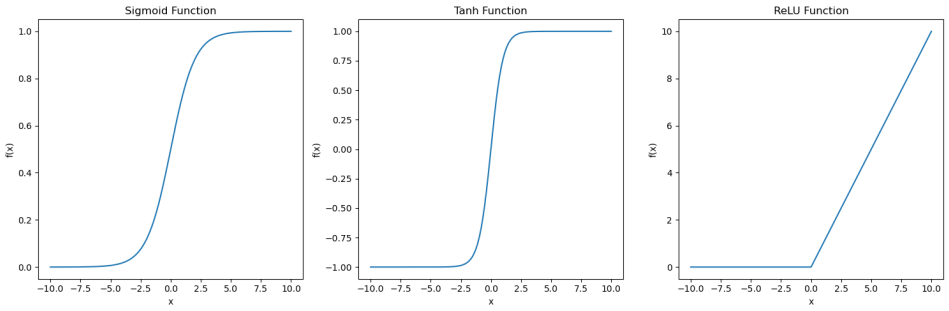

Figure 1: Figure showing common activation functions for Neural Networks, (left to right [Sigmoid, Tanh, ReLU])

2.1.3 Neural Networks

A neural network is a more complex model composed of multiple layers. Each neuron in a layer receives input from the neurons in the previous

layer, applies weights and biases, and passes the result through an activation function to produce an output. This output is then passed on to the neurons in the next layer. Neural networks can have one or more hidden layers between the input and output layers, each with a specified number of nodes, and each node with its own set of weights and biases. Different activation functions can be used on the different layers.

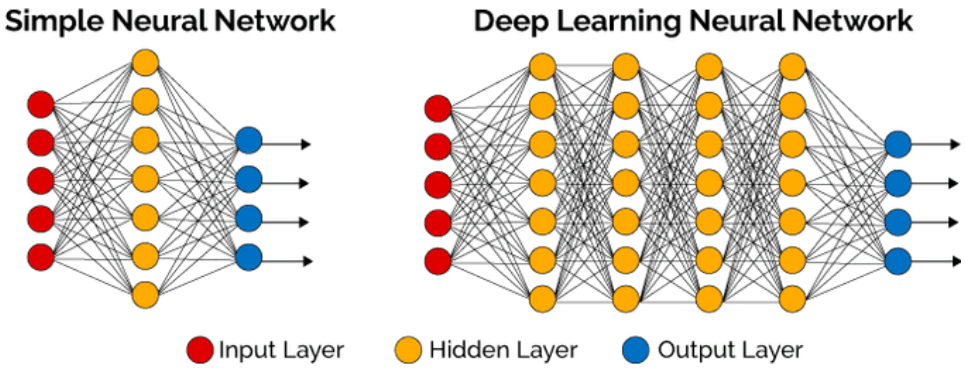

Figure 2: High level view of neural network architectures. Source: [5]

Hidden layers are a key reason why neural networks are able to capture very complex relationships and achieve high performance in many tasks. Deep learning architectures often use many hidden layers to solve complex problems, such as image recognition. Popular models designed by researchers for image recognition tasks can frequently have more than 30 layers [6], with ResNet-152 containing 152 layers [7]. These models can require huge amounts of computing power and resources to train; with ResNet-152 containing 60192808 parameters, all to be optimised [8]. Deep learning image models usually use a convolutional neural network (CNN) architecture, which additionally contain convolutional layers and pooling layers alongside fully connected layers. These are particually useful in computer vision applications since they allow the network to detect local patterns in images. For a more in depth guide, see [9]. For a famous application on classifying handwritten digits, see [10].

2.2 Gradient Based Optimisation

Alongside this increasing complexity of high performance deep neural networks, comes the need for efficient and scalable optimisation algorithms.

2.2.1 Backpropagation

Automatic Differentiation (AD) uses symbolic rules for differentiation which are more accurate than approximations within numerical differentiation. Unlike symbolic differentiation, automatic differentiation evaluates expressions numerically early in the computations rather than carrying out large symbolic computations. In other words, automatic differentiation evaluates derivatives at particular numeric values and does not construct symbolic expressions for derivatives [11]. Backpropagation is a specific case of Reverse mode AD, this propagates derivatives backward from a given output.

Neural Networks naturally lend themselves to backpropagation because of their layered structure, differentiable activation functions, and chain rule of calculus [12]. Being careful to avoid the intricacies of automatic differntiation, backpropagation is a key part of the success of neural networks, enabling very efficient gradient calculations for networks with a large number parameters. This gives rise to the effective use of gradient based optimisation methods such as Gradient Descent. For more information on Automatic Differentiation and Backpropagation, see [13] [14] [15].

2.2.2 Gradient Descent

Batch gradient descent (BGD) computes the gradient of the loss function with respect to θ for the entire training dataset, and makes the update.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t), \tag{6}$$

where the learning rate η determines the size of the steps we take each iteration to reach a local minimum. Too high of a learning rate may mean we step past the local minimum resulting in missing a potentially optimal solution. Too low of a learning rate can also be bad as it may result in slow convergence and increase the likelihood of converging to a suboptimal minimum.

In BGD we must load our entire dataset into memory which is not always possible. We also are unable to feed new data into the model since we must process the whole batch.

```
for i in range(epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

Here we evaluate the gradient with respect to the parameters given all the data, and update our parameters in the direction of the negative gradient with a step size defined by the learning rate. This process is repeated with a predefined number of epochs.

2.2.3 Stochastic Gradient Descent

Stochastic Gradient Descent instead performs a parameter update for each training datapoint, x_i, y_i. In practice, this is usually performed with mini batches of a selected size to find a balance with computational efficiency and convergence properties. Mini batches can also take advantage of parallel processing by processing multiple data points simultaneously before making the parameter update.

$$\theta = \theta - \eta \nabla_\theta J(\theta; x_i; y_i) \tag{7}$$

Since we update for each x_i, y_i, this algorithm is often much faster to converge since BGD will compute the gradient over similar and redundant data before updating the parameters. The frequent updates of SGD with a higher variance gradient estimate can cause the loss function to fluctuate heavily. To reduce the fluctuations and risk of stepping past a minimum, we can slowly decrease the learning rate as training goes on.

```
for i in range(epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

The key difference here is that we randomly sample from the data and update our parameters for each datapoint.

Figure 3 shows the fluctuation of SGD over 5000 iterations, we however note that in comparison to BGD, we only evaluate 32 data points at each iteration so the convergence is much faster. This will only improve as the dataset scales up. Decreasing the learning rate may help reduce fluctuation however may mean the model will converge too slowly with big data. Algorithms such as the adam optimiser or adagrad use techniques of adaptive learning rate to automatically adjust the learning rate during training based on the observed behavior of the loss function.

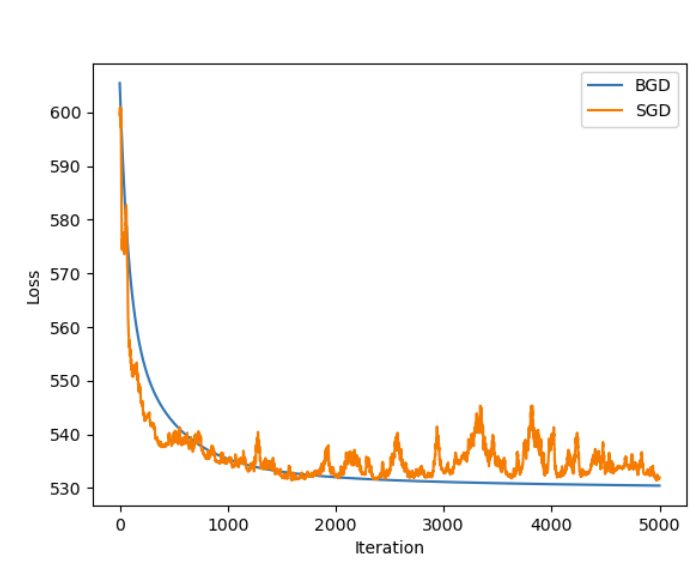

Figure 3: Fluctuation of SGD for Boston Housing dataset [16] prediction model

3 Adaptive Learning Rate Algorithms

Adaptive learning rate methods are an extension to stochastic gradient descent which adapts the learning rate for each parameter based on the historical gradient information. There is a disadvantage of SGD that it scales the gradient uniformly in all directions and keeps a constant learning rate throughout, which may lead to poor performance in our neural network when close to the minima. To address this problem, recent work has proposed a variety of adaptive methods such as scaling the gradient by square roots of some form of the average of the squared values of past gradients. Popular methods include adam optimiser which is widely used in machine learning applications, or AdaGrad [1].

AdaGrad adapts the learning rate for each parameter based on the accumulated squared gradients. The key idea is to increase the learning rate for infrequently updated parameters and decrease it for frequently updated ones [17].

The AdaGrad update rule can be written as:

$$g_t = \nabla L(\theta_t)$$

$$G_t = G_{t-1} + g_t \odot g_t$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

The algorithm maintains a running sum of squared gradients for each model parameter and uses this information to adapt the learning rate for each parameter individually. "$\odot$" refers to the element wise product, which makes it possible to update each parameter with its own learning rate.

Adam is an extension of the ideas from AdaGrad and RMSProp (Root Mean Square Propagation). It uses both the first moment (mean) and the second moment (uncentered variance) of the gradients to adapt the learning rate for each parameter. The key idea is to combine the benefits of both momentum-based methods and adaptive learning rates [18].

The Adam update rule can be written as:

$$g_t = \nabla L(\theta_t)$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \odot \hat{m}_t$$

In this algorithm, β_1 and β_2 are hyperparameters which are used to control the exponential decay rates of the first and second moment estimates respectively. β_1 controls the decay rate for the first moment estimate, which influences the momentum term in the update rule - Momentum helps by accumulating gradients over multiple iterations and using the running average of past gradients to update the model parameters, relating to the idea of momentum in physics. β_2 controls the decay rate for the second moment estimate, relating to the uncentered variance of the gradient. In their 2015 conference paper introducing the Adam optimiser, Kingma, Ba, recommends settings for the tested machine learning problems are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$. These recommended settings are closely followed by implementations in various popular deep learning libraries [19].

Both AdaGrad and Adam optimisers adjust the learning rate for each parameter during the optimisation process. AdaGrad does this by accumulating the squared gradients [3], while Adam extends this idea by considering both the first and second moments of the gradients [20]. These adaptive learning rate methods can lead to faster convergence and improved performance in many optimisation problems.

3.1 Example

In this example, we construct a neural network and train our parameters using SGD, Adam, and AdaGrad. We use the California Housing dataset, where the target is to predict the average house value [21].

Our neural network consists of an input layer, one hidden layer, and an output layer. The input layer takes in the 8 features as input. The input is passed through the first fully connected layer, which computes the weighted sum of the inputs and adds a bias term. The ReLU activation function is then applied and we pass this result through the second fully connected layer to compute the final output, which is a single continuous value representing the predicted house price.

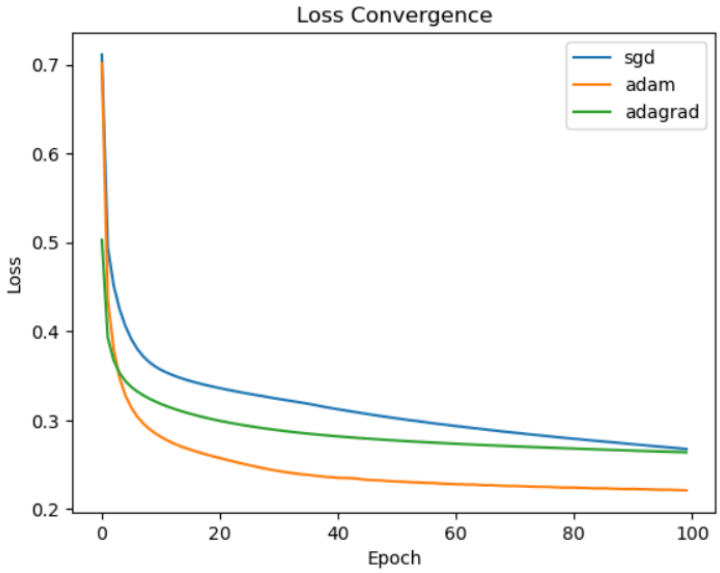

Figure 4: The convergence of the training loss over epochs for each optimizer (SGD, Adam, and Adagrad)

We therefore have many weights and biases in the fully connected layers we want to train to minimise our loss function. We use the Mean Squared Error (MSE) loss. Given n samples, the true target values y_i and the predicted values $\hat{y}_i$, the MSE loss can be written as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

We create identical neural networks for each method of optimisation and plot the results over 10 epochs, with a batch size of 32.

We see from the results that the adam optimiser converges the fastest during training. This could be due to its adaptive learning rate and momentum qualities. The adam optimiser also has the least variance during training, which could be due to the adaptive learning rates for each parameter helping the algorithm to make more fine tuned adjustments.

3.2 Conclusion

When implementing gradient descent methods, we must be careful when setting our learning rate. A constant learning rate may be too small, caus-

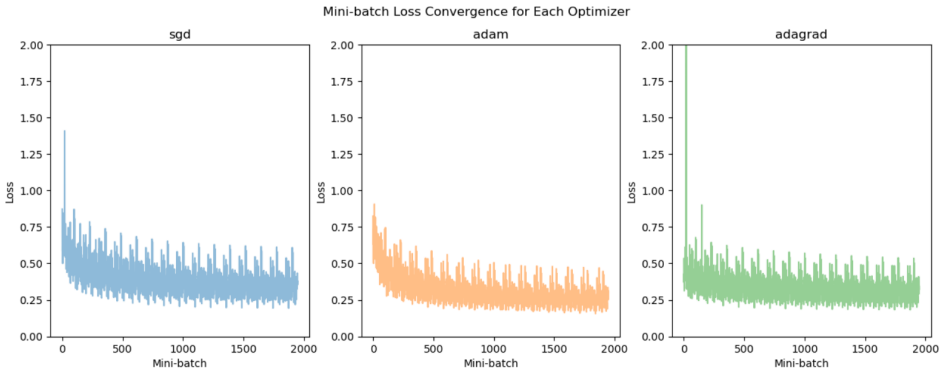

Figure 5: The mini-batch loss for each optimizer (SGD, Adam, and Adagrad) over a selected range of mini-batches.

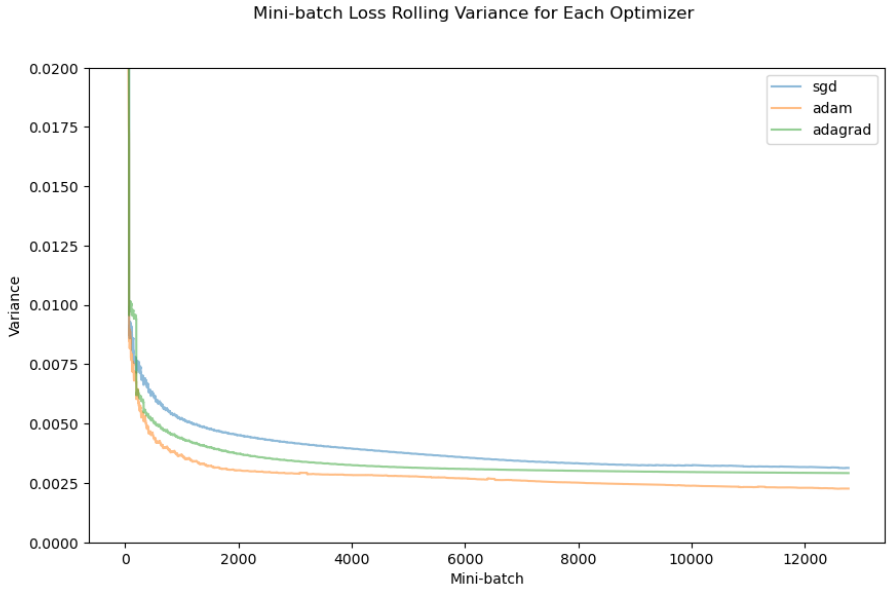

Figure 6: The rolling variance of the mini-batch loss for each optimiser (SGD, Adam, and Adagrad) during training.

ing the optimisation algorithm to take small steps and converge very slowly towards the minimum or too large, causing the optimizer to overshoot the minimum and oscillate around it. Adaptive learning rate methods for SGD offer significant improvements in the optimisation process, leading to faster and more stable convergence in various machine learning tasks. These methods adapt the learning rate for each parameter individually based on their historical gradients allowing for more fine-grained and efficient updates during the training process, leading to faster and more stable convergence.

4 Activation Functions

4.1 Introduction

Activation Functions (AF) are an important element in Neural networks, allowing models to learn abstract features through non-linear transformations. Many different AFs exist, and have found widespread use in models, ranging in computational efficiency, non linearity, and optimisation difficulty [22].

We need activation functions for our hidden layers in the Neural Network since if we were to sum the hidden layers as the weighted sum of the inputs, this would be equivalent to a linear regression model [23] i.e. there would be no non linearity in our model. Linear regression models are often computationally simple but struggle to capture complex, non linear relationships between the features and target. By adding in non linearity, we will be able to much better capture these complex relationships, which is typically our goal with a neural network model.

4.2 Linear Functions

Despite wanting to find complex non linear relationships between our features and targets, we may use a linear AF as part of our network. Linear AFs can be useful for certain types of regression problems, where the output of the network is expected to be a continuous value. Linearity in the activation function means that we can output a wide range of values, which also makes them less sensitive to small changes in the inputs. It may therefore be common to use a linear function in the output layer of a regression problem since the outputs will not be constricted to a certain range like many non linear AF. Used with a combination of non linear AF inside the hidden layers of the network, this will not affect the networks ability to learn complex non linear relationships.

4.3 Non Linear Function

The activation functions themselves influence the network's expressivity, that is, the network's ability to approximate target functions [24]. As we previously mentioned, linear activation functions are only able to approximate linear functions. Non linear functions therefore aim to capture non linear model complexity to approximate non linear targets. We described popular choices in our introduction to neural networks, such as ReLU, logistic, and

Tanh.

4.3.1 Desired Properties

- Continuous and Differentiable: In stochastic gradient descent, we will use the chain rule and backpropagate through the network, training the weights and biases. It is therefore essential for the function to be differentiable with respect to its input. Continuity is a result of the differentiable property [25].
- Bounded: If the activation function is not bounded the output values may explode. A bounded function can therefore be desirable but is not necessary
- Zero-centered: A function is zero centered when its range contains both positive and negative values. If this is not the case, our output will always be always positive or negative. This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights [26].
- Computational complexity: Training a neural network is a computationally expensive exercise, and further computational cost involved in calculating the activation functions can slow down this process further, especially in deep models with a large amount of nodes.

These properties are not essential as we see in our studied and commonly used activation functions, but not meeting this criteria can have certain drawbacks.

4.3.2 Binary step function

The Binary step function is the most simple activation function defined as follows

```
if x<0:
    return 0
else:
    return 1
```

This therefore makes a binary choice if the node is activated or deactivated. This is usually only used for something such as a binary classifier, since it cannot deal with multiple classes. This function is not differentiable, whih runs into issues when backpropagating.

4.3.3 Sigmoid functions

A sigmoid function is a bounded and differentiable function that is non decreasing and has exactly one inflection point. Motivation for the use of sigmoid functions within neural network relates back to biological ideas of neural networks.

Logistic (sigmoid):

$$\text{logistic}(x) = \frac{1}{1 + e^{-x}}. \tag{8}$$

Arctan (Inverse tangent):

$$\arctan(x) = \tan^{-1}(x) \tag{9}$$

Tanh (Hyperbolic tangent):

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{10}$$

Softsign:

$$\text{softsign}(x) = \frac{x}{1 + |x|} \tag{11}$$

The logistic sigmoid maps input values to the (0, 1) interval, while the tanh maps them to the (-1, 1) interval. This boundedness can help with controlling the output values of neurons and stabilising the learning process. This however gives rise to the vanishing gradient problem, a central problem in training neural networks with backpropagation methods.

4.4 Vanishing Gradient Problem

As more layers using certain activation functions are added to neural networks, the gradients of the loss function can approach zero [27]. Certain AFs restrict the input into a small range, such as $(0, 1)$, and repeatedly doing this means that large changes in the input only result in very small changes in the output [28]. This motivates the Vanishing Gradient problem since if our loss function gradient approaches zero, the gradient is too small to effectively train the network since we cannot effectively change the model parameters to make any difference in the network output.

Activation functions which restrict the input into a small range suffer from this problem the most

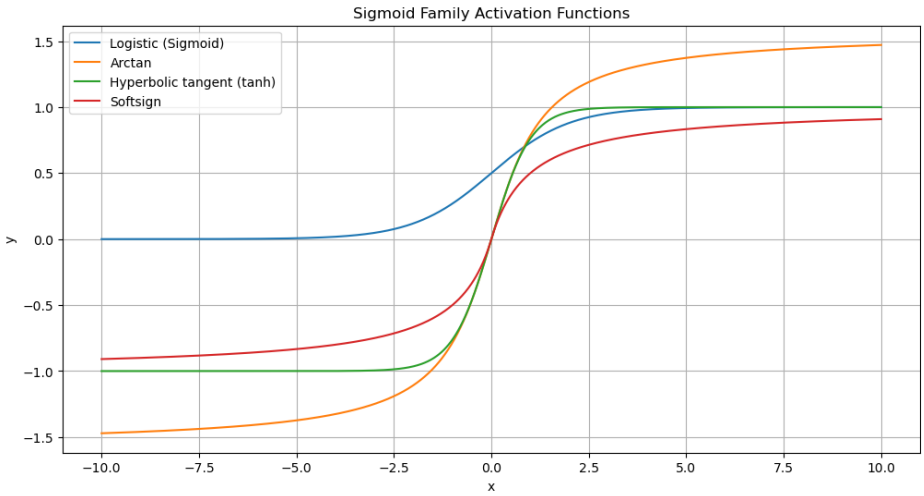

Figure 7: Plot of the sigmoid family activation functions.

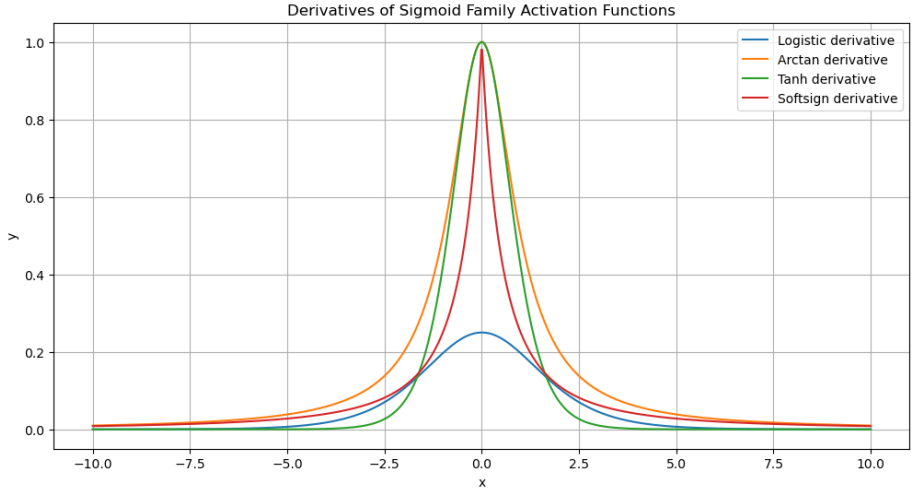

Figure 8: Plot of the derivatives of the sigmoid family activation functions.

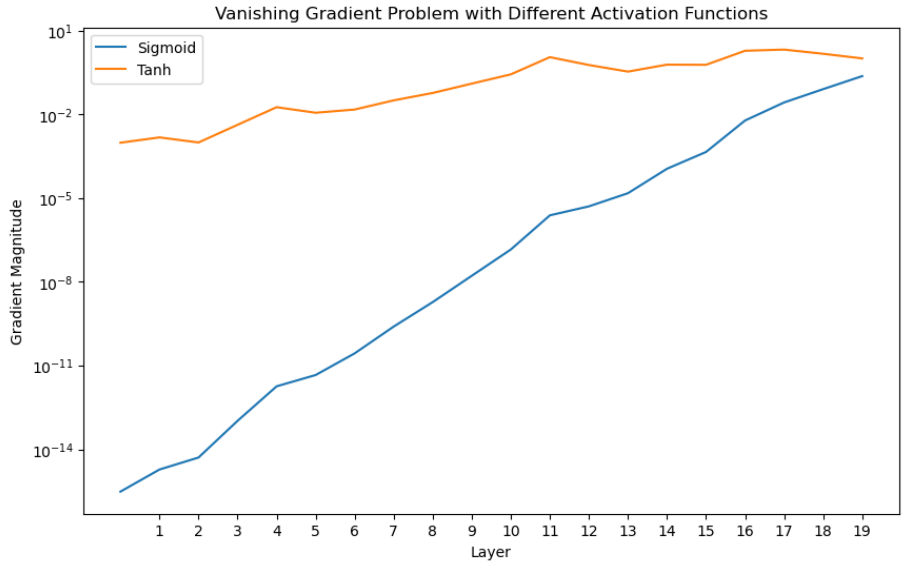

Figure 9: We see that as we backpropagate through the layers of our 20 layer network (right to left), the gradients become very small. Both the Sigmoid and Tanh functions suffer from the vanishing gradient problem. Simple example created with synthetic data.

4.4.1 Solutions

The ReLU activation function is often proposed as a solution for the vanishing gradient problem. The ReLU function is not bounded, so the output is not restricted to within a small range. The ReLU function itself however can be susceptable to the Dying ReLU problem. The Dying ReLU problem happens when our inputs are negative, since the ReLU will output zero. This means the nodes can essentially become inactive. When a large amount of nodes return output zero, the gradients fail to flow during backpropagation, and the weights are not updated [29].

Residual networks are another solution. These add residual connections or skip connections, essentially skipping over layers. Even if the gradients vanish in the network, the skip connections ensure that the gradients can still propagate through the network (see figure 10). This allows us to train a much deeper network (even hundreds of layers [7]) without so much worry on vanishing gradients making the training ineffective.

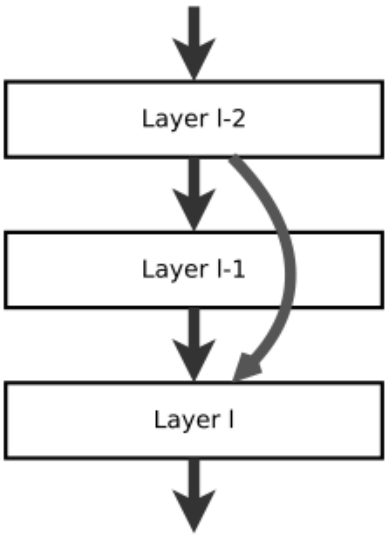

Figure 10: A skip connection in a residual neural network. Source: [30]

5 Reflection

In this paper we have produced an introduction to Neural networks and focused on how we use optimisation to train large models and select activation functions. While we write this paper in the context of training neural networks, the ideas and algorithms reviewed in this paper can be applied to a large amount of problems in data science where we have some gradient computable objective function to minimise or maximise.

SGD is a simple algorithm with immense usefulness due to low cost per iteration, meaning it can be applied to very large problems. However, ideas presented in adaptive learning rate methods such as adam optimiser allow for great improvements. Adaptive learning rates have obvious benefits such as being able to make much larger steps at the beginning of minimisation, and more fine tuned steps when we are much closer to the minimum. Ideas incorporated such as momentum improve our ability to get out of sub-optimal local minima.

In our example, we showed that for a common neural network architecture trained on a widely used dataset, adam optimiser had significantly better performance on ability to find a good minimum, speed of convergence, and the lowest variance during training. In libraries such as pytorch, adam implementation is no additional effort. We might therefore want to use this as a default method for optimisation in neural networks.

Our choice of activation function can be critical to making sure the network is learning effectively. We review the issue of the vanishing gradient problem, and some possible solutions. This gives insight as to how we might choose an appropriate activation function when building a network. We would wish to go further on experimenting with how much impact our choice can make, and how gradient issues can arise.

5.1 Authors note

I feel it was worthwhile to extend the course lecture notes in this direction since it relates heavily to issues such as scalability which we investigate in later chapters. Optimisation methods are a part of the success of the majority of machine learning models. This is genuinely an area that I am looking to keep track of for my future projects when optimisation is slow or the convergence is not as expected. I have always defaulted to ReLU activation functions in my own work, so it was insightful to learn about

desired properties that we would want in an activation function, and how no function is especially perfect satisfying all the conditions. I continue to study famous largely cited networks to find out about how researchers and data science practitioners are designing the architecture to maximise the ability for the network to learn these complex relationships.

References

[1] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," *CoRR*, vol. abs/1902.09843, 2019. [Online]. Available: http://arxiv.org/abs/1902.09843

[2] T. Blain, "Advancements in variational inference: Methods, optimisation techniques, and applications," 2023, Unpublished, 40cp Project.

[3] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[4] Wikipedia contributors, "Stochastic approximation — Wikipedia, the free encyclopedia," 2023, [Online; accessed 8-April-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Stochastic_approximation&oldid=1144917516

[5] deepai, "What is a hidden layer?" [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning

[6] PyTorch, "Pytorch hub for researchers," 2023. [Online]. Available: https://pytorch.org/hub/research-models

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[8] PyTorch, "resnet152," 2015. [Online]. Available: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet152.html

[9] IBM, "What are convolutional neural networks?" [Online]. Available: https://www.ibm.com/topics/convolutional-neural-networks

[10] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 12 1989. [Online]. Available: https://doi.org/10.1162/neco.1989.1.4.541

[11] A. Griewank, "A mathematical view of automatic differentiation," *Acta Numerica*, vol. 12, p. 321–398, 2003.

[12] Hecht-Nielsen, "Theory of the backpropagation neural network," in *International 1989 Joint Conference on Neural Networks*, 1989, pp. 593–605 vol.1.

[13] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul, "Automatic differentiation in machine learning: a survey," *CoRR*, vol. abs/1502.05767, 2015. [Online]. Available: http://arxiv.org/abs/1502.05767

[14] mathworks, "Automatic differentiation background." [Online]. Available: https://uk.mathworks.com/help/deeplearning/ug/deep-learning-with-automatic-differentiation-in-matlab.html

[15] B. v. d. Berg, T. Schrijvers, J. McKinna, and A. Vandenbroucke, "Forward-or reverse-mode automatic differentiation: What's the difference?" *arXiv preprint arXiv:2212.11088*, 2022. [Online]. Available: https://arxiv.org/abs/2212.11088

[16] "Boston housing dataset," 1978. [Online]. Available: https://scikit-learn.org/0.16/modules/generated/sklearn.datasets.load_boston.html

[17] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980

[19] J. Brownlee, "Gentle introduction to the adam optimization algorithm for deep learning," 2017. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[20] DeepAI, "What is the adam optimization algorithm?" [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/adam-machine-learning

[21] "California housing dataset description," 1990. [Online]. Available: https://developers.google.com/machine-learning/crash-course/california-housing-data-description

[22] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," 2022.

[23] S. Singh, "Activation functions for deep neural networks," 2023. [Online]. Available: https://www.knowledgehut.com/blog/data-science/activation-functions-in-neural-networks

[24] J. Lederer, "Activation functions in artificial neural networks: A systematic overview," 2021.

[25] L. Datta, "A survey on activation functions and their relation with xavier and he normal initialization," 2020.

[26] S. CS231n, "Cs231n convolutional neural networks for visual recognition." [Online]. Available: https://cs231n.github.io/neural-networks-1/

[27] DeepAI, "Vanishing gradient problem." [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/vanishing-gradient-problem

[28] C.-F. Wang, "The vanishing gradient problem: The problem, its causes, its significance, and its solutions," 2019. [Online]. Available: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

[29] K. Leung, "The dying relu problem, clearly explained," 2021. [Online]. Available: https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24

[30] Wikipedia contributors, "Residual neural network — Wikipedia, the free encyclopedia," 2023, [Online; accessed 27-April-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Residual_neural_network&oldid=1142090329