F70DB project: SGD and ADAM

Thibault Gasteau

31/03/2022

Contents

Abstract

This project explore stochastic gradient algorithms and ADAM, those popular tools for training deep neural networks. It start with a discussion to understand the connection between deep neural network and stochastic gradient algorithm. Suppose that we are given a training dataset that we want to use to "calibrate" our neural network that is characterised by a set of parameters. To this end, we need to find specific values for the parameters such that the error for the network's output evaluated on the training set is minimized. Solve this minimization is in principle doable by applying classical gradient descent methods from the optimization theory, but in practical applications the datasets are so large that computing all the gradients is prohibitively expensive we will run some simulations in R to show it. That's why after we study the theory of stochastic gradient algorithms and especially using mini-batching method, in which we randomly choose a small subset of the training dataset, in order to reduce the computational cost of the gradient descent algorithm . This algorithm has numerous different variants, depending on how we choose the mini-batch, that's why after we explain how the convergence of our algorithm is depending on assumptions. After having introduced maybe the most popular optimization algorithm we will then introduce ADAM that is really promising , we will then understand its convergence under convexity and talk later about the issues of ADAM, especially the issue of worse generalizaiton than SGD in some cases. We will finish this paper by implementing several optimization algorithms like ADAM or SGD using R.

Acknowledgement

I would like to thank my supervisor, Dr Mateusz Majka at Heriot-Watt University, I am grateful for the guidance, encouragement and advice he has provided throughout the semester. I would like to give special thanks to my tutor Professor John Phillips , that supervised me during those 4 years of university. I would also like to thank the Actuarial Mathematics and Statistics department at Heriot-Watt University and especially thank every teacher, lecturer , professor or doctor that taught me all the courses i followed during my 4 years of education at the university, those 4 years made me able to understand, apprehend and write this project , with the knowledge they instilled in me.

1 Introduction: Deep learning

Deep learning is essentially a neural network with three or more layers. These neural networks attempt to simulate the behavior of the human brain allowing it to “learn” from large amounts of data. While a neural network with a single layer can still make approximate predictions, additional hidden layers can help to optimize and refine for accuracy. Deep learning drives many artificial intelligence applications and services that improve automation, performing analytical

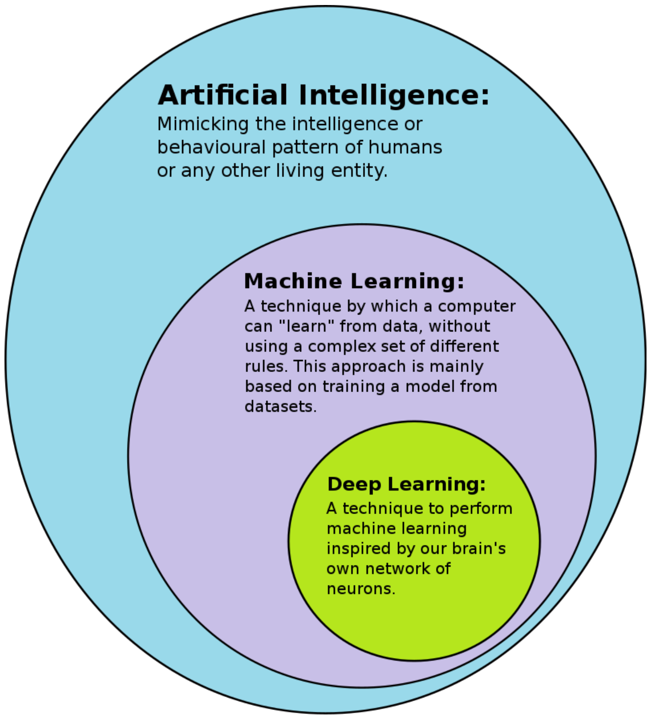

Artificial Intelligence:
Mimicking the intelligence or behavioural pattern of humans or any other living entity.
Machine Learning:
A technique by which a computer can "learn" from data, without using a complex set of different rules. This approach is mainly based on training a model from datasets.
Deep Learning:
A technique to perform machine learning inspired by our brain's own network of neurons.

Figure 1: AI, machine and deep learning: explanation and implication

and physical tasks without human intervention. Deep learning technology lies behind everyday products and services such as digital assistants, voice-enabled TV remotes, and credit card fraud detection as well as emerging technologies such as self-driving cars.

To visualize where deep learning sits in the artificial intelligence big family , we are looking at Figure 1 that can be seen on Deep learning Wikipedia page so we can understand a bit better.

Artificial neural network(ANN) are containing an input layer, one hidden layer, and an output layer. Each artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that neuron is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.

Artificial neural network are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another. The idea behind an artifical neural network is to train a network by choosing the parameters that are the weight and biases to minimize the cost function that is in fact telling how good our neural netowrk is for a certain task, but we will explain it more deeply later.

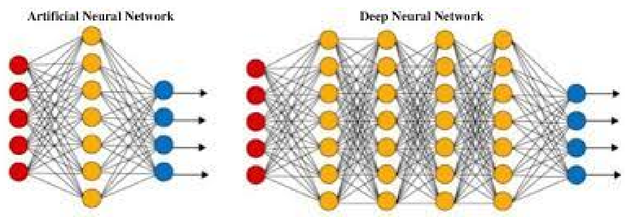


Figure 2: Difference between ANN and DNN

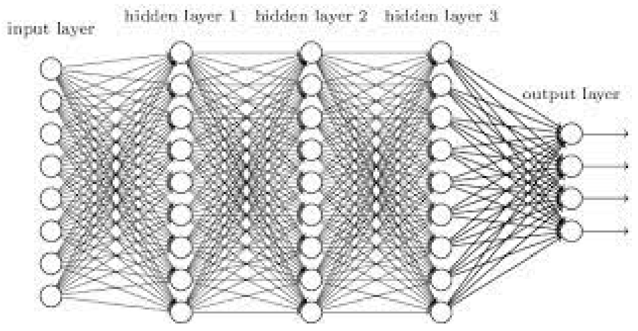


Figure 3: Example of DNN

1.1 Deep neural network

A deep neural network(DNN) is an artificial neural network(ANN) with multiple hidden layers between the input and output layers. Figure 2 from (Mostafa, El-Attar, Abd-Elhafeez, & Awad, 2021) illustrate it and shows the difference between ANN and DNN . There are different types of neural networks but they always consist of the same components: neurons and synapses, it is the connection between neurons, weights, biases, and activation functions.

Between those visible input and output layers there is hidden layers that are training the data sets by choosing an optimal parameter. Then we applied the neural network to new data sets which is called generalisation. The weight is a matrix and the bias is a vector that we multiply giving a single vector v, and we will explain why soon. Parameters are picked randomly until the error which is equal to the desired output subtracted by the actual output gets better. This error will always be superior to 0. Also the activation function here is a sigmoid function , it smooths out the output of the neural network , a continuous function is better than a discrete one, but we will explain why in our first part.

Figure 3 coming from (Nielsen, 2015)is an example of a deep neural network with 5 layers and so 3 hidden layers.

1.2 Train a neural network

The problem of training a neural network is equivalent to ask us the problem of minimizing the cost function(also called loss function or objective function). When we start off with our neural network we initialize our weights randomly. Obviously, it won't give us very good results. In the process of training, we want to start with a bad performing neural network and wind up with network with high accuracy. Improving the network is possible, because we can change its function by adjusting weights. We want to find another function that performs better than the initial one. There are a lot of algorithms that optimize functions. These algorithms can be gradient-based or not, in sense that they are not only using the information provided by the function, but also by its gradient. A gradient is a differential operator that, operating upon a function of several variables, results in a vector, the coordinates that are the partial derivatives of the function.We know it can maybe seems a bit confusing right now but we will come back to it soon. Then, one of the gradient-based algorithms and the one we will be covering in this article is called Stochastic Gradient Descent or Stochastic gradient algorithm.

1.3 Stochastic gradient algorithm : overview

Stochastic gradient algorithm is a method to find the optimal parameter configuration for a machine learning algorithm. It iteratively makes small adjustments to a machine learning network configuration to decrease the error of the network. Instead of decreasing the error like deep neural network , or finding the gradient for the entire data set, this method merely decreases the error by approximating the gradient for a randomly selected batch (which may be as small as single training sample). In practice, the random selection is achieved by randomly shuffling the dataset and working through mini-batches in a stepwise fashion. It's really important not to confuse stochastic gradient descent and gradient descent. Anyway, we will explain both in this article.

Part I
Connection between deep neural network and stochastic gradient algorithm

Neural networks are a biologically-inspired programming paradigm which enables a computer to learn from observational data. Deep learning is a powerful set of techniques for learning in neural networks. Neural networks and deep learning currently provide the best solutions to many problems in image recognition, speech recognition, and natural language processing.

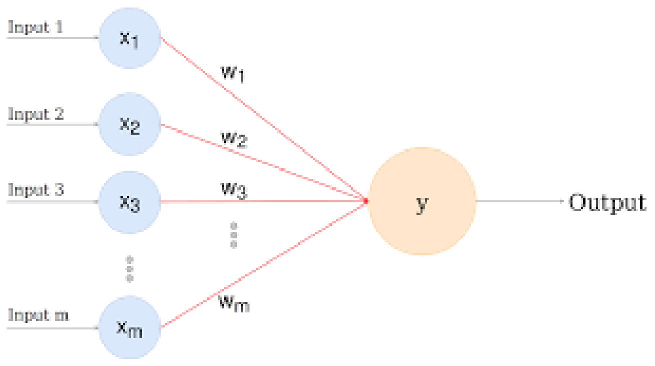


Figure 4: A perceptron

2 Perceptron

To explain deep neural networks we will start by explaining a type of artificial neuron called a perceptron. Today, it's more common to use other models of artificial neurons, usually the main neuron model used is one called the sigmoid neuron. A perceptron takes several binary inputs, x_1,x_2,...,x_n and produces a single binary output. Perceptron was invented by Frank Rosenblatt in 1957. He introduced weights, w_1,w_2,...,w_n, that are real numbers expressing the importance of the respective inputs to the output. The neuron's output is either 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. We illustrate a perceptron in Figure 4.

To put it in algebraic terms:

$$output = \begin{cases} 0, & \text{if } \sum_j \ w_j \ x_j \leq \text{thresold} \\ 1, & \text{if } \sum_j \ w_j \ x_j > \text{thresold} \end{cases} \tag{1}$$

A perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions.

In Figure 5 there is an example of a Multi-Layer Perceptron coming from (Pérez-Enciso & Zingaretti, 2019) with details of neurons and synapses.

In this network, the first column of perceptrons called the first layer of perceptrons is making five very simple decisions, by weighing the input evidence. In the second layer each of those perceptrons are making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third,fourth,fifth and sixth layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

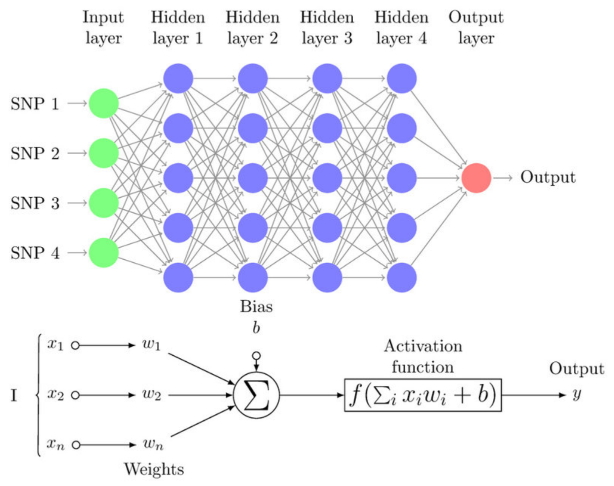


Figure 5: A perceptron with 6 layers

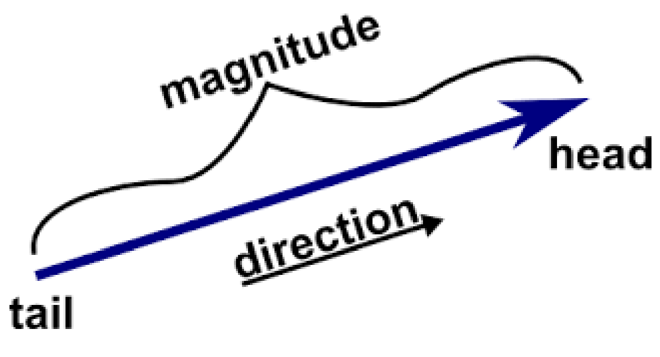


Figure 6: A vector

We then simplify the way we describe perceptrons. The condition $\sum_j w_j x_j$ > threshold is annoying, and we can make two notational changes to simplify it. The first change is to write $\sum_j w_j x_j >$ threshold as a dot product, w·x≡$\sum_j w_j x_j$, where w and x are vectors(illustrated in Figure 6 from(D & DQ, n.d.)) whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's bias, b ≡ - threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$output = \begin{cases} 0, & \text{if w·x +b} \leq 0 \\ 1, & \text{if w·x +b} > 0 \end{cases} \tag{2}$$

We can think of the bias as a measure of how easy it is to get the perceptron to output a 1. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1. Now that we have introduce the bias we will always use it and not the threshold anymore.

We can use networks of perceptrons to compute any logical function at all.

In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way which is a problem for our neural network, that is the main problem about discrete functions.

That's why we now use a continous function with sigmoid neurons.

3 Sigmoid neurons

Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Just like a perceptron, the sigmoid neuron has inputs, x_1,x_2,. . .,x_n But instead of being just 0 or 1 these inputs can also take on any values between 0 and 1, so for example 0.88 is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w_1,w_2,. . .,w_n and an overall bias, b, but the output is not 0 or 1. Instead, it's σ(w·x+b), where σ is called the sigmoid function , illusrated below and is defined by:

$$\sigma(z) \equiv \frac{1}{(1+e^{-z})} \tag{3}$$

To put it in a more explicit way, the output of a sigmoid neuron with inputs x_1,x_2,. . .,x_n weights w_1,w_2,. . .,w_n and bias b is

$$\frac{1}{1+\exp\left(-\sum_j w_j x_j - b\right)} \tag{4}$$

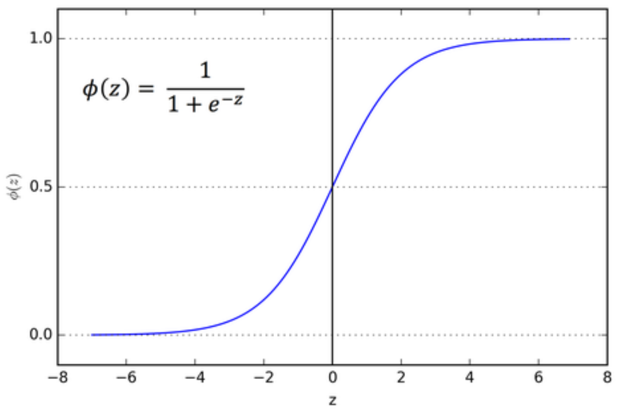


Figure 7: A sigmoid function

In Figure 7 from (Sharma, Sharma, & Athaiya, 2017) we can see a sigmoid function.

It's only when w·x+b is of modest size that there's a big deviation between the sigmoid and the perceptron model.

By using the actual σ function we get a smoothed out perceptron. Indeed, it's the smoothness of the σ function that is the crucial fact. The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron. In fact, calculus tells us that Δoutput is well approximated by:

$$\Delta output \approx \sum_j \frac{\partial output}{\partial w_j} \Delta w_j + \frac{\partial output}{\partial b} \Delta b \tag{5}$$

It's actually saying something very simple : Δoutput is a linear function of the changes Δw_j and Δb in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output. So while sigmoid neurons have much of the same qualitative behaviour as perceptrons, they make it much easier to figure out how changing the weights and biases will change the output.

Now that we have explained sigmoid neurons , we will look at the architecture of neural networks.

4 Architecture of neural networks

As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons. The rightmost or output layer contains the output neurons. The middle layer is called a hidden layer, since the neurons in this layer are neither inputs nor outputs. Networks can have multiple hidden layers.

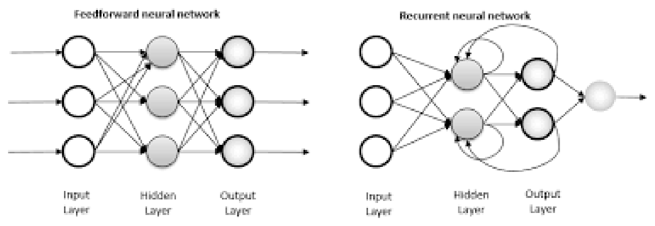


Figure 8: Feedforward and recurrent neural networks

Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called feedforward neural networks. This means there are no loops in the network , information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σ function depended on the output. That'd be hard to make sense of, and so that's why we didn't allow such loops.

However, there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

Figure 8 that can be seen in (Pekel & Soner Kara, 2017)illustrate it.

Recurrent neural networks have been less influential than feedforward networks, in part because the learning algorithms for recurrent neural networks are less powerful. But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks. And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks. Despite the fact that we are not using recurrent neural networks , it's still important to be aware of their existence to understand neural networks.

We have now explain properly deep neural networks, we will now talk about the connection between DNN and stochastic gradient descent to train them.

5 Cost function and gradient descent

Training a network corresponds to choosing the parameters, that are the weights and biases, that minimize the cost function. The weights and biases take the form of matrices and vectors, but at this stage it is convenient to imagine them stored as a single vector that we call v.

We now want an algorithm that let us find weights and biases so that the output from the network approximates y(x) that is the desired output, for all training inputs x.

To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{6}$$

Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x. Of course, the output a depends on x, w and b. The notation $\|v\|$ just denotes the usual length function for a vector v. We call C the quadratic cost function, it's also known as the mean squared error or MSE.

Looking at the form of the quadratic cost function, we see that $C(w, b)$ is non-negative, since every term in the sum is non-negative. The cost $C(w, b)$ becomes small, $C(w, b)\approx 0$, precisely when $y(x)$ is approximately equal to the output, a, for all training inputs, x. So our training algorithm is supposed to find weights and biases so that $C(w, b) \approx 0$. If $C(w, b)$ is large, that would mean that $y(x)$ is not close to the output a for a large number of inputs. So the aim of our training algorithm will be to minimize the cost $C(w, b)$ as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as gradient descent but we will show this algorithm limits and explain an other algorithm, stochastic gradient algorithm later.

Using a smooth cost function like the quadratic cost turns out to be easy to figure out how to make small changes in the weights and biases to get an improvement in the cost. That's why we focus first on minimizing the quadratic cost, and only after that we will examine the accuracy.

So for now we're going to forget all about the specific form of the cost function, the connection to neural networks, and so on. Instead, we're going to imagine that we've simply been given a function of many variables and we want to minimize that function. We're going to develop a technique called gradient descent which can be used to solve such minimization problems. Then we'll come back to the specific function we want to minimize for neural networks.

Let's suppose we're trying to minimize some function, $C(v)$. This could be any real-valued function of many variables, $v = v_1$,v_2,$\ldots v_n$. Note that we've replaced the w and b notation by v to emphasize that this could be any function.

What we'd like is to find where C achieves its global minimum. A general function, C, may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum like we could do with just few variables.

One way of attacking the problem is to use calculus to try to find the minimum analytically. We could compute derivatives and then try using them to find places where C is an extremum. That might work when C is a function of one or a few variables, but it'll turn into a nightmare when we have many more variables, and for neural networks we'll often want far more variables. The biggest neural networks have cost functions that depend on billions of weights and biases in an extremely complicated way. Using calculus to minimize that just won't work.

So calculus doesn't work. Fortunately, there is a beautiful analogy that suggests an algorithm that works pretty well. We start by thinking of our function as a kind of a valley. We imagine a ball rolling down the slope of the valley, our experience and logic will tells us that the ball will probably roll to the bottom of the valley. We use this idea as a way to find a minimum for the function, we'd randomly choose a starting point for an imaginary ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley. We could do this simulation simply by computing derivatives of C , those derivatives would tell us everything we need to know about the local "shape" of the valley, and therefore how our ball should roll.

C is a function of many variables. Suppose that C is a function of m variables, v_1,. . . ,v_m. Then the change ΔC in C produced by a small change :

$$\Delta v = (\Delta v_1, ..., \Delta v_m)^T \tag{7}$$

is:

$$\Delta C \approx \nabla C \cdot \Delta v \tag{8}$$

where ∇C is called the the gradient vector

$$\nabla C \equiv (\partial C/\partial v_1, \dots, \partial C/\partial v_m)^T \tag{9}$$

And so we choose:

$$\Delta v = -\eta \nabla C \tag{10}$$

where η is a small, positive parameter known as the learning rate and so we're guaranteed that our approximate expression for ΔC will be negative. This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule:

$$v \rightarrow v - \eta \nabla C \tag{11}$$

We can think of this update rule as defining the gradient descent algorithm. It gives us a way of repeatedly changing the position v in order to find a minimum of the function C. The rule doesn't always work - several things can go wrong and prevent gradient descent from finding the global minimum of C.

To make gradient descent work correctly, we need to choose the learning rate η to be small enough so that our equation is a good approximation. If we don't, we might end up with $\Delta C > 0$, which obviously would not be good! At the same time, we don't want η to be too small, since that will make the change Δv tiny, and thus the gradient descent algorithm will work very slowly. In practical implementations, η is often varied so that our equation remains a good approximation, but the algorithm isn't too slow.

But, in practice gradient descent often works extremely well, and in neural networks we'll find that it's a powerful way of minimizing the cost function, and so helping the neural network learn.

Indeed, there's even a sense in which gradient descent is the optimal strategy for searching for a minimum. Let's suppose that we're trying to make a move Δv in position to decrease C as much as possible, gradient descent can be viewed as a way of taking small steps in the direction that does the most to immediately decrease C.

We now apply gradient descent to a neural network so that it can learn. The idea is to use gradient descent to find the weights w_k and biases b_l that minimize the cost in our equation . To see how this works, let's restate the gradient descent update rule, with the weights and biases replacing the variables v_j. In other words, our "position" now has components w_k and b_l, and the gradient vector ∇C has corresponding components $\frac{\partial C}{\partial w_k}$ and $\frac{\partial C}{\partial b_l}$. Writing out the gradient descent update rule in terms of components, we have:

$$w_k \rightarrow w_k - \eta \frac{\partial C}{\partial w_k} \tag{12}$$

$$b_l \rightarrow = b_l - \eta \frac{\partial C}{\partial b_l} \tag{13}$$

These equations defines the gradient descent or steepest descent method. We choose an initial vector and iterate with the equation until some stopping criterion has been met, or until the number of iterations has exceeded the computational budget.

By repeatedly applying this update rule we can finally roll down the valley, and hopefully find a minimum of the cost function. In other words, this is a rule which can be used to learn in a neural network.

For now I just want to mention one problem. To understand what the problem is, let's look back at the quadratic cost equation . In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

An idea called stochastic gradient algorithm can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs. By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning.

We now need to introduce the idea of convexity. An important property of a convex function is that every local minimum of the function is also a global minimum.

Another important property of convex functions is that for every w we can construct a tangent to f at w that lies below f everywhere. If f is differentiable, this tangent is the linear function l(u) = f(w) + $<$ ∇f(w), $u - w >$, where ∇f(w)

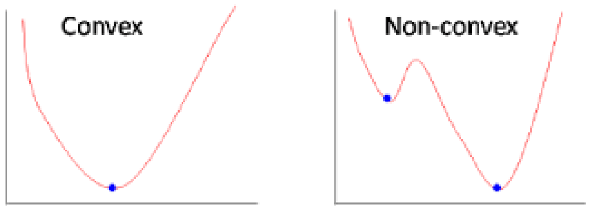


Figure 9: Convex and nonconvex functions

is the gradient of f at w, namely, the vector of partial derivatives of f, ∇f(w) =

$$(\frac{\partial f(w)}{\partial w_1}, ..., \frac{\partial f(w)}{\partial w_d}) \tag{14}$$

For convex differentiable functions, ∀u, f(u) ≥ f(w) + < ∇f(w), u - w >.

Figure 9 from (He, Rexford, & Chiang, 2010) shows the difference between convex and nonconvex functions

We now will define convexity using (Shalev-Shwartz & Ben-David, 2014) theorem.

Convex set : A set C in a vector space is convex if for any two vectors u, v in C, the line segment between u and v is contained in C. That is, for any $\alpha \in$ [0, 1] we have that αu + (1 −α)v ∈ C.

Convex Function: Let C be a convex set. A function f : C → R is convex if for every u, v ∈ C and $\alpha \in$ [0, 1], f(αu + (1 −α)v) ≤ αf(u) + (1 −α)f(v) .

In words, f is convex if for any u, v, the graph of f between u and v lies below the line segment joining f(u) and f(v).

We now describe the standard gradient descent approach to minimize a differentiable convex function f(w) clearly.

The gradient of a differentiable function f : R^d → R at w, denoted ∇f(w), is the vector of partial derivatives of f, namely,

$$\nabla f(w) = (\frac{\partial f(w)}{\partial f w_1}, ..., \frac{\partial f(w)}{\partial f w_d}) \tag{15}$$

Gradient descent is an iterative algorithm. We start with an initial value of w, for example $w^{(1)} = 0$ and then, at each iteration, we take a step in the direction of the negative of the gradient at the current point. That is the update step :

$$w^{(t+1)} = w^{(t)} - \eta \nabla f(w^{(t)}) \tag{16}$$

where $\eta > 0$ is called the step size or leaning rate.

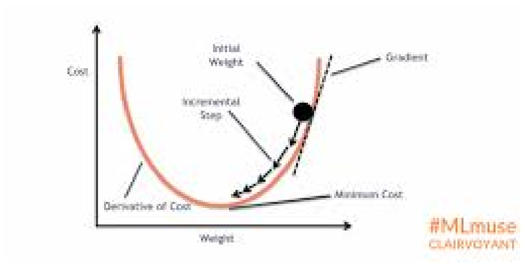


Figure 10: Illustration of gradient descent

Intuitively, since the gradient points is in the direction of the greatest rate of increase of f around $w^{(t)}$, the algorithm makes a small step in the opposite direction, thus decreasing the value of the function.

After T iterations, the algorithm outputs the averaged vector:

$$\overline{w} = \frac{1}{T} \sum_{t=1}^{T} w^{(t)} \tag{17}$$

.

The output could also be the last vector, $w^{(T)}$, or the best performing vector, $argmin_t \in [T]$ f$w^{(t)}$. But taking the average turns out to be rather useful, especially when we generalize gradient descent to nondifferentiable functions and to the stochastic case (that is what we are interested in this article).

There is an illustration in Figure 10 to understand how gradient descent is working.

6 Example of a gradient descent algorithm in R

We then are implementing a gradient descent algorithm in R. Let's get some brief words about our dataset. I choose a dataset on kaggle containing retail e-commerce orders dataset from Pakistan. It contains half a million transaction records from March 2016 to August 2018. The data was collected from various e-commerce merchants as part of a research study

```
df <- read.csv("C:\\Users\\User\\Documents\\ecom.csv")
v1 <- df[[6]]
v2 <- df[[7]]
plot(v1,v2, xlab= "quantity ordered" , ylab = "price total", main = "Plot of the quantity or
model <- lm(v2 ~ v1, data = df)
coef(model)
y_preds <- predict(model)
abline(model)
```

```
errors <- unname((v2 - y_preds) ^ 2)
MSE <-sum(errors) / length(v2)

gradientDesc <- function(x, y, learn_rate, conv_threshold, n, max_iter) {
plot(v1, v2, col = "blue", pch = 20)
m <- runif(1, 0, 1)
c <- runif(1, 0, 1)
yhat <- m * x + c
MSE2 <- sum((y - yhat) ^ 2) / n
converged = F
iterations = 0
while(converged == F) {
## Implement the gradient descent algorithm
m_new <- m - learn_rate * ((1 / n) * (sum((yhat - y) * x)))
c_new <- c - learn_rate * ((1 / n) * (sum(yhat - y)))
m <- m_new
c <- c_new
yhat <- m * x + c
MSE_new <- sum((y - yhat) ^ 2) / n
if(MSE - MSE_new <= conv_threshold) {
abline(c, m)
converged = T
return(paste("Optimal intercept:", c, "Optimal slope:", m))
}
iterations = iterations + 1
if(iterations > max_iter) {
abline(c, m)
converged = T
return(paste("Optimal intercept:", c, "Optimal slope:", m))
}
}
}
# Run the function
gradientDesc(x, y, learn_rate, conv_threshold, n, max_iter)
```

In our dataset we will study the quantity ordered against the total price paid in Pakistani Rupee in e-commerce orders from Pakistan from March 2016 to August 2018. Linear regression is a classic supervised statistical technique for predictive modelling which is based on the linear hypothesis: $y = mx + c$ where y is the response or outcome variable, m is the gradient of the linear trend-line, x is the predictor variable and c is the intercept. The intercept is the point on the y-axis where the value of the predictor x is zero.

In order to apply the linear hypothesis to a dataset with the end aim of modelling the situation under investigation, there needs to be a linear relationship between the variables in question.

We load our dataset with $n = 499998$ numbers. We extract the 2 columns

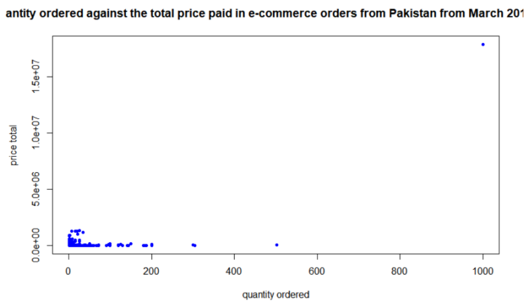


Figure 11: Scatterplot of the 2 variables

we are interested in from dataframe

We then do a scatterplot of these 2 variables(Figure 11).

In figure 11 , there is an obvious outlier value but even if we exclude the value, the problem is the same after, we find another outlier value and it is like this for the whole dataset. Our goal is to reduce the value of the MSE . We then creates a linear model from which we can extract the intercept and slope that are the coefficients. We extract those coefficients:

(Intercept) -10001.58 , v1 13781.38

The negative intercept tells us where the linear model predicts revenue (y) would be when subs (x) is 0.

We see that the intercept is set at -10001.58 on the y-axis and that the gradient is 13781.38 . The positive gradient tells us that there is a positive relationship between v2(price total) and v1(quantity ordered) with one unit increase in v1 resulting in a 13781.38 pakistani rupee increase in v2.

We then represents the predicted y value from the linear model after plugging in values for the intercept and slope while y represents the observed value of the response variable. We plot the predicted values while we plot y predictions on the scatterplot as a regression line.

We then calculate the MSE: The MSE is : 1142623402

We now have the MSE.

We then implement the Gradient Descent Algorithm to find the optimal intercept and gradient which correspond to the lowest possible MSE .

The goal of the algorithm is to find the intercept and gradient values which correspond to the lowest possible MSE. It achieves this through iteration over each set of xy data pairs whereby new intercept and gradient values are calculated as well as a new MSE. Then, the new MSE is subtracted from the old MSE and, if the difference is negligible, the optimal values are found.

We now explain the function created below , we pass the function our x and y variables, the learning rate which is the magnitude of the steps the algorithm takes along the slope of the MSE function, this is set arbitrarily, the convergence

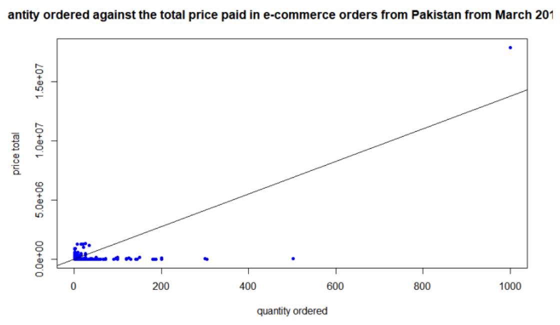


Figure 12: Plot of the predicted values

threshold is the difference between the old MSE and new MSE on each iteration, we pass the value for n and we set the maximum number of iterations we wish to carry out before the loop terminates.

We know that in linear regression y=mx+c, where y is the response variable, m is the gradient of the linear trend-line, x is the predictor variable and c is the intercept. We know that m and c are following uniform distribution and runif generates random deviates. yhat predicted value of y in the regression equation. We then set up the MSE. We then define inital set up for convergence and iterations, of course we say that our algorithm does not converge otherwise we can't implement the gradient descent algorithm. As our algorithm is implemented we have some new m and c values and a new MSE value. The convergence thresold is the difference between the old MSE and new MSE on each iteration. If it's superior or equal to 0 it means that it converge. If the algorithm does converge we return the last c and m value that are then the optimal intercept and slope but the algorithm can also stop because it reached the maximum number of iteration . In this situation we would choose the last c and m value too.

We then want to run the function but as our algorithm is a gradient descent one, we can't compute every single data unfortunatelly.

We now introduce stochastic gradient algorithm.

Part II
Theory of Stochastic gradient algorithm

In stochastic gradient algorithm we do not require the update direction to be based exactly on the gradient. Instead, we allow the direction to be a random vector and only require that its expected value at each iteration will equal the gradient direction.

Stochastic Gradient Algorithm (SGA) for minimizing f(w):

```
parameters: Scalar $\eta > 0$, integer T > 0
initialize: $w^{(1)} = 0$
for t = 1, 2, ... , T
choose $v_t$ at random from a distribution such that $E[v_t| w^{(t)}] \in \partial f(w^{(t)})$
update $w^{(t+1)} = w^{(t)} - \eta v_t$
output $\bar{w} = \frac{1}{T} \sum_{t=1}^{T} w^{(t)}$
```

The advantage of SGA is that SGA is an efficient algorithm that can be implemented in a few lines of code and is simple.

7 SGA: sampling with replacement

When we have a large number of parameters and a large number of training points, computing the gradient vector at every iteration of the gradient descent method can be prohibitively expensive. A much cheaper alternative is to replace the mean of the individual gradients over all training points by the gradient at a single, randomly chosen, training point. This leads to the simplest form of what is called the stochastic gradient method. A single step may be summarized as
1. Choose an integer i uniformly at random from 1, 2, 3, . . . , N.
2. Update:

$$v \to v - \eta \nabla C_{x^i}(v) \tag{18}$$

In words, at each step, the stochastic gradient method uses one randomly chosen training point to represent the full training set. As the iteration proceeds, the method sees more training points. So there is some hope that this dramatic reduction in cost-per-iteration will be worthwhile overall. We note that, even for very small η, the update is not guaranteed to reduce the overall cost function, we have traded the mean for a single sample. Also , the phrase stochastic gradient descent is widely used, we prefer to use stochastic gradient algorithm here. The version of the stochastic gradient method that we introduced is the simplest from a large range of possibilities. In particular, the index i was chosen by sampling with replacement after using a training point, it is returned to the training set and is just as likely as any other point to be chosen at the next step.

8 SGA: sampling without replacement

An alternative is to sample without replacement: that is, to cycle through each of the N training points in a random order. Performing N steps in this manner is referred to as completing an epoch and may be summarized as follows:

1. Shuffle the integers 1, 2, 3, . . . , N into a new order, k_1, k_2, . . . , k_N .
2. for i = 1 upto N, update

$$v \rightarrow v - \eta \nabla C_{x^{ki}}(v) \tag{19}$$

But if we regard the stochastic gradient method as approximating the mean over all training points by a single sample, then it is natural to consider a compromise where we use a small sample average. That's why we now introduce mini-batch.

9 Mini-batch

For some m << N we could take steps of the following form.

1. Choose m integers, k_1, k_2, ... , k_m, uniformly at random from 1, 2, 3, ... , N.
2. Update

$$v \rightarrow v - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C_{x^{ki}}(v) \tag{20}$$

In this iteration , the set x^{ki} from i = 1 to n is known as a mini-batch. There is a without replacement alternative where, assuming N = Km for some K, we split the training set randomly into K distinct mini-batches and cycle through them. Because the stochastic gradient method is usually implemented within the context of a very large scale computation, algorithmic choices such as minibatch size and the form of randomization are often driven by the requirements of high performance computing architectures. Also, it is, of course, possible to vary these choices, along with others, such as the learning rate, dynamically as the training progresses in an attempt to accelerate convergence, that's what we will explain in next part.

To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We'll label those random training inputs X_1,X_2,...,X_m, and refer to them as a mini-batch.

Provided the sample size m is large enough we expect that the average value of the ∇CX_j will be roughly equal to the average over all ∇C_x, numerically it means that :

$$\frac{\sum_{j=1}^{m} \nabla C X_j}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{21}$$

where the second sum is over the entire set of training data. Swapping sides we obtain:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C X_j \tag{22}$$

Confirming that we can estimate the overall gradient by computing gradients just for the randomly chosen mini-batch.

To connect this explicitly to learning in neural networks, suppose w_k and b_l denote the weights and biases in our neural network. Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training like this:

$$w_k \rightarrow w_k - \frac{\eta}{m} \sum_j \frac{\partial C X_j}{\partial w_k} \tag{23}$$

$$b_l \rightarrow b_l - \frac{\eta}{m} \sum_j \frac{\partial C X_j}{\partial b_l} \tag{24}$$

Where the sums are over all the training examples X_j in the current mini-batch. Then we pick out another randomly chosen mini-batch and train with those. And so on, until we've exhausted the training inputs, which is said to complete an epoch of training. At that point we start over with a new training epoch.

We can think of stochastic gradient descent as being like political polling: it's much easier to sample a small mini-batch than it is to apply gradient descent to the full batch, just as carrying out a poll is easier than running a full election. For example, if we have a training set of size n=100,000 and choose a mini-batch size of m=20, this means we'll get a factor of 5,000 speedup in estimating the gradient. Of course, the estimate won't be perfect, there will be statistical fluctuations , but it doesn't need to be perfect: all we really care about is moving in a general direction that will help decrease C, and that means we don't need an exact computation of the gradient. In practice, stochastic gradient descent is a commonly used and powerful technique for learning in neural networks, and it's the basis for most of the learning techniques.

Part III
Different convergence depending on assumptions

We have seen in part I and part II that we were implementing SGD with mini-batch to accelerate the convergence of our algorithm. We will now see assumptions that can change the convergence time of our algortihm .

10 Analysis of SGD for Convex-Lipschitz-Bounded Functions

We have already defined convexity , we now need to define Lipschitz function.

Lipschitzness: Let C $\in R^d$
A function f : $R^d \rightarrow R^k$ is ρ-Lipschitz over C if for every w_1, $w_2 \in$ C we have that $||f(w1) - f(w2)|| \leq \rho||w1 - w2||$.

Intuitively, a Lipschitz function cannot change too fast. Then in Theorem 1:
Let B, $\rho > 0$. Let f be a convex function and let w* $\in argmin_w : \|w\| \leq B$ f(w). Assume that SGD is run for T iterations with $\eta = \frac{sqrt(B^2)}{\rho^2 T}$
Assume also that for all t,$\|v_t\| \leq \rho$ with probability 1.
Then, E [f($\bar{w}$)] - f(w*) $\leq \frac{B\rho}{\sqrt{T}}$

Therefore, for any ϵ> 0, to achieve E [f($\bar{w}$)] - f(w*) $\leq \epsilon$, it suffices to run the SGA algorithm for a number of iterations that satisfies T $\geq \frac{B^2\rho^2}{\epsilon^2}$ But we have seen something interesting with this theorem . If we assume that our function is Convex-Lipschitz-Bounded we need to run the algorithm T $\geq \frac{B^2\rho^2}{\epsilon^2}$ time. We now describe several variants of Stochastic Gradient Algorithm.

11 Variable Step Size

Another variant of SGA is decreasing the step size as a function of t. That is, rather than updating with a constant η, we use η_t. For instance, we can set:

$$\eta_t = \frac{B}{\rho\sqrt{t}} \tag{25}$$

And it achieves a bound, the idea is that when we are closer to the minimum of the function, we take our steps more carefully, so as not to "overshoot" the minimum. It doesn't accelerate the convergence but increase the accuracy of our algorithm.

12 Other Averaging Techniques

We have set the output vector to be $\bar{w} = \frac{1}{T} \sum_{t=1}^{T} w^{(t)}$

There are alternative approaches such as outputting $w^{(t)}$ for some random t $\in$ [t], or outputting the average of $w^{(t)}$ over the last αT iterations, for some α $\in$ (0, 1). One can also take a weighted average of the last few iterates. These more sophisticated averaging schemes can improve the convergence speed in some situations, such as in the case of strongly convex functions defined in the following

13 Strongly convex functions

In this section we show a variant of SGA that enjoys a faster convergence rate for problems in which the objective function is strongly convex. We use the definition from(Shalev-Shwartz & Ben-David, 2014).

A function f is λ-strongly convex if for all w, u and $\alpha \in (0, 1)$ we have:
f(αw + (1 - α)u) $\leq$ αf(w) + (1 - α)f(u) - $\frac{\lambda}{2}$ α(1-α) $\|w-u\|^2$
Then:
SGA for minimizing a λ-strongly convex function:
Goal: Solve $min_{w \in H}$ f(w).
parameter: T.
initialize: $w^{(1)} = 0$
for t = 1, ... , T.
Choose a random vector v_t such that E[v_t | $w^{(t)}$] $\in$ ∂f($w^{(t)}$).
Set η_t = 1/(λt).
Set $w^{(t+\frac{1}{2})}$ =$w^{(t)}$ - $\eta_t v_t$.
Set $w^{(t+1)}$ = arg$min_{w \in H}$ $||w - w^{(t+\frac{1}{2})}||^2$.
output = $\overline{w}$ = $\frac{1}{T}$ $\sum_{t=1}^{T}$ $w^{(t)}$.

Assume that f is λ-strongly convex and that E[$\|v_t\|^2$] $\leq$ ρ^2.
Let w*$\in$ $argmin_w \in H$ f(w) be an optimal solution. Then,
E[f($\bar{w}$)] - f(w*) $\leq$ (ρ^2) (1+log(T))/(2λT)

14 Conclusion of SGD

Throughout this part about SGD we've been able to explain a lot of things using severals papers like (Boyd, Boyd, & Vandenberghe, 2004),(Zhou, Han, & Guo, 2021),(Higham & Higham, 2019),(Goodfellow, Bengio, & Courville, 2016), (Bubeck, 2014), (McDonald, 2017), (Ruder, 2016), (D & DQ, n.d.) or even (Pérez-Enciso & Zingaretti, 2019). We've started by explaining deep neural networks and especially perceptron and sigmoid neurons , we've talked about the architecture of these neural networks and we wanted to trained them, that's why we introduce the cost function and gradient descent , but we've seen with our example in R that gradient descent was limited by our computational power . That's how we made the connection between deep neural networks and stochastic gradient algorithms. SGA are really powerful tools and we've seen that there were different way to compute those algorithm but usually we choosed to use mini-batch. The idea behind SGA is to reduce the convergence time. Depending on assumptions convergence time differs that's what we have for Convex-Lipschitz-Bounded Functions, strongly convex function, but also using a variable step size or other averaging techniques. Now that we have introduced one of the most popular optimization algorithm we will talk about one of the most promising.

15 Introduction: ADAM

ADAM is an adaptive learning rate optimization algorithm that has been designed to train deep neural networks. The first paper was published in (Kingma

& Ba, 2014) and ADAM was presented at ICLR 2015 which is a very prestigious conference for deep learning practitioners. The paper was very promising, it contains severals plots showing a huge performance gains for the speed of training in comparison to other algorithms such as SGDNesterov, RMSprop or even AdaGrad. Unfortunately, after a while people started to notice some problems with ADAM and especially researchers like Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro and Benjamin Recht showed that in some cases ADAM actually finds worse solution than stochastic gradient descent. A lot of research has been done to address the problems of ADAM.

In this paper, I will first introduce the ADAM algorithm that was presented in the original paper, and then walk through the latest research article to understand the result on convergence of ADAM under convexity and demonstrate some potential reasons why the algorithms can sometimes works worse than classical Stochastic Gradient Descent Algorithm. And i will especially talk about the issue of worse generalization to understand how we could improve the algorithm with numerical examples to narrow the gap between Stochastic Gradient Descent Algorithm and ADAM.

ADAM is an adaptive algorithms , adaptive algorithms leverages the power of adaptive learning rates methods to find individual learning rates for each parameter. ADAM also has advantages of AdaGrad and RMSprop.

AdaGrad is a modified stochastic gradient descent algorithm with per-parameter learning rate, published in ($Duchi, Hazan,$ $\&Singer$,2011). The learning rate increases for sparser parameters and decreases for ones that are less sparse. This strategy often improves convergence performance over standard stochastic gradient descent in settings where data is sparse and sparse parameters are more informative. It still has a base learning rate η, but this is multiplied with the elements of a vector $G_{j,j}$ which is the diagonal of the outer product matrix. The outer product of two coordinate vectors is a matrix. If the two vectors have dimensions n and m, then their outer product is an n × m matrix.

$$G = \sum_{\tau=1}^{t} g_\tau g_\tau^T \tag{26}$$

where $g_\tau = \nabla Q_i(\mathrm{w})$, the gradient, at iteration τ. The diagonal is given by:

$$G_{j,j} = \sum_{\tau=1}^{t} g_{\tau,j}^2 \tag{27}$$

This vector is updated after every iteration. The formula for an update written as per-parameter updates is now:

$$w_j \to w_j - \frac{\eta}{\sqrt{G_{j,j}}} g_j \tag{28}$$

The advantages of AdaGrad are that extreme parameter updates get dampened, while parameters that get few or small updates receive higher learning rates.

So ADAM has advantages of AdaGrad that works really well in settings with sparse gradients, but still struggles in non-convex optimization of neural networks, and RMSprop , which tackles to resolve some of the problems of AdaGrad works really well in on-line settings and we can see it below. ADAM has been raising in popularity exponentially according to (Karpathy, 2017).

RMSprop for Root Mean Square Propagation is a method in which the learning rate is adapted for each of the parameters. The idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight. The magnitude of the gradient is the maximum rate of change at the point. So, first the running average is calculated in terms of means square, more precisely :

$$v(w,t) = \gamma v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2 \tag{29}$$

where γ is the forgetting factor which gives exponentially less weight to older error samples. And the parameters are updated as follows :

$$w \to w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w) \tag{30}$$

Adam uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum. Let's now explain it more deeply.

We now explain Stochastic Gradient Descent with momentum so we can understand the features in common with ADAM . SGD with momentum remembers the update Δw at each iteration, and determines the next update as a linear combination of the gradient and the previous update such that :

$$\Delta w = \alpha \Delta w - \eta \nabla Q_i(w) \tag{31}$$

$$w \to w + \Delta w \tag{32}$$

And so:

$$w \to w - \eta \nabla Q_i(w) + \alpha \Delta w \tag{33}$$

where w which minimizes Q(w) is to be estimated , η the learning rate and α is an exponential decay between 0 and 1 that determines the relative contribution of the current gradient and earlier gradients to the weight change.

ADAM is an adaptive learning rate method, which means that it computes individual learning rates for different parameters. The name is coming from adaptive moment estimation, and the reason of the name is because ADAM uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network. The N-th moment of a random variable is defined as the expected value of that variable to the power of n. So in a mathematical expression: m_n= E$[X^n]$

The gradient of the cost function of neural network can be considered as a random variable because it is usually evaluated on some small random batch of data. The first moment is the mean, and the second moment is the uncentered variance (we don't subtract the mean during variance calculation).

To estimate the moments, ADAM use exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{34}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{35}$$

Where m and v are moving averages, g is gradient on current mini-batch, and betas are hyper-parameters of the algorithm. $\beta_1 = 0.9$ and $\beta_2 = 0.999$ can be considered as nice default values . The vectors of moving averages are initialized with zeros at the first iteration. Usually no one ever changes these values as they were used in ($Kingma$ &Ba,2014) after hyper-parameters tuning test.

To understand how these values are correlated with the moment we are looking at expected values of our moving averages. Since m and v are estimates of first and second moments, we want to have the following property:

$$E[m_t] = E[g_t] \tag{36}$$

$$E[v_t] = E[g_t^2] \tag{37}$$

Expected values of the estimators should be equal to the parameter we are trying to estimate, in our above equation this is the case so the parameter is also the expected value. If these properties held true that would mean that we have unbiased estimators. Unfortunately we will see that it is not the case. These does not hold true for our moving averages. Because we initialize averages with zeros, the estimators are biased towards zero.

Let's prove it for m , the proof for v would be analogous. To prove that we need to formula m to the very first gradient. We unroll a couple values of m to see the pattern we're going to use:

$$m_0 = 0 \tag{38}$$

$$m_1 = (1 - \beta_1) g_1 \tag{39}$$

$$m_2 = \beta_1 (1 - \beta_1) g_1 + (1 - \beta_1) g_2 \tag{40}$$

$$m_3 = \beta_1^2 (1 - \beta_1) g_1 + \beta_1 (1 - \beta_1) g_2 + (1 - \beta_1) g_3 \tag{41}$$

As we can see in these equation , the more we are expanding the value of m, the less first values of gradients contribute to the overall value, as they get multiplied by smaller and smaller beta. Understanding this we can rewrite the formula for our moving average as:

$$m_t = (1 - \beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i \tag{42}$$

Now, we will take a look at the expected value of m, to see how it's related to the true first moment so we can correct for the divergence of the two moments :

$$E[m_t] = E[g_i](1 - \beta_1^t) + \zeta \tag{43}$$

From this equation we found using the proof in (Bushaev, 2018) we can see two main things.

1.We have biased estimator. This is not just true for ADAM only, the same holds for every algorithms using moving averages like SGD with momentum, RMSprop, or the Heavy-Ball Method that will be introduced later for example.

2.But these biased estimator won't have much effect unless it's the beginning of the training, because the value beta to the power of t is quickly going towards zero.

Now we need to correct the estimator so that the expected value is the one we want. This is usually referred as the bias correction and the final formulas for our estimator will be as follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{44}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{45}$$

The only thing left to do is to use those moving averages to scale learning rate individually for each parameter. In ADAM it is very simple, to perform weight update we do the following:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{46}$$

Where w is model weights, η is the step size and it can depend on iteration , and ϵ is to prevent division from zero error. As β_1 and β_2 , η and ϵ have optimal default settings that are 0.001 and 10^{-8} introduced in (Kingma & Ba, 2014) And that's the update rule for ADAM.

We now have fully introduced ADAM but there are several variation of ADAM like Adamax that is introduced in (Kingma & Ba, 2014) or Nadam in (Dozat, 2016) that can be useful for major deep learning frameworks for example that we will now explain.

The idea behind Adamax is to look at the value v as the L2 norm of the individual current and past gradients.

In Linear Algebra, a Norm refers to the total length of all the vectors in a space. L2 norm also known as the Euclidean norm is the shortest distance to

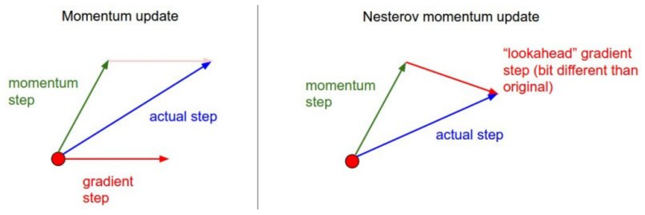


Figure 13: Visualisation of Nesterov momentum

go from one point to another while L-infinity norm gives the largest magnitude among each element of a vector. In L-infinity norm, only the largest element has any effect.

We then can generalize the L2 norm to a Lp norm update rule, but it gets pretty unstable for large values of p. But if we use the special case of L-infinity norm, it results in a surprisingly stable and well-performing algorithm. Where u_t is the exponentially weighted infinity norm and it's equal to :

$$u_t = max(\beta_2 \cdot u_{t-1}, |g_t|) \tag{47}$$

with initial value $u_0 = 0$ and note that, conveniently enough, we don't need to correct for initialization bias in this case. Also note that the magnitude of parameter updates has a simpler bound with AdaMax than Adam, namely: $|\Delta t| \leq \alpha$.

For Nadam as the name suggests, the idea is to use Nesterov momentum term for the first moving averages. With Nesterov accelerated momentum we make a big jump in the direction of the previous accumulated gradient and then we measure the gradient where we ended up to make a correction. There's a great visualization from cs231n lecture notes in Figure 13.

The advantage of this momentum is that it's a slightly modified version of Momentum with stronger theoretical convergence guarantees for convex functions. In the standard Momentum method, the gradient is computed using current parameters (θ_t). Nesterov momentum achieves stronger convergence by applying the velocity (v_t)which is the parameter we are trying to optimize to the parameters in order to compute interim parameters ($\theta = \theta_t + \mu * v_t$), where μ is the decay rate or friction, that tries to control the velocity and prevents overshooting the valley while allowing faster descent . These interim parameters are then used to compute the gradient, called a "lookahead" gradient step or a Nesterov Accelerated Gradient.

Let's take a look at update rule of the SGD momentum. Where m_t denotes the momentum and w_t denotes the weight

$$m_t = \beta m_{t-1} + \eta g_t \tag{48}$$

Where g_t denotes the gradient

$$w_t = w_{t-1} - m_t = w_{t-1} - \beta m_{t-1} - \eta g_t \tag{49}$$

As we can see above , the update rule is equivalent to taking a step in the direction of momentum vector and then taking a step in the direction of gradient. But, the momentum step does not depend on the current gradient as the update rule of equation (48) is an addition of the previous momentum * β and the current gradient * η and not a multiplication or division, as we can see in Figure 13 . So we can get a better quality gradient step direction by updating the parameters with the momentum step and then computing the gradient. Then the update rule is:

$$g_t = \nabla f(w_{t-1} - \beta m_{t-1}) \tag{50}$$

where f is the loss function to optimize.

$$m_t = \beta m_{t-1} + \eta g_t \tag{51}$$

$$w_t = w_{t-1} - m_t \tag{52}$$

So, using Nesterov accelerated momentum, we make a big jump in the direction of the previous accumulated gradient, then measure the gradient where we ended up to make a correction.

Let us make a few remarks about the ADAM properties based on (Kingma & Ba, 2014) and using their proofs:

1.Actual step size taken by ADAM in each iteration is approximately bounded by the step size hyper-parameter. This property add intuitive understanding to previous unintuitive learning rate hyper-parameter.

2.Step size of ADAM update rule is invariant to the magnitude of the gradient as it can depends on iteration, which helps a lot when going through areas with tiny gradients (such as saddle points or ravines). In these areas SGD struggles to quickly navigate through them.

3.ADAM was designed to combine the advantages of AdaGrad, which works well with sparse gradients, and RMSprop, which works well in on-line settings. Having both of these enables us to use ADAM for broader range of tasks. ADAM can also be looked at as the combination of RMSprop and SGD with momentum.

Part IV
The result on convergence of ADAM under convexity

In the original paper Diederik P. Kingma and Jimmy Lei Ba proved that ADAM converges to the global minimum under convexity , however, other authors later found out that their proof contained a few mistakes. In (Bock, Goppold, & Weiß, 2018) Sebastian Bock, Josef Goppold and Martin Weiß said that there were some mistakes in the convergence proof from Kingma and Ba but still find that the algorithm converges and provided proof in their paper. Another recent article was from Google employees and was presented at ICLR 2018 and even won best paper award. To go deeper to their paper we should first describe the framework used by ADAM authors to try to prove that ADAM converges for convex functions.

After this short introduction we will go in a chronological order with the different papers in the objective to understand if ADAM does converge for convex functions and if some authors did mistakes and how those possible errors impact the convergence under convexity.

16 Original Paper

In (Kingma & Ba, 2014) the authors analyzed the convergence of ADAM using the online learning framework proposed in (Zinkevich, 2003). In the presented settings, we have an unknown sequence of convex cost functions $f_1\theta$,$f_2\theta$, ..., $f_T\theta$.

At each time t, our goal is to predict the parameter θ_t and evaluate it on a previously unknown cost function f_t. Since the nature of the sequence is unknown in advance, we evaluate our algorithm using the regret, that is the sum of all the previous differences between the online prediction $f_t(\theta_t)$ and the best fixed point parameter $f_t(\theta^*)$ from a feasible set X for all the previous steps.

The algorithm, that solves the problem (ADAM) in each timestamp t chooses a point x[t] (parameters of the model) and then receives the loss function f for the current timestamp.

To understand how well the algorithm works, the value of regret of the algorithm after T rounds is defined as follows:

$$R(T) = \sum_{t=1}^{T} [f_t(\theta_t) - f_t(\theta^*)] \tag{53}$$

where R is regret, f_t is the loss function on the mini batch, θ is vector of model parameters (weights), and θ^* is the optimal value of weight vector. In the 2003 paper Martin Zinkevich proved that gradient descent converges to optimal solutions in this setting, using the property of the convex functions:

$$f_t(x) \geq (\nabla f_t(x_t))(x - x_t) + f_t(x_t) \tag{54}$$

In (Kingma & Ba, 2014) the authors showed that ADAM has O($\sqrt{T}$) regret bound , a proof is given in the appendix of the paper, so on average the model converges to an optimal solution. Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

From Theorem 4.1 in (Kingma & Ba, 2014) and from corollary 4.2 we know that ADAM achieves the following guarantee, for all T $\geq$ 1.

$$\frac{R(T)}{T} = O(\frac{1}{\sqrt{T}}) \tag{55}$$

Thus :

$$lim_{T\to\infty}\frac{R(T)}{T} = 0 \tag{56}$$

17 An improvement of the convergence proof of the ADAM-Optimizer

Other researchers tried to use the same method to prove that their algorithm converges to an optimal solution but several mistakes were spotted . One of the first paper to talk about them was (Bock et al., 2018) coming from the fact that the Lemma 10.4 in the original paper could not be proved and then it will be considered as a conjecture , despite it the authors proved that:

$$lim_{T\to\infty}\frac{R(T)}{T} \leq lim_{T\to\infty}(\frac{1}{\sqrt{T}}) + (\frac{1}{\sqrt{T}}) + (\frac{1}{T}) = 0 \tag{57}$$

And so this proves the convergence speed O($\frac{1}{\sqrt{T}}$) of the ADAM method.

18 On the Convergence of ADAM and Beyond

But in (Reddi, Kale, & Kumar, 2019) in their paper presented at ICLR 2018, the authors have found several mistakes in both proofs, the main one is lying from the following quantity of interest appearing in both (Kingma & Ba, 2014)and (Bock et al., 2018) proofs:

$$\Gamma_{t+1} = (\frac{\sqrt{V_{t+1}}}{\eta_{t+1}} - \frac{\sqrt{V_t}}{\eta_t}) \tag{58}$$

Where V_t is as an abstract function that scales learning rate for parameters which differs for each individual algorithms. For ADAM it's the moving averages of past squared gradients, for AdaGrad it's the sum of all past and current gradients, for SGD it's the fixed value 1. This quantity essentially measures the change in the inverse of learning rate of the adaptive method with respect to time.

The authors found that the proof is only working if this value is positive. SGD or AdaGrad are always positive as we can see from the formulas but for ADAM and RMSprop the value of V can be different and can act unexpectedly.

Below is an example of sequence linear functions where ADAM fails ton converge :

$$f_t(x) = \begin{cases} \text{C}x \text{ for t mod } 3 = 1, \\ -x \text{ otherwise} \end{cases} \tag{59}$$

Where C > 2. For this sequence , we can see and deduct that the optimal solution is x = -1, it provides the minimum regret however, the authors showed in the paper that if $\beta_1 = 0$ and $\beta_2 = \frac{1}{(1+C^2)}$, ADAM converges to highly sub-optimal value of x = 1. The algorithm obtains the large gradient C once every 3 steps, while the other 2 steps observes the gradient -1 , which moves the algorithm in the wrong direction. The large gradient C is unable to counteract this effect since it is scaled down by a factor of almost C for the given value of β_2, and hence the algorithm converges to 1 rather than -1. Since values of step size are often decreasing over time, they proposed a fix of keeping the maximum of values V and use it instead of the moving average to update parameters. The resulting algorithm is called Amsgrad.

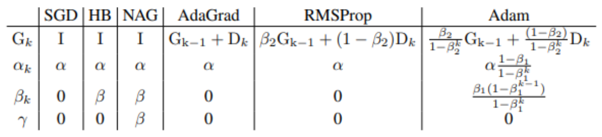
Part V

The issue of worse generalization of ADAM than SGD in some cases

Whenever we train our own neural networks, we need to take care of something called the generalization of the neural network. This essentially means how good our model is at learning from the given data and applying the learnt information elsewhere.

When training a neural network, there's going to be some data that the neural network trains on, and there's going to be some data reserved for checking the performance of the neural network. If the neural network performs well on the data which it has not trained on, we can say it has generalized well on the given data. In machine learning they are called the training, validation, and test data sets. The training dataset is the sample of data used to fit the model. The validation dataset is the sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyper-parameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration. The test dataset is the sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.

To evaluate the performance of those algorithms, we are using several machine learning models and datasets. For example we can use logistic regression, multi-layer fully connected neural networks and deep convolutional neural networks to try to demonstrate that ADAM can efficiently solve practical deep learning problems. When we compare the different algorithms we always use the same parameter initialization to be relevant, while hyper-parameters like



	SGD	HB	NAG	AdaGrad	RMSProp	Adam
G_k	I	I	I	$G_{k-1} + D_k$	$\beta_2 G_{k-1} + (1-\beta_2) D_k$	$\frac{\beta_2}{1-\beta_2^k} G_{k-1} + \frac{(1-\beta_2)}{1-\beta_2^k} D_k$
α_k	α	α	α	α	α	$\alpha \frac{1-\beta_1}{1-\beta_1^k}$
β_k	0	β	β	0	0	$\frac{\beta_1 (1-\beta_1^{k-1})}{1-\beta_1^k}$
γ	0	0	β	0	0	0

Figure 14: Parameter settings of algorithms used in deep learning. Here, D_k = diag($g_k \circ g_k$) and $G_k \rightarrow H_k \circ H_k$. We omit the additional ϵ added to the adaptive methods, which is only needed to ensure non-singularity of the matrices H_k

learning rate or momentum are look over a dense grid by authors to find the optimal hyper-parameter setting and then results are reported using them.

We can find some of those parameters values in the Figure 14 in (Wilson, Roelofs, Stern, Srebro, & Recht, 2017), where computation are explained.

Where $\circ$ denotes the entry-wise or Hadamard product.

In this context, generalization refers to the performance of a solution w on a broader population. Performance is often defined in terms of a different loss function than the function f used in training. For example, in classification tasks, we typically define generalization in terms of classification error rather than cross-entropy .

When ADAM was introduced in (Kingma & Ba, 2014), people at ICLR 2015 were very excited about the power of ADAM and for good reason, the paper contained some very optimistic charts, showing huge performance gains in terms of speed of training in comparison to other algorithms like SGDNesterov, RMSprop or AdaGrad.

19 Logistic regression

In (Kingma & Ba, 2014), the authors evaluated their proposed method on L2-regularized multi-class logistic regression using the MNIST dataset, this is a large database of handwritten digits that is commonly used for training various image processing systems. One of the advantage of logistic regression is that it has a well-studied convex objective, making it appropriate for comparison of different optimizers without worrying about local minimum issues. The stepsize α is adjusted by $\frac{1}{\sqrt{t}}$ decay in the logistic regression experiments, namely $\alpha_t = \frac{\alpha}{\sqrt{t}}$.

The authors compare ADAM to accelerated SGD with Nesterov momentum and AdaGrad using minibatch size of 128. The logistic regression classifies the class label directly on the 784 dimension image vectors.

From the Figure 15 from(Kingma & Ba, 2014) we can see and found that the ADAM algorithm have a pretty similar convergence as SGD with Nesterov momentum and they are both way faster than AdaGrad to converge. This is not really surprising because in (Duchi et al., 2011) we have learnt that AdaGrad is

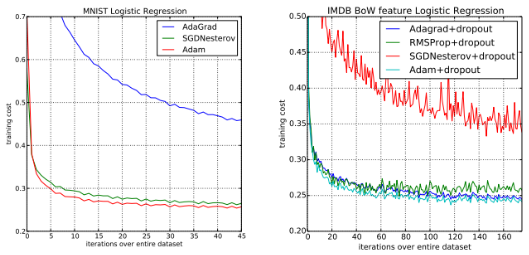


Figure 15: Logistic regression training negative log likelihood on MNIST images and IMDB movie reviews with 10,000 bag-of-words (BoW) feature vectors.

efficient with sparse features and gradients as one of its main theoretical results while SGD is low at learning sparse features. On the other side ADAM should have similar performance as AdaGrad because of the decay rate ($\frac{1}{\sqrt{t}}$)

Then the authors of (Kingma & Ba, 2014) used the IMDB movie review dataset from (Maas et al., 2011) to look at sparse feature problem. They preprocess the IMDB movie reviews into bag-of-words (BoW) feature vectors including the first 10,000 most frequent words. The 10,000 dimension BoW feature vector for each review is highly sparse. And a 50% dropout noise was applied to the BoW features during training to avoid overfitting proposed in (Wang & Manning, 2013). We can see from this second figure that with dropout ADAM, RMSprop and AdaGrad have a similar convergence time, while all of them outperforms SGD with Nesterov momentum by a large margin. ADAM have a similar convergence to AdaGrad and then can take advantage of sparse features and obtain faster convergence rate in comparison to SGD with Nesterov momentum.

Dropout and other stochastic regularization approaches are good at preventing overfitting and are frequently employed in practise because of their simplicity. Dropout stochastic regularization is a regularization technique to reduce overfitting in artificial neural networks by preventing complex co-adaptations on training data. An overfitted model is a statistical model that contains more parameters than can be justified by the data. It is an efficient way of performing model averaging with neural networks. The term dropout refers to randomly "dropping out", or omitting, units (both hidden and visible) during the training process of a neural network.

20 Multi-layer neural networks

To test ADAM for non-convex objective functions, multi-layer neural networks are powerful models. Despite the fact that our convergence analysis does not apply to non-convex problems, we have empirically found that ADAM often outperforms other methods in these situations.

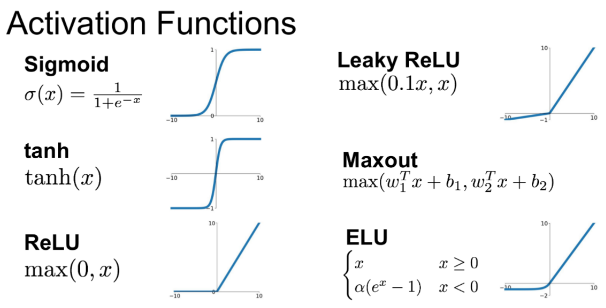


Figure 16: Figure of the most popular activation functions in deep learning

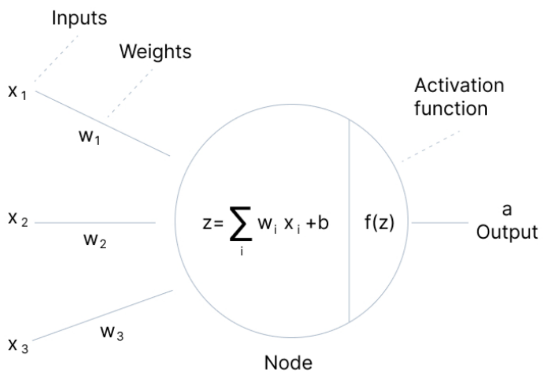


Figure 17: Example of a neuron with activation function in neural network

In the experiment, we used a neural network model with two fully connected hidden layers with 1000 hidden units each and ReLU activation with minibatch size of 128.

An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations, we have an example of the most popular activation functions in deep learning in Figure 16 from (Jayawardana & Bandaranayake, 2021).

The role of the Activation Function is to derive output from a set of input values fed to a node (or a layer). The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output. Note: All hidden layers usually use the same activation function. However, the output layer will typically use a different activation function from the hidden layers. The choice depends on the goal or type of prediction made by the model.

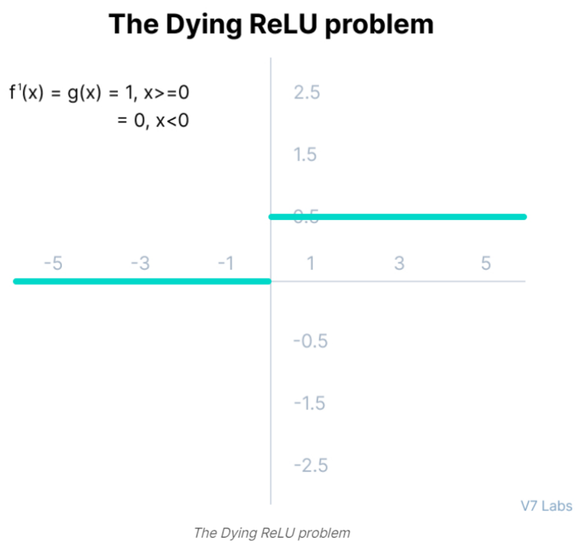


Figure 18: Dying ReLU problem

In this experiment we use ReLU activation function that stands for Rectified Linear Unit.

Although it gives an impression of a linear function in Figure 16 , ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.

The main catch here is that the ReLU function does not activate all the neurons at the same time.

The neurons will only be deactivated if the output of the linear transformation is less than 0.

The advantages of using ReLU as an activation function are as follows:

since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions. ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property. The limitation faced by this function is the dying ReLU problem, that can be see in Figure 18 .

The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.

All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

So, in the experiment in multi-layer neural networks , to avoid over-fitting, we first investigate several optimizers using the typical deterministic cross-

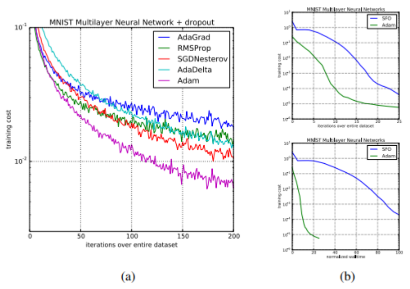


Figure 19: Training of multilayer neural networks on MNIST images. (a) Neural networks using dropout stochastic regularization. (b) Neural networks with deterministic cost function. We compare with the sum-of-functions (SFO) optimizer (Sohl-Dickstein et al., 2014)

entropy objective function with L2 weight decay on the parameters.

The sum-of-functions (SFO) method in (Sohl-Dickstein, Poole, & Ganguli, 2014) is a recently proposed quasi-Newton method(method used to either find zeroes or local maxima and minima of functions) that works with minibatches of data and has demonstrated good results in multi-layer neural network optimization. We used their implementation and compared with ADAM to train such models. SFO assumes deterministic subfunctions, and stochastic regularization fails to converge on cost functions.

Figure 19 show that ADAM progresses faster in the number of iterations in comparison to SGDNesterov, AdaDelta,RMSprop,AdaGrad but also in the time spent on the wall clock in comparison to SFO that is 5-10x slower each iteration than ADAM due to the cost of updating curvature information, and has a memory demand that is linear in the number of minibatches.

On multi-layer neural networks trained with dropout noise, we compare ADAM's effectiveness to other stochastic first order methods. The results are shown in Figure 19 from (Kingma & Ba, 2014) and ADAM has a higher convergence rate than the other approaches.

Then in (Kingma & Ba, 2014), after multi-layer neural networks, the authors experimented in convolutional neural networks.

In computer vision tasks, convolutional neural networks (CNNs) with several layers of convolution, pooling, and non-linear units have shown a lot of success. Weight sharing in CNNs, unlike other fully connected neural networks, leads in vastly differing gradients in various layers. Weight sharing is an old-school technique for reducing the number of weights in a network that must be trained. It is exactly what it sounds like: the reuse of weights on nodes that are close to one another in some way. When employing SGD, a smaller learning rate for the convolution layers is frequently employed in practise. In deep CNNs, we demonstrate ADAM's usefulness. Following a fully linked layer of 1000 rectified linear hidden units (ReLUs), our CNN design contains three alternating stages

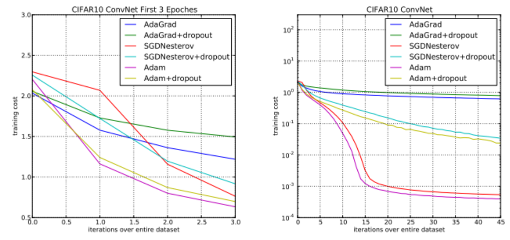


Figure 20: Convolutional neural networks training cost. (left) Training cost for the first three epochs. (right) Training cost over 45 epochs. CIFAR-10 with c64-c64-c128-1000 architecture.

of 5x5 convolution filters and 3x3 max pooling with stride of 2.

The input image are pre-processed by whitening, and dropout noise is applied to the input layer and fully connected layer. Similarly to prior studies, the minibatch size is set to 128.

Interestingly, despite the fact that both ADAM and AdaGrad make rapid progress in lowering the cost in the early stages of the training in comparison to SGDNesterov, as shown in Figure 20 (left) from (Kingma & Ba, 2014), ADAM and SGDNesterov eventually converge more faster than AdaGrad for CNNs (right). Even though ADAM shows marginal improvement over SGD with momentum, it adapts learning rate scale for different layers instead of hand picking manually as in SGD.

In every figure of the original paper we can see that ADAM is always one the most performing algorithm .

Then, in (Dozat, 2016) paper presented diagrams that showed even better results than (Kingma & Ba, 2014).

To evaluate the performance of this Nesterov-accelerated ADAM (Nadam), we used a convolutional auto encoder adapted from(Jones, 2015) with three convolutional layers and two dense layers in each encoder and decoder to compress images from the MNIST dataset (LeCun, 1998) into a 16-dimensional vector space, then reconstruct the original image which is known to be a difficult task.

Six optimizations algorithms were tested RMSprop, SGD, Momentum, Nesterov accelerated gradient(NAG), Nadam , ADAM, and all of them used initialization bias correction and decaying means (rather than decaying sums) where relevant.

The best learning rate found for SGD was 0.2, for Momentum/NAG was 0.5, for RMSProp was 0.001, and for ADAM/Nadam was 0.002. μ was set to 0.975, ν was set to 0.999 and ϵ to $1e^{-8}$.

Figure 21 from (Dozat, 2016) shows that, despite having the most hyperparameters, Nadam and ADAM produce the greatest outcomes even with no modification beyond the learning rate. Nadam, in particular, outperform the other algorithms in terms of minimising training and validation loss, including

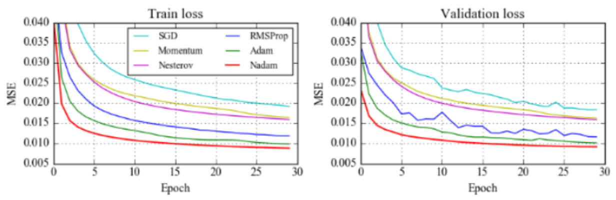


Figure 21: Training and validation loss of different optimizers on the MNIST dataset

ADAM.

However, after some time, people began to notice that, despite its superior training time, ADAM does not always converge to an optimal solution, and that for some tasks (such as image classification on popular CIFAR datasets), state-of-the-art results can still be obtained only by using SGD with momentum.

We now move to an actual examination of deep neural networks to see if we detect a similar discrepancy in generalization after establishing that adaptive and non-adaptive approaches can discover different solutions in the convex environment using (Wilson et al., 2017). We compare three common adaptive approaches – AdaGrad, RMSProp, and ADAM – against two non-adaptive methods – SGD and the Heavy-ball method (HB). We look at the performances of the algorithms on four deep learning problems: (C1) the image classification task CIFAR-10 (L1) character-level language modeling on the novel War and Peace, and (L2) discriminative parsing and (L3) generative parsing on Penn Treebank.

First of all we will introduce the Polyak's Heavy-Ball method: The Heavy-vall method was inspired by the gradient descent issues in (Polyak, 1964). In the gradient descent update formula :

$$x_{t+1} = x_t - \alpha \nabla f(x_t) \tag{60}$$

where α is the learning rate and ∇ f(x_t) is the gradient. But there is some issues with this gradient descent formula, we see that a large learning rate leads to bigger steps than we would like in high curvature regions, while a small learning rate leads to smaller steps than we would like in low curvature regions. So Polyak wanted two things to fix the formula:

1.To make smaller steps in regions of high curvature to dampen oscillations.

2.To make larger steps and accelerate in regions of low curvature.

One way to do both is to guide the next steps towards the previous direction, that's why he introduced momentum and the new update rule that can do both:

$$x_{t+1} = x_t - \alpha \nabla f(x_t) + \beta(x_t - x_{t+1}) \tag{61}$$

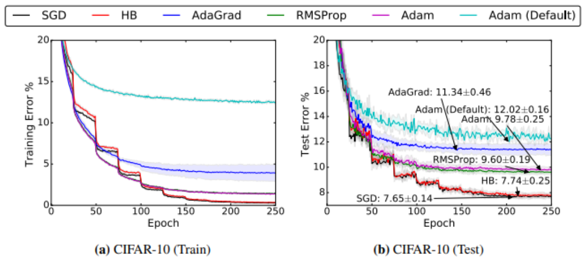


Figure 22: Training (left) and top-1 test error (right) on CIFAR-10. The annotations indicate where the best performance is attained for each method. The shading represents ± one standard deviation computed across five runs from random initial starting points. In all cases, adaptive methods are performing worse on both train and test than non-adaptive methods.

Then, the momentum has 2 effects:

1. We penalize changes in direction to take smaller steps

2. When the direction doesn't change we take bigger steps

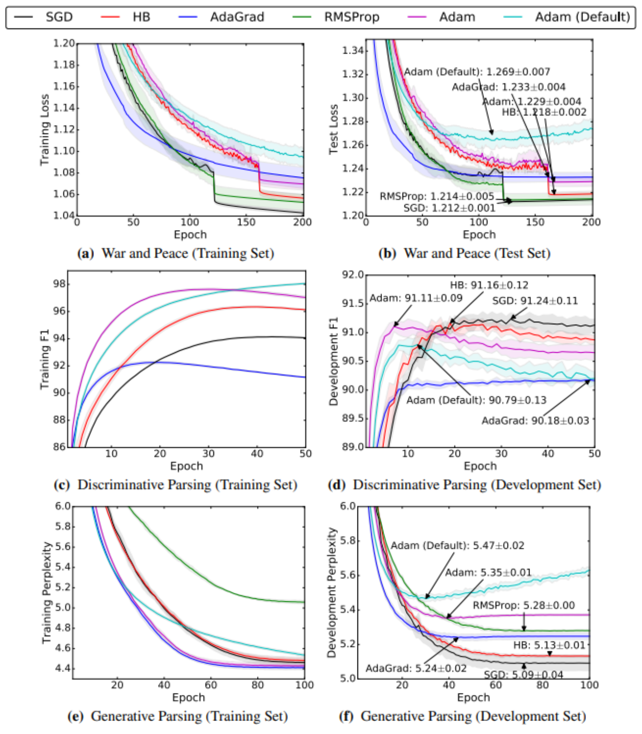
And that is what we wanted to fix in the Gradient Descent algorithm.

Using the initialization scheme described in each code repository, we repeat each experiment five times from randomly initialised starting points. Each model is given a budget based on the number of epochs it will be trained with. We chose the settings that achieved the best peak performance on the development set by the end of the fixed period epoch where a development set was available. We chose the settings that resulted in the lowest training loss at the end of the fixed epoch budget because CIFAR-10 lacked an explicit development set.

21 Convolutional Neural Network

For CIFAR-10, the authors used the VGG+BN+Dropout network from (Zagoruyko, 2015), which has a baseline test error of 7.55 percent in previous work. On both the training and test datasets, Figure 22 depicts the learning curve for each algorithm.

We see in Figure 22 from (Wilson et al., 2017) that the solutions found by SGD and HB generalize better than those discovered by adaptive approaches. The best non-adaptive algorithm, SGD, had test error of 7.65 ± 0.14 %, while the best adaptive approach, RMSprop, had a test error of 9.60 ± 0.19%, it is even worst for ADAM and Adam(default) that had a test error of 9.78 ± 0.25% and 12.02 ± 0.16 %.



(a) War and Peace (Training Set)

(b) War and Peace (Test Set)

(c) Discriminative Parsing (Training Set)

(d) Discriminative Parsing (Development Set)

(e) Generative Parsing (Training Set)

(f) Generative Parsing (Development Set)

Figure 23: Performance curves on the training data (left) and the development/test data (right) for three experiments on natural language tasks. The annotations indicate where the best performance is attained for each method. The shading represents one standard deviation computed across five runs from random initial starting points.

Adaptive approaches appear to perform better than non-adaptive methods early in training, but starting at epoch 50, even though the adaptive methods' training error is still lower, SGD and HB begin to outperform adaptive methods on the test error. On both the train and test, SGD and HB outperform all adaptive algorithms by epoch 100. AdaGrad's rate of improvement flattens out the earliest of all adaptive algorithms. ADAM and ADAM(default) are way behind SGD, HB, and RMSprop , the only algorithm that ADAM outperform is AdaGrad. We also discovered that increasing the step size reduced the adaptive methods' performance in the first 50 or so epochs, but the aggressive step size exacerbated the flatlining behaviour, which no step decay scheme could fix.

22 Character-Level Language Modeling

We train a character-level language model on the text of War and Peace using the torch-rnn library, with a fixed budget of 200 epochs. The results are shown in Figure 23 (a) and (b) from (Wilson et al., 2017). The optimal configuration for all algorithms except AdaGrad under the fixed-decay scheme was to decay relatively late in terms of the total number of epochs, either 60 or 80 percent through the entire number of epochs and by a large amount, dividing the step size by 10.

The results of the exhaustive search over the decay frequency and amount were paralleled (within the same standard deviation) by the dev-decay scheme; we show the curves from the fixed policy. With a test loss of 1.212 ± 0.001, SGD had the lowest overall result. AdaGrad makes rapid progress at first, but then plateaus. Adaptive methods appear to be more sensitive to initialization schemes than non-adaptive methods, with larger variation on both the train and test runs. Surprisingly, RMSprop comes in second place behind SGD on test loss , demonstrating that adaptive approaches can identify solutions that generalize well, while ADAM with 1.229 ± 0.004 is still behind H-B and Adam(default) with 1.269 ± 0.007 is last .

23 Constituency Parsing

The hierarchical structure of a sentence is predicted using a constituency parser, which breaks it down into nested clause-level, phrase-level, and word-level units.

We test two state-of-the-art parsers: Cross and Huang's stand-alone discriminative parser and the generative parser, Choe and Charniak's reranking parser . We utilise the dev-decay strategy with $\delta = 0.9$ in both cases for learning rate decay.

23.1 Discriminative model

Cross and Huang propose a transition-based framework that reduces constituency parsing to a sequence prediction problem, giving a one-to-one correspondence between parse trees and sequences of structural and labeling actions. We trained for 50 epochs on the Penn Treebank, comparing labelled F1 scores on training and development data across time, using their code with default settings. We excluded RMSprop from our tests because it was not implemented in the DyNet version we used. Figures 23 (c) and (d) show the results . On the development set, we discovered that SGD had the best overall performance, followed by HB and ADAM, with AdaGrad trailing far behind. By the end of the run, Adam's default configuration without learning rate decay had the best overall training performance, but it was noticeably worse than tuned Adam on the development set. Adam reached its highest development F1 of 91.11 after only 6 epochs, whereas SGD took 18 epochs to attain this number and didn't reach its highest F1 of 91.24 until epoch 31. Adam, on the other hand, continued to improve on the training set well after its best development performance was obtained, while the peaks for SGD were more closely aligned.

23.2 Generative model

Choe and Charniak demonstrate that constituency parsing may be modelled as a language modelling problem, with trees represented by depth-first traversals. This formulation necessitates the use of a separate base system to produce candidate parse trees, which the generative model then rescored. We retrained their model on the Penn Treebank for 100 epochs using an updated version of their code base. However, we made two minor changes to reduce computational costs: (a) we used a smaller LSTM hidden dimension of 500 instead of 1500, finding that performance decreased only slightly; and (b) we accordingly lowered the dropout ratio from 0.7 to 0.5.

We looked into the relationship between training and development perplexity to avoid any confusion with the performance of a base parser because they found a high conflation between perplexity (the exponential of the average loss) and labelled F1 on the development set.

Figures 23(e) and 11(f) show the results . SGD and HB had the best perplexities on the development set, with SGD slightly ahead. Adam has the worst development perplexities despite having one of the best performance curves on the training dataset.

Then, the following are the primary findings of the experiments:

1. Adaptive methods find solutions that generalize worse than those found by non-adaptive methods.
2. Even when the adaptive methods achieve the same training loss or lower than non-adaptive methods, the development or test performance is worse.
3. Adaptive methods often display faster initial progress on the training set, but their performance quickly plateaus on the development set.
4.Though conventional wisdom suggests that ADAM does not require tuning, we find that tuning the initial learning rate and decay scheme for ADAM yields significant improvements over its default settings in all cases.

Since then, a lot of study has been done to try to understand ADAM's weak generalisation and bridge the gap with SGD and we will present some of them below.

One of the solution might be in (Keskar & Socher, 2017) for example, where Nitish Shirish Keskar and Richard Socher demonstrated that switching to SGD during training improved generalization power over using ADAM alone. They've discovered that ADAM outperforms SGD in the early phases of training, but that his learning plateaus later. They introduced a simple technique known as SWATS, in which they train deep neural networks with ADAM and then switch to SGD when certain criteria are met. With momentum, they were able to attain outcomes comparable to SGD.

24 Weight decay with ADAM

One paper that actually turned out to help ADAM is (Loshchilov & Hutter, 2018). There are numerous contributions and insights regarding Adam and weight decay in this paper. They first show that, contrary to popular belief, Regularization is not the same as weight decay, though they are equivalent for stochastic gradient descent. Weight decay was first introduced in 1988 in the following manner:

$$w_{t+1} = (1 - \lambda)w_t - \eta \nabla f_t w_t \tag{62}$$

Where λ is the weight decay hyper-parameter to tune. As defined above, weight decay is applied in the last step, when making the weight update, penalizing large weights. SGD has traditionally been implemented using L2 regularization, which involves modifying the cost function to contain the weight vector's L2 norm:

$$f_t^{reg}(w_t) = f_t(w_t) + \frac{\lambda}{2}||w_t||_2^2 \tag{63}$$

Traditionally, stochastic gradient descent algorithms, as well as ADAM, inherited this type of weight decay regularization implementation. For ADAM, however, regularization is not the same as weight decay. When utilising L2 regularization, the penalty we use for large weights is scaled by a moving average of the past and present squared gradients, resulting in weights with large typical gradient magnitudes being regularized by a smaller relative amount than other weights. Weight decay, on the other hand, regularizes all weights by the same factor. To use weight decay with ADAM, we need to make the following changes to the update rule:

$$w_t = w_{t-1} - \eta(\frac{\hat{m}}{\sqrt{\hat{v}_t + \epsilon}} + \lambda w_{t-1}) \tag{64}$$

After showing that these types of regularization differ for ADAM, authors continue to show how well it works with ADAM and SGD. The difference in results is shown with the diagram in Figure 24 from (Loshchilov & Hutter, 2018).

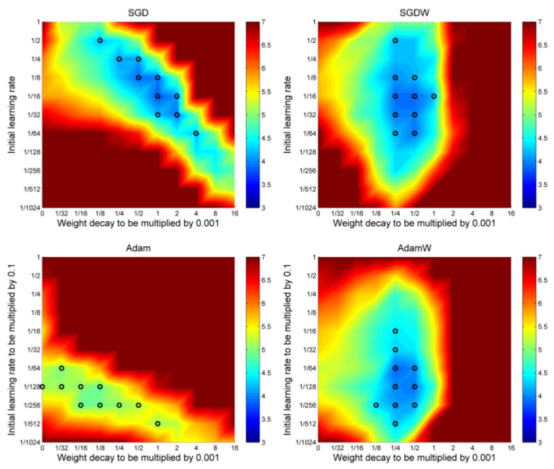


Figure 24: The Top-1 test error of a 26 2x64d ResNet on CIFAR-10 measured after 100 epochs. The proposed SGDW and AdamW (right column) have a more separable hyper-parameter space.

The relationship between learning rate and regularization method is depicted in these diagrams. The colour represents the test error for this pair of hyper-parameters, which is high or low. As we can see above, weight decay not only lowers test error, but it also helps in the decoupling of learning rate and regularization hyper-parameter.

Weight decay is not decoupled from learning rate in the case of SGD (Figure 24, top left), and the Figure 24 clearly shows that the basin of best hyper-parameter settings (depicted by colour and top-10 hyper-parameter settings by black circles) is not aligned with the x-axis or y-axis but rather lies on the diagonal. This shows that the two hyper-parameters are interdependent and must be changed at the same time, as changing just one of them could significantly deteriorate results.

Figure 24 (top right) shows the findings for our new SGDW, which show that SGDW decouples weight decay and initial learning rate. The suggested method separates the two hyper-parameters: even if the learning rate is not yet properly tuned (for example, see Figure 24, top right), leaving it fixed and only tuning the weight decay factor will provide a good value . This is not the case with the original SGD (Figure 24 , top left).

On the left bottom, we can see that if one of the parameters, such as learning rate, is changed, we'll need to modify the L2 factor as well, showing that these two parameters are interdependent. Because of this dependency, hyper-parameter tuning can be a tough undertaking at times. We can see on the right bottom one that as long as we keep it within a certain range of optimal values for one parameter, we can change another one independently.

In summary, the experimental results in Figure 24 shows that the weight decay and learning rate hyper-parameters can be decoupled, which simplifies the hyper-parameter tuning challenge in SGD and improves Adam's performance in SGD with momentum.

Another contribution by the authors of the paper shows that optimal value to use for weight decay actually depends on number of iteration during training. They developed a simple adaptive formula for setting weight decay to deal with this fact:

$$\lambda = \lambda_{norm}\sqrt{\frac{b}{BT}} \tag{65}$$

where b is batch size, B is the total number of training points per epoch and T is the total number of epochs. This replaces the lambda hyper-parameter lambda by the new one, lambda normalized.

They tried to apply the learning rate schedule with warm restarts with the updated version of ADAM after fixing weight decay. Warm restarts greatly helped stochastic gradient descent. However, ADAM was previously far behind SGD. With the improved weight decay, ADAM's restart results are significantly better, but it's still not as good as SGDR's.(SGD with warm restart)

25 ND-ADAM

In (Zhang, Ma, Li, & Wu, 2017) the authors proposed another method to fix ADAM. The paper notices two problems with ADAM that may cause worse generalization:

1. SGD updates lies within the span of historical gradient , whereas ADAM update are not.

2. While the magnitudes of ADAM parameter updates are invariant by gradient descaling, the effect of the updates on the same overall network function varies with parameter magnitudes.

To solve these issues, the authors suggest the "Normalized direction-preserving ADAM method". Adam is tweaked by the algorithms in the following ways. Instead of estimating the average gradient magnitude for each individual parameter, it estimates the average squared L2 norm of the gradient vector. Because V is now a scalar value and M is a vector pointing in the same direction as W, the update's direction is the negative direction of m and thus is in the span of the historical gradients of w. And, the algorithms project it onto the unit sphere before using gradient, and then the weights get normalised by their norm after the update.

Part VI
Implement ADAM and SGD on some numerical example

Using R, we have implemented several optimization alogirthms, SGD, Momentum, AdaGrad , ADAM , Nadam and RMSprop. The code with comments is included below.

```
## SGD
SGD = setRefClass("SGD",
fields = list(eta="numeric", t="numeric"),
methods = list(
step = function(gradient) {
inc  =  - eta * gradient / t
t <<- t + 1
return(inc)
},
initialize = function(eta, t=1) {
.self$eta = eta
.self$t = t
}
)
)

#Example SGD
set.seed(724) # Reproduce results
n = 1000 # Number of iterations
pars = rnorm(1000) # Generate random number using normal distribution (Initial parameter va
output = matrix(nrow=n, ncol=1000)
output[1,] = pars #matrix giving the distribution parameter values for each instance in the
sgd = SGD$new(eta=0.5)
for (i in 2:n) {
# Gradient for quadratic cost
grad = pars + rnorm(1000)
pars = pars + sgd$step(grad)
output[i,] = pars
}
plot(output, type='l')




##Momentum
momentum = setRefClass("momentum", #setRefClass() returns a generator function suitable for
fields = list(eta="numeric", alpha="numeric", w="numeric", t="numeric"), #named list of the
methods = list( #a named list of function definitions that can be invoked on objects from th
step = function(gradient) { #The "step" method take an estimated gradient as argument ,  fun
w <<- w * alpha - eta * gradient / t #update rule
inc = w # increment it
t <<- t + 1
return(inc) # returns an increment to apply to the current parameter values
},
initialize = function(eta, alpha=0.5, w=0, t=1) { # best hyper-parameters inital values foun
.self$eta = eta #we set-up every inital values
.self$alpha = alpha
.self$w = w
.self$t = t
}
)
)

#Example Momentum
set.seed(724) # Reproduce results
n = 1000 # Number of iterations
pars = rnorm(1000) # Generate random number using normal distribution (Initial parameter val
output = matrix(nrow=n, ncol=1000)
output[1,] = pars #matrix giving the distribution parameter values for each instance in the
mom = momentum$new(eta=0.5)
for (i in 2:n) {
# Gradient for quadratic cost with minimum at origin
grad = pars + rnorm(1000)
pars = pars + mom$step(grad)
output[i,] = pars
}
plot(output, type='l')



##AdaGrad
adaGrad = setRefClass("adaGrad",
fields = list(eta="numeric", epsilon="numeric", alpha="numeric", grad_squared="numeric", w="
methods = list(
step = function(gradient) {
grad_squared <<- grad_squared + c(gradient)^2
w <<- w * alpha - eta * c(gradient) / (epsilon + sqrt(grad_squared))
return(w)
},
initialize = function(eta, alpha=0, epsilon=1E-7) {
.self$eta = eta
.self$epsilon = epsilon
```

```
.self$alpha = alpha
.self$grad_squared = 0 # grad_squared and w start off as scalars but become vectors of the r
.self$w = 0
}
)
)

#Example AdaGrad
set.seed(724) # Reproduce results
n = 1000 # Number of iterations
pars = rnorm(1000) # Generate random number using normal distribution (Initial parameter val
output = matrix(nrow=n, ncol=1000)
output[1,] = pars #matrix giving the distribution parameter values for each instance in the
adaG= adaGrad$new(eta=0.5)
for (i in 2:n) {
# Gradient for quadratic cost
grad = pars + rnorm(1000)
pars = pars + adaG$step(grad)
output[i,] = pars
}
plot(output, type='l')



##ADAM
ADAM = setRefClass("ADAM",
fields = list(eta="numeric", beta1="numeric", beta2="numeric", epsilon="numeric", m="numeric
methods = list(
step = function(gradient) {
t <<- t + 1
m <<- beta1*m + (1-beta1)*c(gradient)
v <<- beta2*v + (1-beta2)*(c(gradient)^2)
m_hat = m / (1-beta1^t)
v_hat = v / (1-beta2^t)
inc = - eta * m_hat / (sqrt(v_hat) + epsilon)
return(inc)
},
initialize = function(eta=0.001, beta1=0.9, beta2=0.999, epsilon=1E-8) {
.self$eta = eta
.self$beta1 = beta1
.self$beta2 = beta2
.self$epsilon = epsilon
.self$m = 0 # m and v start off as scalars but become vectors of the right length on their f
.self$v = 0
.self$t = 0
}
)
)


#Example ADAM
set.seed(724) # Reproduce results
n = 1000 # Number of iterations
pars = rnorm(1000) # Generate random number using normal distribution (Initial parameter val
output = matrix(nrow=n, ncol=1000)
output[1,] = pars #matrix giving the distribution parameter values for each instance in the
adam = ADAM$new(eta=0.5)
for (i in 2:n) {
# Gradient for quadratic cost
grad = pars + rnorm(1000)
pars = pars + adam$step(grad)
output[i,] = pars
}
plot(output, type='l')

##Nadam
Nadam = setRefClass("Nadam",
fields = list(eta="numeric", beta1="numeric", beta2="numeric", epsilon="numeric", m="numeric
methods = list(
step = function(gradient) {
t <<- t + 1
m <<- beta1*m + (1-beta1)*c(gradient)
v <<- beta2*v + (1-beta2)*(c(gradient)^2)
m_hat = m / (1-beta1^t) + (1 - beta1) * gradient / (1-beta1^t)
v_hat = v / (1-beta2^t)
inc =  - eta * m_hat / (sqrt(v_hat) + epsilon)
return(inc)
},
initialize = function(eta=0.001, beta1=0.9, beta2=0.999, epsilon=1E-8) {
.self$eta = eta
.self$beta1 = beta1
.self$beta2 = beta2
.self$epsilon = epsilon
.self$m = 0 # m and v start off as scalars but become vectors of the right length on their f
.self$v = 0
.self$t = 0
}
)
)
```

```
#Example Nadam
set.seed(724) # Reproduce results
n = 1000 # Number of iterations
pars = rnorm(1000) # Generate random number using normal distribution (Initial parameter val
output = matrix(nrow=n, ncol=1000)
output[1,] = pars #matrix giving the distribution parameter values for each instance in the
nadam = Nadam$new(eta=0.5)
for (i in 2:n) {
# Gradient for quadratic cost
grad = pars + rnorm(1000)
pars = pars + nadam$step(grad)
output[i,] = pars
}
plot(output, type='l')



##RMSprop
rmsProp = setRefClass("rmsProp",
fields = list(eta="numeric", grad_squared="numeric", epsilon="numeric", acc_grad="numeric"),
methods = list(
step = function(gradient) {
acc_grad <<- grad_squared * acc_grad + (1-grad_squared)*c(gradient)^2
inc =  - (eta  / sqrt(acc_grad+epsilon)) *  gradient
return(inc)
},
initialize = function(eta=1E-3, grad_squared=0.9, epsilon=1E-6) { ##Defaults taken from a mi
.self$eta = eta
.self$grad_squared = grad_squared
.self$epsilon = epsilon
.self$acc_grad = 0 ##acc_grad start off as a scalar but becomes a vector of the right length
}
)
)


#Example RMSprop
set.seed(724) # Reproduce results
n = 1000 # Number of iterations
pars = rnorm(1000) # Generate random number using normal distribution (Initial parameter val
output = matrix(nrow=n, ncol=1000)
output[1,] = pars #matrix giving the distribution parameter values for each instance in the
rms = rmsProp$new(eta=0.5)
for (i in 2:n) {
# Gradient for quadratic cost
```

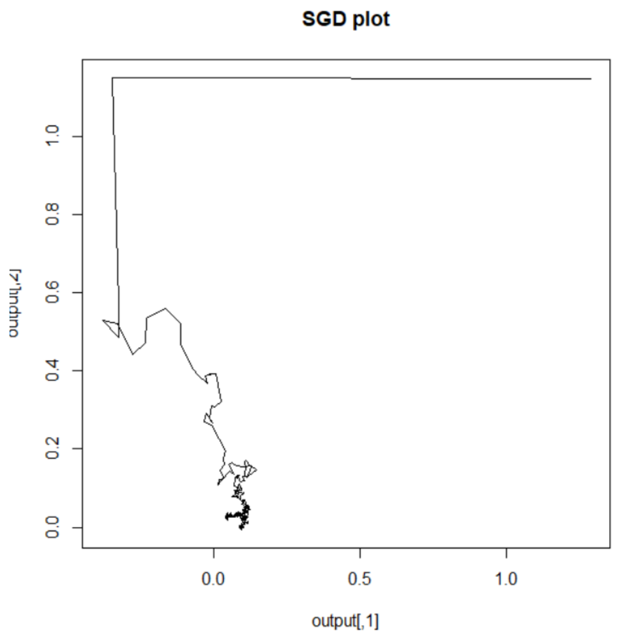


Figure 25: SGD plot

```
grad = pars + rnorm(1000)
pars = pars + rms$step(grad)
output[i,] = pars
}
plot(output, type='l')
```

We now include the plot of every algorithm we have found in Figure 25 to 30. We set up our algorithms with the same parameters value for the numerical examples so we can compare how those different optimization algorithms are working. We can see that despite all of them using gradient the results are very sparse and we can understand why some algorithms generalize better in some cases than others.

26 Conclusion

ADAM is one of the best deep learning optimization algorithms, and its popularity is rapidly expanding. While some users have reported issues with ADAM in specific areas like generalization in some cases or the convergence under convexity , researchers are working on methods to bring Adam's results up to par with SGD's with momentum.

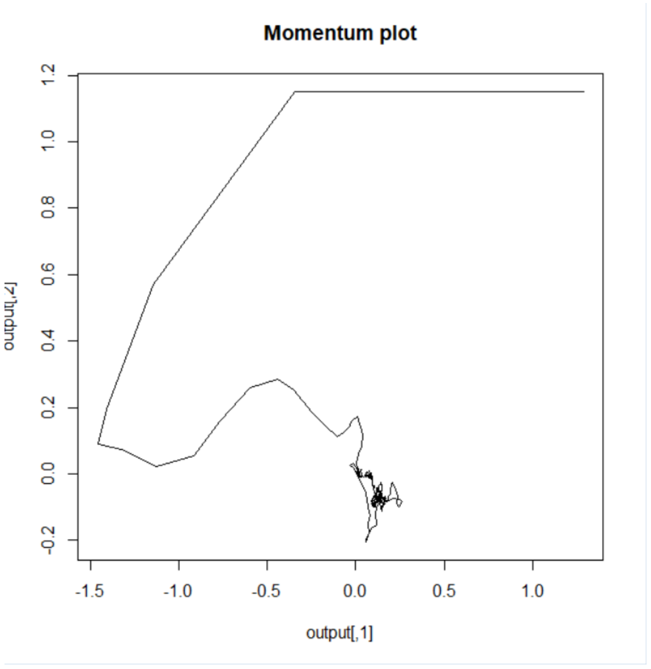


Figure 26: Momentum plot

References

Bock, S., Goppold, J., & Weiß, M. (2018). An improvement of the convergence proof of the adam-optimizer. *arXiv preprint arXiv:1804.10587*.

Boyd, S., Boyd, S. P., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

Bubeck, S. (2014). Convex optimization: Algorithms and complexity. *arXiv preprint arXiv:1405.4980*.

Bushaev, V. (2018). Adam—latest trends in deep learning optimization. *Towards Data Science, Listopad*.

D, F., & DQ, N. (n.d.). An introduction to vectors. *Math Insight*. Retrieved from `http://mathinsight.org/vector`$_i$*ntroduction*

Dozat, T. (2016). Incorporating nesterov momentum into adam.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, *12*(7).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

He, J., Rexford, J., & Chiang, M. (2010, 01). Design for optimizability: Traffic management of a future internet. In (p. 3-18). doi: 10.1007/978-1-84882-765-3$_1$

Higham, C. F., & Higham, D. J. (2019). Deep learning: An introduction for

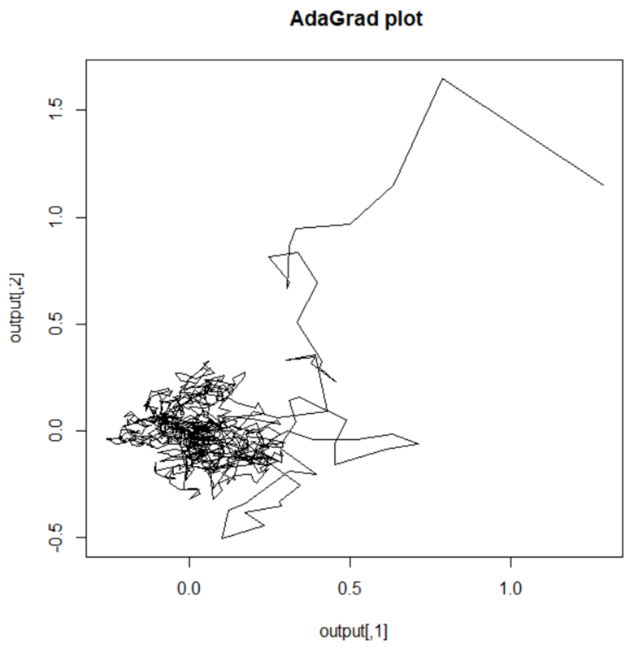


Figure 27: AdaGrad plot

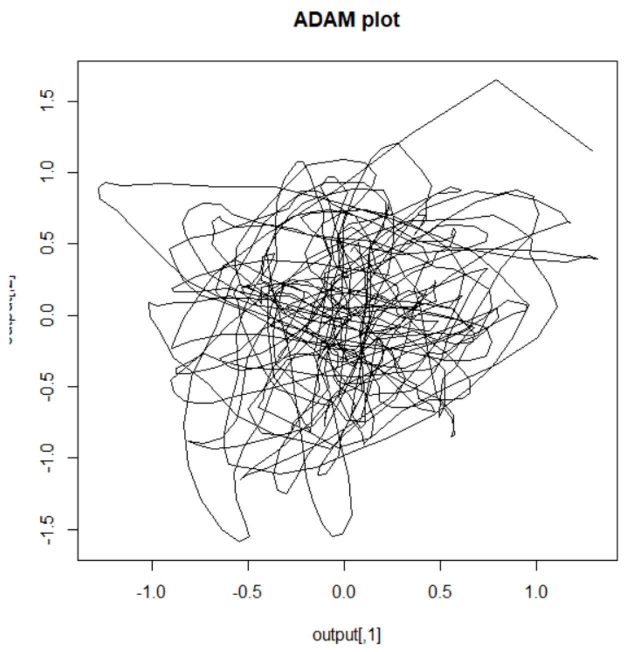


Figure 28: ADAM plot

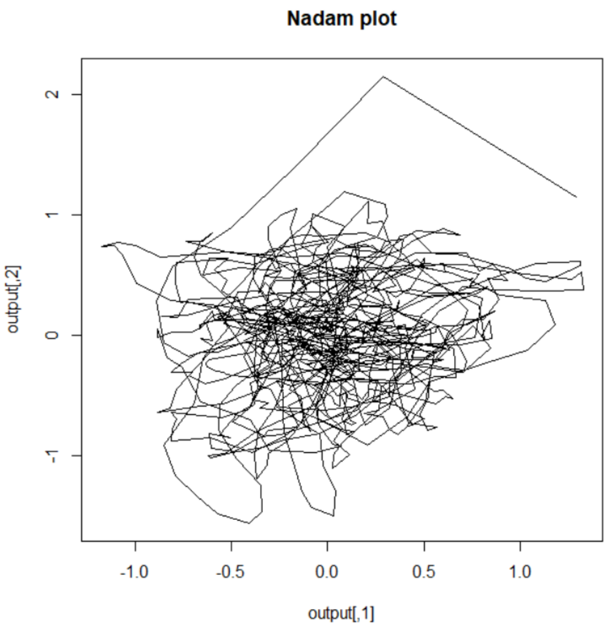


Figure 29: Nadam plot

applied mathematicians. *Siam review*, *61*(4), 860–891.

Jayawardana, R., & Bandaranayake, T. (2021, 04). *Analysis of optimizing neural networks and artificial intelligent models for guidance, control, and navigation systems.*

Jones, M. S. (2015). Convolutional autoencoders in python/theano/lasagne. *Blog post (retrieved February 17, 2016), April*.

Karpathy, A. (2017). A peek at trends in machine learning. *Medium. com*.

Keskar, N. S., & Socher, R. (2017). Improving generalization performance by switching from adam to sgd. *arXiv preprint arXiv:1712.07628*.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

Loshchilov, I., & Hutter, F. (2018). Fixing weight decay regularization in adam.

Maas, A., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies* (pp. 142–150).

McDonald, C. (2017). Machine learning fundamentals (i): Cost functions and gradient descent. *Towards Data Science*, *27*.

Mostafa, B. M., El-Attar, N., Abd-Elhafeez, S., & Awad, W. (2021). Machine

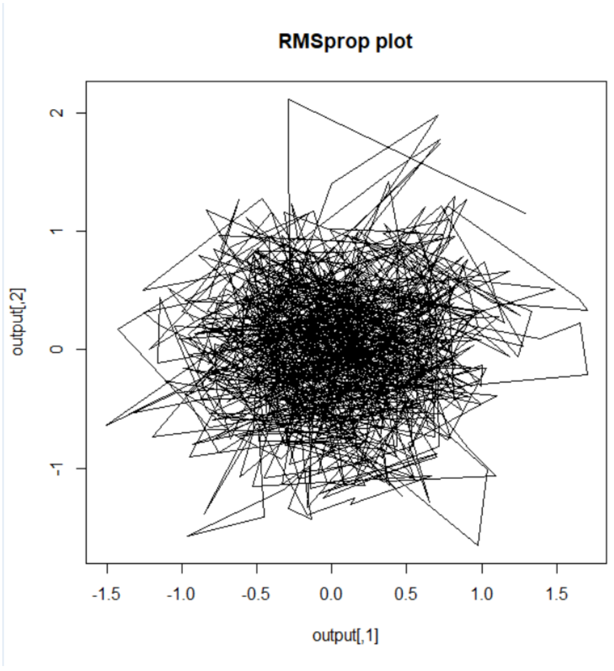


Figure 30: RMSprop plot

and deep learning approaches in genome. *Alfarama Journal of Basic & Applied Sciences*, *2*(1), 105–113.

Nielsen, M. A. (2015). *Neural networks and deep learning* (Vol. 25). Determination press San Francisco, CA.

Pekel, E., & Soner Kara, S. (2017). A comprehensive review for artificial neural network application to public transportation. *Sigma: Journal of Engineering & Natural Sciences/Mühendislik ve Fen Bilimleri Dergisi*, *35*(1).

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, *4*(5), 1–17.

Pérez-Enciso, & Zingaretti, L. (2019, 07). A guide for using deep learning for complex trait genomic prediction. *Genes*, *10*, 553. doi: 10.3390/genes10070553

Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms.* Cambridge university press.

Sharma, S., Sharma, S., & Athaiya, A. (2017). Activation functions in neural networks. *towards data science*, *6*(12), 310–316.

Sohl-Dickstein, J., Poole, B., & Ganguli, S. (2014). Fast large-scale optimization by unifying stochastic gradient and quasi-newton methods. In *International conference on machine learning* (pp. 604–612).

Wang, S., & Manning, C. (2013). Fast dropout training. In *international conference on machine learning* (pp. 118–126).

Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., & Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning. *Advances in neural information processing systems*, *30*.

Zagoruyko, S. (2015). *Torch blog.*

Zhang, Z., Ma, L., Li, Z., & Wu, C. (2017). Normalized direction-preserving adam. *arXiv preprint arXiv:1709.04546*.

Zhou, B.-c., Han, C.-y., & Guo, T.-d. (2021). Convergence of stochastic gradient descent in deep neural network. *Acta Mathematicae Applicatae Sinica, English Series*, *37*(1), 126–136.

Zinkevich, M. (2003). Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th international conference on machine learning (icml-03)* (pp. 928–936).