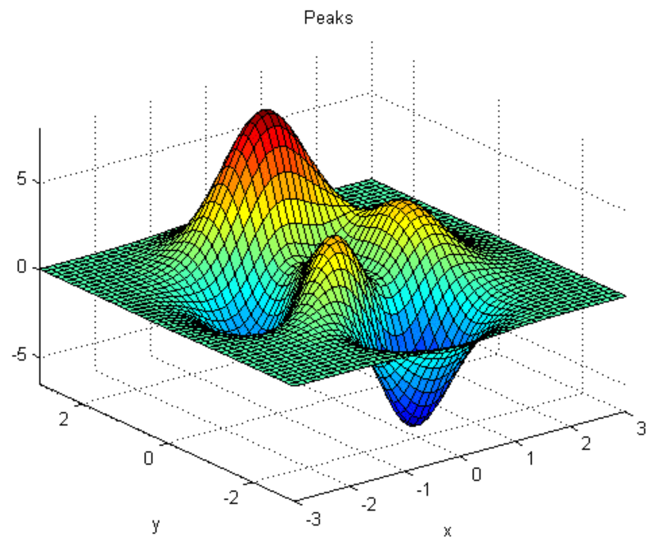
Zaven and Sonia Akian College of Science and Engineering (CSE)
03/05/2019

Computer Science
CS213 Optimization
Section B

Optimization Algorithms for Neural Networks

Made by:
Robert Gevorgyan, Maro Grigoryan,
Harutyun Gevorgyan



Contents

1 Introduction

The paper will discuss Neural Networks, what neural networks are, how they are used and then move on to the optimization algorithms for Neural Networks. We will specifically discuss Gradient Descent and mainly talk about optimizing ordinary Gradient Descent. The paper is written as a final project for CS213 Optimization course. We chose the topic because we wanted to learn more about Neural Networks and use this project as an opportunity to dig deeper into Machine/Deep Learning. The main concern of this paper will be Gradient Descent optimization and will try to find an answer to the question "Which GD algorithms are the best fit for Neural Networks optimization". We spent the majority of our time on reading research papers connected to the topic and critically comparing some of the papers to each other. We tried to find as much of different optimization ways as possible. Besides reading the papers we tried to implement those algorithms using Python and *Matlab*. Later on we started working on visualizations and after that on using these algorithms to implement a neural network which will solve the predefined task. We used the help of Machine Learning community, even talking and asking help from some of the leaders of the field. They provided us with some leadership, ideas and directions.

2 Project

2.1 Introduction to Neural Networks

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

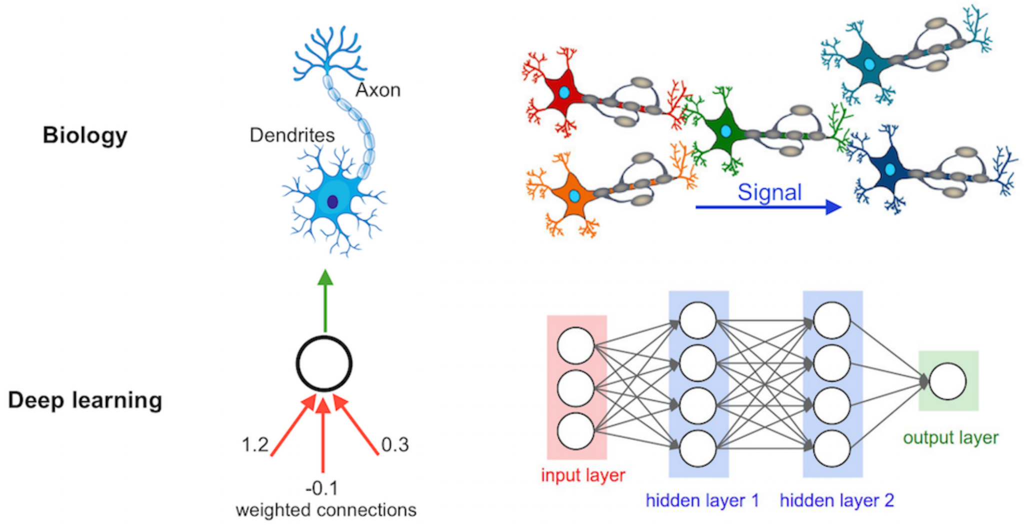


Figure 1: Biological Neuron and Deep Learning analog

Before we jump to optimization algorithms for neural nets, firstly we need to understand how the cost function is generated. This is why we decided to put some brief explanation of the structure of neural net.

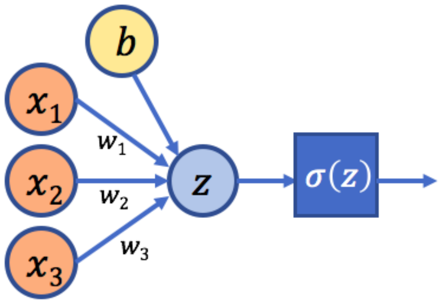


Figure 2: Cost Function

This function Z is defined as a weighted sum of its inputs—we multiply the inputs x by variables w called weights to get the output. You can think of weights as the strengths of the connections between the input and the output. For example, if w1 has a higher value than w2, that implies that the input x1 influences the output more than x2 does. The value b is called the bias term and is responsible for shifting the function so that it's not constrained to the origin. Now, the value of Z can be anything ranging from -$\propto$ to +$\propto$.To put limitations for the final decision "activation functions" were introduced. Several such functions are used in practice: $Sigmoid$, $ReLu$, $Tanh$, etc.

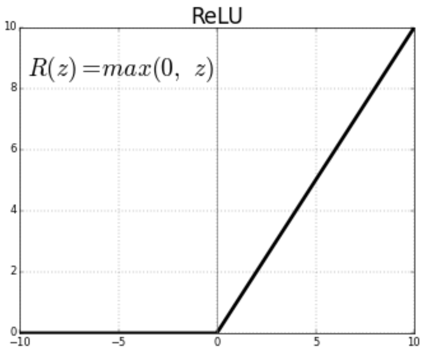


Figure 3: ReLU Graph

The $ReLu$ function is as shown above. It gives an output x if x is positive and 0 otherwise.

Finally, ANN is simply a set of connected neurons organized in layers: input layer, hidden layer, output layer. Now the most important question: how do we choose the weights w1, w2, ... The weights are initialized to small random numbers and then the cost function is constructed. The main task remains to minimize the cost function and appropriately update the weights of features. Here is an example of a simple neural network. We have chosen MSE (Mean squared error) as a cost and $ReLu$ as an activation function . MSE simply squares the difference between every network output and true label, and takes the average. Now the main problem that remains unsolved is to find a way to solve is the minimization problem of our cost function.

$$x = [x_1, x_2, x_3, ..., x_n]^T$$

$$Cost(Y, Z) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - Z_i)^2$$

$$Cost(Y, X, w, b) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - activation(X_i))^2$$

$$Cost(Y, X, w, b) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - max(0, w * X_i + b))^2$$

Problem: Minimize Cost function

There are many different mathematical methods to solve this problem. However, the most efficient and simple way is the Gradient Descent, and this is why the GD methods are the widely used minimization methods for Neural Networks.

2.2 General Gradient descent

2.2.1 Overview

As we already said, Gradient Descent is an optimization algorithm which is used to minimize functions. The general idea is that we move towards the opposite direction of the gradient. The direction of gradient shows the direction of increase of the subject function, so moving in opposite direction of gradient makes much sense. Now Gradient Descent is too broad, so it has many different ways of implementing. Those implementations depend on the size of the data, dimensions, and the specific use case. However, all of this methods follow a general formula.

$$x_n = x_{n-1} - \gamma \Delta F(x_{n-1}); n >= 0$$

Here x-es are our minimizer estimations, and gamma is the learning rate or step size.
In the context of deep learning F is our Cost function, x-es are our weights and gamma is the learning rate which is usually denoted as alpha instead.

$$w_n = w_{n-1} - \alpha \cdot \Delta Cost(w_{n-1}); n >= 0$$

Some of the Gradient Descent methods depending on the step size are Steepest Descent, Newton's method, Conjugate Gradient Methods, etc. Depending on the batch size (i.e. the group of the data points we take) there is Batch Gradient Descent, Mini-Batch GD and Stochastic GD. Plus to this there are various possible optimizations to the GD which will be discussed later in this paper.

2.2.2 Types of gradient descent based on the size of training data (Batch gradient descent, Mini batch gradient descent, Stochastic gradient descent)

- Batch Gradient Descent

Batch gradient descent, or vanilla (i.e. pure, original) gradient descent, has the same formula as the general formula mentioned above. It computes the gradient of the function with respect to the WHOLE set of data points.

$$w_n = w_{n-1} - \alpha \cdot \Delta Cost(w_{n-1}); n >= 0$$

Mathematically, this is the best way to handle the problem as the batch gradient descent surely converges to a minimizer (be it local or, in case of a convex function/surface, even global). However, practically speaking it's difficult to use. For one iteration, we need to go over all the data points. This creates different problems - memory insufficiency, slow update speed, and being unable to update the model we have in case of adding new data points. We can imaging reading an enormous dataset into memory and computing partial derivatives of the huge cost function. Another approaches exist to solve the problem of computational intensity.

- Mini Batch Gradient Descent

The most obvious workarounds for the above mentioned batch GD problems would be taking small batches. That's exactly what Mini Batch GD does! It takes a small random subset from the whole set of data points (the subset size in practice usually varies from 50 to 256) and performs an update using this small batch.
This method creates its own challenges. As we do not use the whole set, we are not able to ensure good convergence. Besides that, choosing a learning rate can become a challenge when using Mini-batch GD, as too small step size will give us slow response time and too big step size will cause a lot of fluctuation and overshooting and even missing the minimizer which may ultimately bring to divergence. The other problem is that Mini-batch GD tends to get stuck in the proximity of a saddle point as the gradient close to the saddle point is usually very small (close to zero vector).
As the subset is chosen randomly, Mini Batch GD in practice is sometimes also referred as Stochastic GD. However, the real SGD is a little bit different.

$$w_n = w_{n-1} - \alpha \cdot \Delta Cost(w_{n-1}; x^{(i:i+n)}; y^{(i:i+n)}); n >= 0$$

- Stochastic Gradient Descent

Stochastic gradient descent, aka SGD updates parameters for every single data point one by one.

$$w_n = w_{n-1} - \alpha \cdot \Delta Cost(w_{n-1}; x^{(i)}; y^{(i)}); n >= 0$$

This is the other extreme to Batch GD. It solves some of the problems of Batch GD such as speed of learning and on-the-fly updating issue, but it also has a tons of problems itself. Some of those problems are in common with Mini-Batch GD - high variance jumps which cause heavy fluctuations in the objective function, overshooting complications. Stochastic Gradient Descent (as well as Mini-Batch GD) is more risky and may be more rewarding as it has a chance to jump out of the local minima areas and probably find a better minimizer. However, this risk may also bring us to the opposite thing - complicating convergence. As a workaround to this, practice shows that when slowly tuning our learning rate by decreasing it, SGD shows almost the same convergence as vanilla GD. As you can see, the Mini-Batch GD is the golden middle of Batch GD and SGD, so this is why it's the most used in practice.

2.3 Challenges (Tuning parameters, convergence)

So here let's summarize the challenges we face when using GD. The problems of Batch GD is partially fixed with using Mini-Batch GD or SG. As we can notice, SGD fixes speed and memory issues in Batch gradient descent but fails in other matters. Mini Batch GD deals with high variance updates in SGD and at the same time introduces new challenges. Bellow is the short pro and con list for each of the GD type.

1. Batch GD
   - Pros:
     - Guaranteed to converge to global minimum for convex surfaces and to a local minimum for non-convex surfaces.
   - Cons:
     - Very slow
     - High memory usage
     - Does not work in case of large datasets (running out of memory)
     - No online learning
2. Stochastic Batch GD
   - Pros:
     - Much faster than Batch gradient descent
     - Low memory usage
     - Allows online learning
   - Cons:
     - High variance updates
     - Sometimes convergence issue
3. Mini Batch GD
   - Pros:
     - Reduces variance of updates
     - Allows online learning

- Cons:
  - Batch size is an hyper parameter and its choice can be quite challenging

Summarizing the challenges:

1. Parameter Tuning an adequate learning rate so that we can avoid both slow convergence and high fluctuation.
2. Possibility to adapt to the dataset - We usually do not know the dataset completely prior to training, so the workaround of "decreasing the learning rate slowly to ensure Batch GD-like convergence" does not work well in practice, as we cannot schedule those tunings because we don't know the data at start.
3. Sparse dataset problem - Not all of the features have the same value/importance to us. We need to be aware of our features and be able to define learning rate for each of them separately.
4. Local minima and saddle point traps - We want our algorithm to have a way to be able to avoid not-so-good local minima and to be able to get out of the neighborhoods of saddle points.

2.4 SGD algorithms

These problems were known to specialists for quite some time and they created several algorithms which try to solve those problems. Those algorithms play with the general formula - step size, direction we choose etc. Let's take a more detailed look.

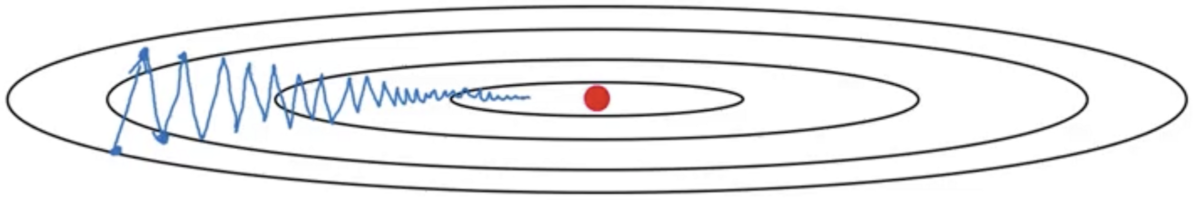


Figure 4: Source: Andrew Ng

2.4.1 List of SGD algorithms

- Momentum
- Nesterov Accelerated Gradient
- Adagrad
- Adadelta
- RMSprop
- Adam
- Adam extensions

2.4.2 Thorough explanation of some SGD algorithms

- Momentum SGD

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.

$$v_n = \beta \cdot v_{n-1} + (1-\beta) \cdot \Delta Cost(w_{n-1})$$

$$w_n = w_{n-1} - \alpha \cdot v_n; n >= 0$$

We take an exponential average of the gradient steps.

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot \Delta Cost(w_t)$$

$$v_{t_1} = \beta \cdot v_{t-2} + (1-\beta) \cdot \Delta Cost(w_{t-1})$$

$$v_{t_2} = \beta \cdot v_{t-3} + (1-\beta) \cdot \Delta Cost(w_{t-2})$$

$$v_t = \beta\beta(1-\beta) \cdot w_{t-2} + ... + \beta(1-\beta) \cdot w_{t-1} + ... + (1-\beta)w_t$$

Now we can see that the current value of v_t is dependent on all previous values of v, which got coefficients (weights) in the sum. This weight is β to power of i multiplied by (1- β) for (t - i)th value of the gradient. As β is a number between 0 and 1, older v_t-s get smaller and smaller coefficients. At some point it becomes so small, that we can even neglect it. This way it is easy to see that the momentum is a weighted average of the gradients.
So why do we need momentum in SGD? To put it easier, momentum helps us to keep the speed of the progress in fast directions and speed up the progress in slow directions. The image below shows how momentum helps us to do that.

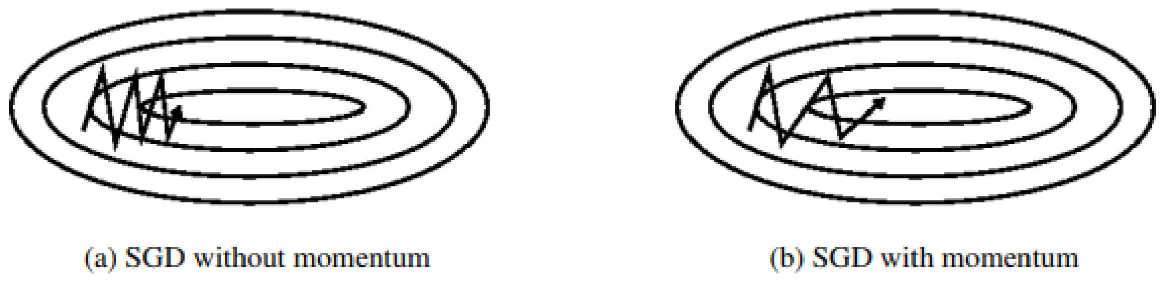


Figure 5: Source: Genevieve B. Orr

As you can see the updates in horizontal direction without momentum are very slow while the updates of vertical direction are big. This causes that unnecessary fluctuation which slows the convergence to the minimum. Momentum here helps to reduce the step size in vertical direction, as we sum together many vectors that point towards opposite directions and increase the step size in horizontal direction, because we sum vectors that face the same direction. The well-known analogy to momentum that helps to understand it better is a ball rolling down the hill. We hold the ball in 0 speed and release

it. The ball accumulates kinetic energy as it rolls down, making it faster. In this analogy, beta is the air resistance, which helps to bring the ball to terminal state in some moment of time n.
Take a look at the algorithm visualizations below to see how close the convergence of momentum resembles the rolling ball analogy.

- RMSProp

We already saw how momentum can speed up gradient descent. There is also another algorithm called *RMSProp* that is widely used in deep learning applications. The name stands for Root Mean Square Prop. As we saw before, the main problem with Gradient Descent was the possibility of huge oscillation in vertical direction and slow progress in horizontal direction and this is what *RMSProp* is intended to solve.

$$s_n = \beta \cdot s_{n-1} + (1-\beta) \cdot \Delta Cost(w_{n-1})^2$$

$$w_n = w_{n-1} - \alpha \cdot \frac{\Delta Cost(w_{n-1})}{\sqrt{s_n}}$$

The intuition is similar to momentum, but *RMSProp* works on amplifying the slow directions and slowing the fast directions in a different manner. In slow directions, s_n will be a small number. As it is in the denominator of the fraction, the update in that direction will be bigger. For fast directions, the *RMSProp* causes the opposite effect.

In practice, a very small positive parameter ε is added to s_{dw} just to avoid 0 in the denominator

$$w_n = w_{n-1} - \alpha \cdot \frac{\Delta Cost(w_{n-1})}{\sqrt{s_n} + \varepsilon}$$

RMSprop divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests β to be set to 0.9, while a tested good default value for the learning rate α is 0.001

- Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like *RMSprop*, Adam also keeps an exponentially decaying average of past gradients, similar to momentum. Taking into account this description one can imagine Adam as a combination of previous two: Momentum and *RMSprop*. The algorithm leverages the power of adaptive learning rates methods to find individual learning rates for each parameter. 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. Having both of these enables us to use Adam for broader range of tasks.

$$v_n = \beta_1 \cdot v_{n-1} + (1-\beta_1) \cdot \Delta Cost(w_{n-1})$$

$$s_n = \beta_2 \cdot s_{n-1} + (1-\beta_2) \cdot \Delta Cost(w_{n-1})^2$$

$$w_n = w_{n-1} - \alpha \cdot \frac{v_n}{\sqrt{s_n} + \varepsilon}$$

As you can see the ε trick is done here too.

One other trick that is usually used in case of Adam is bias correction. This is done because especially at start when the v_n and s_n are small, they are biased towards zero.

$$v_n = \beta_1 \cdot v_{n-1} + (1 - \beta_1) \cdot \Delta Cost(w_{n-1})$$

$$s_n = \beta_2 \cdot s_{n-1} + (1 - \beta_2) \cdot \Delta Cost(w_{n-1})^2$$

$$v_n^{corr} = \frac{v_n}{1 - \beta_1^n}$$

$$s_n^{corr} = \frac{s_n}{1 - \beta_2^n}$$

So the update now will be:

$$w_n = w_{n-1} - \alpha \cdot \frac{v_n^{corr}}{\sqrt{s_n^{corr}} + \varepsilon}$$

- α - Initial learning rate
- v_n - Exponential average of gradients with respect to w
- s_n - Exponential average of squares of gradients with respect to w
- β_1, β_2 - hyperparameters

Adam is a relatively new algorithm and it's faster than the previous two because of one simple reason: it takes both momentum and RMSProp and merges them together.

In practice, the most common value for β_1 is 0.9, for β_2 is 0.999 and 10^{-8} for ε.

2.4.3 Implementation

We included implementations of above described algorithms for further exploration and analysis. There are many efficient implementations of those algorithms in many programming languages and libraries but for this report standard MATLAB programs were developed. Let's take a look at them one by one.

- Momentum SGD

Here is the matlab code for Momentum SGD in Matlab:

```matlab
function result = momentum(f, params, weights, alpha, b)
    disp('Initial values are:');
    disp(weights);
    % f is the concerned function
    % params are symbolic variables
    % weights are values vector
    % we will use termination condition
    % when gradient is smaller than ep
    % we stop iterating
    n = 0; % step counter
    % we want to return the final weights
    % but do not want to mutate the inputs
    result = weights;
    % calculating the symbolic gradient
    % w.r.t params
    grads = gradient(f, params);
    % we do not have initial "velocity"
    v = 0;
    ep = 1e-5;
    while norm(double(subs(grads, params, result))) >= ep
            n = n + 1;
            % we evaluate the value of
            % the gradient using subs to substitute
            % params with result vector's values in
            % gradient vector
            % as it's a symbolic value, we want to turn
            % it back to double value, hence the double
            % ' at the end takes the transpose
            % as the result is a column vector but we need
            % a row vector
            grad_vals = double(subs(grads, params, result))';
            % calculating the momentum
            v = (b * v) + ((1 - b) * grad_vals);
            % performing the actual update;
            result = result - (alpha * v);
    end
    % just to display number of steps
    disp(['Number of steps: ', num2str(n)]);
    disp('Calculated Weights are:');
    disp(result);
    disp('');
end
```

Let's quickly go over the code. We have a function which return the calculated weights which are stored in result vector while the function operates. We use symbolic toolbox in matlab to create symbolic function f and symbolic params and pass them to momentum function alongside the initial weights vector and hyperparameters alpha and beta. There is a counter n which is specified for us to keep track of each algorithm and to see how many steps it takes for each algorithm to reach to the minimum.
We will show the usage of those functions in the next subsection.

- RMSProp

As you may guess, the code of RMSProp is pretty similar to the Momentum code. Technical part is almost the same, we will change just two lines and get the RMSProp algorithm.

```
function result = RMSProp(f, params, weights, alpha, b, e)
    disp('Initial values are:');
    disp(weights);
    n = 0;
    result = weights;
    grads = gradient(f, params);
    s = 0;
    ep = 1e-5;
    while norm(double(subs(grads, params, result))) >= ep
            n = n + 1;
            grad_vals = double(subs(grads, params, result))';
            % the only part different from
            % the previous algo is here
            s = (b * s) + ((1 - b) * grad_vals .^ 2);
            % in case you want to do bias correction
            % s_corr = s / (1 - b ^ n);
            step = grad_vals ./ (sqrt(s) + e); % sqrt(s_corr)
            result = result - (alpha * step);
    end
    disp(['Number of steps: ', num2str(n)]);
    disp('Calculated Weights are:');
    disp(result);
    disp('');
end
```

The only part that was changed here is that instead of v we now calculate the RMS (named s in the code) and use it to calculate the step which is later used to update the weights of our parameters. As you can see the squaring and dividing is done *elementwise*.
As you can see, we added e in the denominator of *step* to avoid 0 in the denominator. If we got 0 in denominator, the result of that would be Inf (infinity), so the step will always stay 0 and the program will run forever (as, eventually we will update weights by 0 step every time).

- Adam

We can get the code for Adam by simply combining the previous two algorithms together correctly:

```
function result = adam(f, params, weights, alpha, b1, b2, e)
    disp('Initial values are:');
    disp(weights);
    n = 0;
    result = weights;
    grads = gradient(f, params);
    v = 0;
    s = 0;
    ep = 1e-5;
    while norm(double(subs(grads, params, result))) >= ep
            n = n + 1;
            grad_vals = double(subs(grads, params, result))';
            % we calculate v as in momentum
            v = (b1 * v) + ((1 - b1) * grad_vals);
            % and s as in RMSProp
            s = (b2 * s) + ((1 - b2) * grad_vals .^ 2);
            % bias corrections
            v_corr = v / (1 - b1 ^ n);
            s_corr = s / (1 - b2 ^ n);
            % now we use both in a combined step
            step = v_corr ./ (sqrt(s_corr) + e);
            % and then update weights
            result = result - (alpha * step);
    end
    disp(['Number of steps: ', num2str(n)]);
    disp('Calculated Weights are:');
    disp(result);
    disp('');
end
```

Here we use both v and s. One nuance here that is not done in previous cases (but could be done) is bias correction. It simply follows the formulas mentioned earlier. Notice how similar the step is to the step of RMSProp, but instead of $grad_vals$ we have the bias corrected momentum v_corr here.

2.4.4 Testing and comparison with visualizations

In this section we will analyze the performance of above described 3 algorithms (SGD with Momentum, RMSprop and Adam) and perform comparisons.
The first function we will work on is[1] :

$$f(x_1, x_2) = -2 * e^{-\frac{(x_1-1)^2+x_2^2}{0.2}} - 3 * e^{-\frac{(x_1+1)^2+x_2^2}{0.2}} + x_1^2 + x_2^2$$

The first thing we will do is to try to see what the graph of this function looks like. We will draw the surface plot and the level curves. For drawing the surface plot:

[1] This function is taken from https://bl.ocks.org/EmilienDupont/aaf429be5705b219aaaf8d691e27ca87

```
f = @(x1, x2) -2 .* exp(-((x1 - 1).^2 + x2.^2) ./ .2) + -3 .* exp(-((x1 + 1).^2 + x2.^2) ./ .2) + x1.^2 + x2.^2;
[X_3, Y_3] = meshgrid(-2:0.1:2);
z = f(X_3, Y_3);
colormap(jet);
surf(X_3, Y_3, z);
```

The result looks like this.

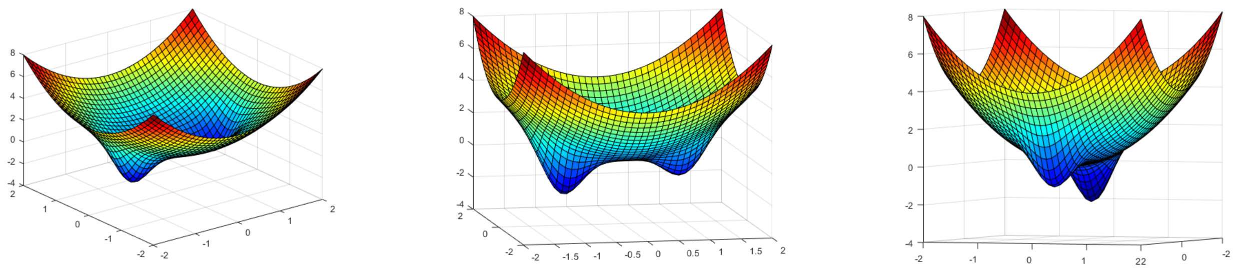


Figure 7: Surface plot from different perspectives

As you can see, the function has 2 local minimums. One of them gets a smaller value.
We chose this function primarily because, if right starting point is chosen, some of the algorithms will converge to the local minimum while others will escape it and converge to the global minimum.
Let's also take a look at level curves.

```
f = @(x1, x2) -2 .* exp(-((x1 - 1).^2 + x2.^2) ./ .2) + -3 .* exp(-((x1 + 1).^2 + x2.^2) ./ .2) + x1.^2 + x2.^2;
[X, Y] = meshgrid(-2:0.001:2);
z = f(X,Y);
contourf(X,Y,z,10);
```

The code will draw 10 level curves of the function, automatically choosing the levels. The result of this is shown in the figure below.

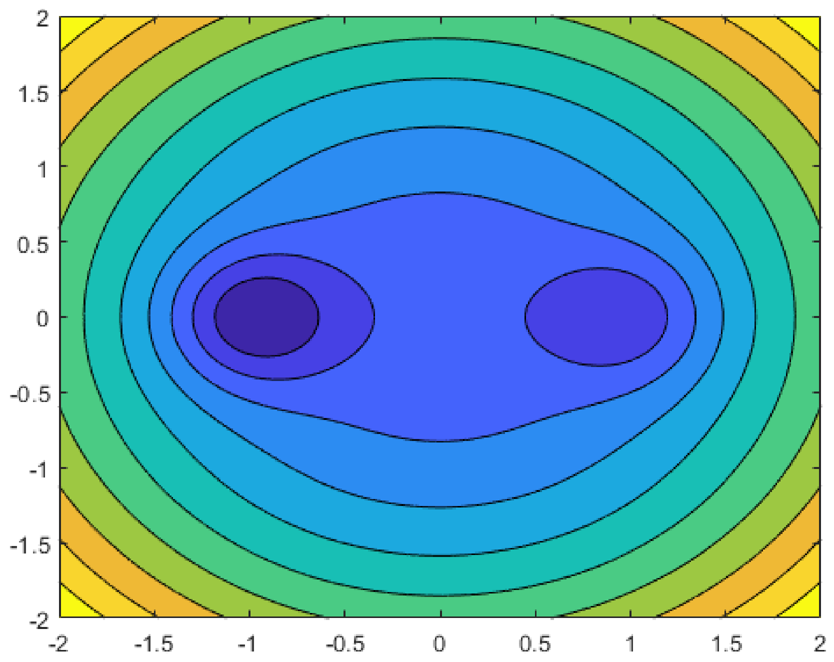


Figure 7: Level Curves of the First Test Function

As you can see from the curve, the function has 2 minimum points. To find those points we will use Matlab's built-in *fminsearch* first, before starting to test our own algorithms.

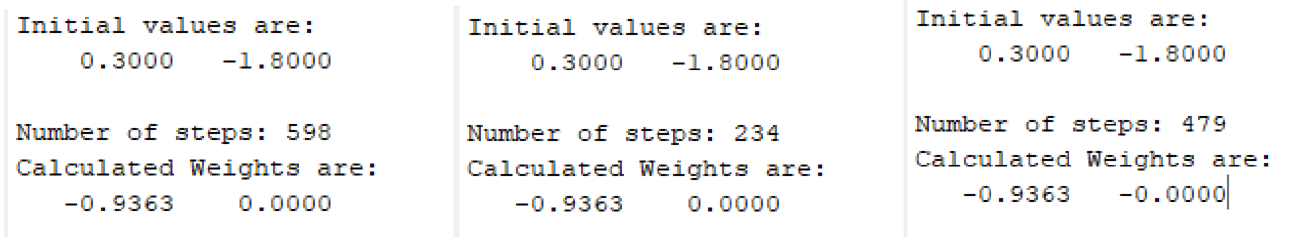
```
f = @(x) -2 .* exp(-((x(1) - 1).^2 + x(2).^2) ./ .2) + -3 .* exp(-((x(1) + 1).^2 + x(2).^2) ./ .2) + x
    (1).^2 + x(2).^2;
fminsearch(f, [-2, 0])
fminsearch(f, [0.5, 1])
```

We picked 2 starting points using the graph to be sure that they converge to different minimum points. The output of the code shows that minimum points are approximately $[-0.9363, 0]$ and $[0.9053, 0]$. The first one is the global minimum point.
Now let's try our functions with different initial points.

```
syms x1 x2;
f = -2 * exp(-((x1 - 1)^2 + x2^2) / .2) + -3 * exp(-((x1 + 1)^2 + x2^2) / .2) + x1^2 + x2^2;
alpha = 1e-2;
b1 = 0.9;
b2 = 0.999;
e = 1e-8;
params = [x1, x2];
testVals1 = [0.3, -1.8];
testVals2 = [0.79, -1.7];
testVals3 = [-0.3, -1.72];

momentum(f, params, testVals1, alpha, b1);
RMSProp(f, params, testVals1, alpha, b1, e);
adam(f, params, testVals1, alpha, b1, b2, e);
```

You can play with the *testVals* to see the differences. Now for the initial point $[0.3, -1.8]$ we have the following results: The result looks like this.



```
Initial values are:
     0.3000    -1.8000

Number of steps: 598
Calculated Weights are:
    -0.9363     0.0000
```

```
Initial values are:
     0.3000    -1.8000

Number of steps: 234
Calculated Weights are:
    -0.9363     0.0000
```

```
Initial values are:
     0.3000    -1.8000

Number of steps: 479
Calculated Weights are:
    -0.9363    -0.0000
```

Figure 8: Results of momentum RMSProp and Adam respectively

Here is the path each of the algorithms covers.

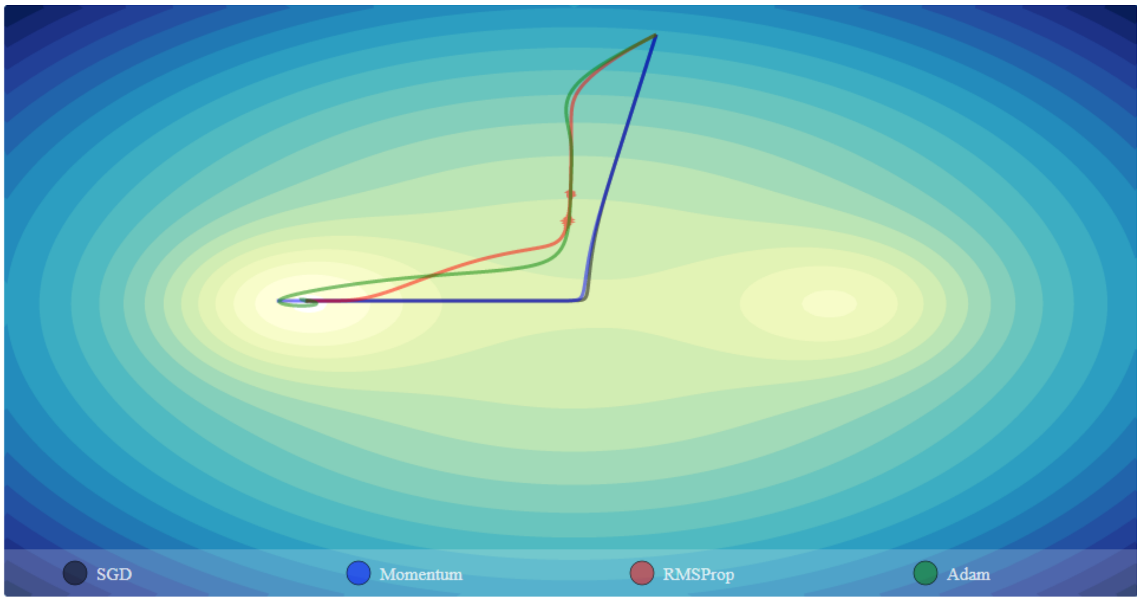


Figure 9: Covered paths from $[0.3, -1.8]$

As you can see Adam and *RMSProp* "realized" the direction faster and converged faster. Vanilla *SGD* is there to show the contrast with optimization algorithms.
Now let's take a point which will make some of the arguments converge to the local minimum point instead of the global one. For that we need to take, for example, *testVal2*. However, in this case RMSProp starts to fluctuate around the minimum without reaching the needed precision. To avoid this we may specify step limit, let's say to stop iterating after 2000 steps. This problem only exists in *Matlab*. The same algorighm works perfectly in *Python* and even in *JavaScript*. The results of the second test are as following:

```
Momentum N of Steps: 252
RMSProp N of Steps: 222
Adam N of Steps: 147
```

As you can see, Adam did almost two times well than the other too. As expected, momentum is the slowest, and which is even worse, it converges to the other, local minimum instead! Here is the paths the algorithms cover:

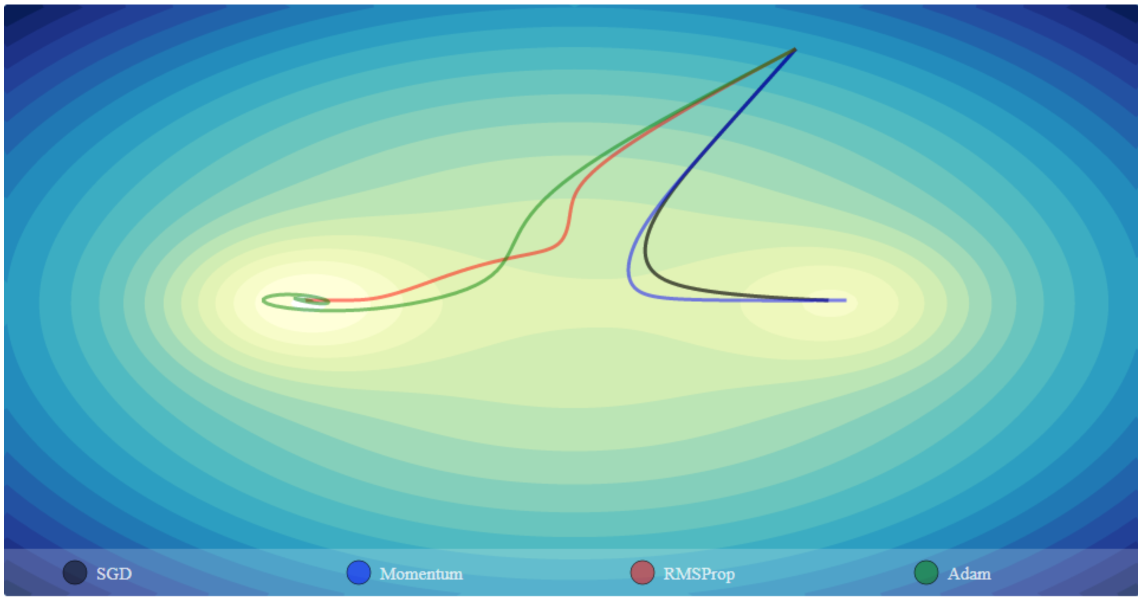


Figure 10: Covered paths from $[0.79, -1.7]$

SimpleSGD as well as *MomentumSGD* couldn't escape the trap and converged to the wrong minimum point while *Adam* and *RMSProp* got out and converged to the right point. Majority of points in the set act quite similarly. Let's take just one of such "regular" points to generalize the results. We take $[-0.3, -1.72]$ as that point. We get the following results:

```
Momentum N of Steps: 281
RMSProp N of Steps: 218
Adam N of Steps: 148
```

And here are the paths:

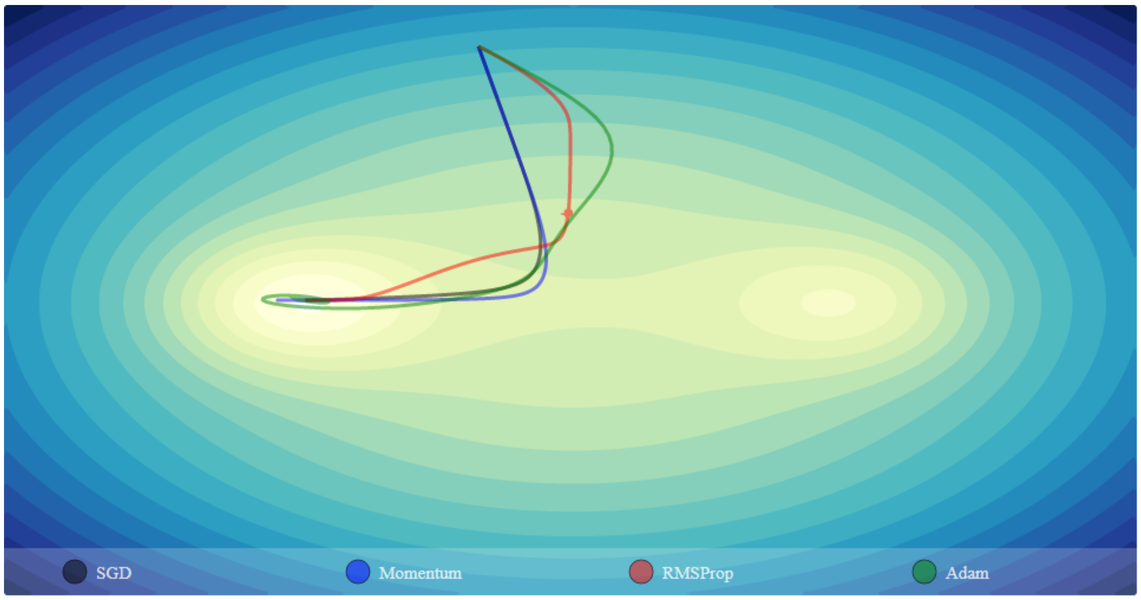


Figure 11: Covered paths from $[-0.3, -1.72]$

All of the algorithms successfully found the right minimum point and Adam again is the fastest one. So, generally, Adam will do a very good job finding the right minimum and doing it fast, except some exceptional cases, where it may fail to find the right minimum point or be slower than the other algorithms. RMSProp again is fairly consistent, but usually does not reach the same speed as Adam does.

Other visualizations (Yuret Radford, 2015)[2] also show that Adaptive algorithms like Adam tend to move faster and navigate difficult portions of functions better.
NOTE: If you want to play around and see more results you are welcome to check our HTML/JS visualization tool. It is based on Emilien Dupont's work, but we changed some part of algorithms and the termination conditions:
https://github.com/robgev/neural-net-gradient-optimization/blob/master/index.html

3 Conclusion

We started the report by explaining the direct connection of neural networks and the optimization algorithms. We explained the importance of deep understanding of those algorithms to be able to explain the results of training and eventually improve those.
Comparison of 3 different types of Gradient Descent was given (Batch GD, Mini Batch GD and SGD). Despite the fact that Mini Batch Gradient Descent is the most popular, it sometimes fails because of the limit of computational resources and an exhaustive amount of data.
Then we discussed the reasons why mathematicians started to consider other approaches for optimization by modifying Stochastic Gradient Descent. The main problem those variations of SGD aimed to

[2]http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

solve was the speed. In this report we considered three different variants of SGD: Momentum, RMSprop, Adam. As the field is quite new, it needs takes time to reconsider those algorithms in different cases and applications to better evaluate their performance and state their use cases by balancing the pros and cons. The report includes thorough explanations of those 3 algorithms to help the reader to get the intuition.
Evaluation results in the previous section showed that *Adam* is generally the choice to go with. It's new and it combines all the known good practices in one powerful algorithm. However, Adam is not well researched and usually we choose the learning rates and hyperparameters simply from practice, without any mathematical base. However, using it with the suggested parameter values will give good results most of the time.
Optimizers are a crucial part of the neural network, understanding how they work would help us to choose which one to use for a specific application. In deep learning tasks, many different optimization algorithms are used sometimes even without thorough analysis. As we could observe in the previous section, the same algorithms can can work excellent in one case and fail in another, which means that a we should consider each case separately and use the appropriate algorithm with the right values of hyperparameters to achieve the desirable results.

4 References

- Ruder, S. (2015). An overview of gradient descent optimization algorithms.Dublin: Insight Centre for Data Analytics, NUI Galway Aylien Ltd. Available at: https://arxiv.org/pdf/1609.04747.pdf [Accessed 20 Feb. 2019].

- Deeplearning.ai (2017). RMSProp (C2W2L07). [video] Available at: https://youtu.be/_e-LFe_igno [Accessed 15 Mar. 2019].

- Deeplearning.ai (2016). Gradient Descent With Momentum (C2W2L06). [video] Available at: https://youtu.be/k8fTYJPd3_I [Accessed 15 Mar. 2019].

- Deeplearning.ai (2017). Adam Optimization Algorithm (C2W2L08). [video] Available at: https://youtu.be/JXQT_vxqwIs [Accessed 15 Mar. 2019].

- Yuret, D., Radford, A. (2015). Alec Radford's animations for optimization algorithms. Retrieved from http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

- Radford, A. (2018). Adam Optimizer. Retrieved from https://gist.github.com/Newmu/acb738767acb4788bac3

- Dupont, E. (2019). Optimization Algorithms Visualization. Retrieved from https://bl.ocks.org/EmilienDupont/aaf429be5705b219aaaf8d691e27ca87

5 Evaluation

Below is the brief evaluation of the process of working on the project.
The main challenge for us was doing everything in *Matlab*. It would be easier to do it in Python as it would also give us access to powerful tools such as *Tensorflow* with help of which we could train real neural networks using our own GD algorithm implementations. *Matlab* was generally counter-intuitive for us (maybe for many other programmers as well) and we lost a lot of our time on syntax/command related problems. The total time spent on project is almost a week (5 * 24 hours approximately) The maths part learning curve was steep at start and then after some research it was pretty clear and straightforward. At start I thought I am not ready to understand this stuff but here we are :) Also, many specialists of the field (e.g. OpenAI developers) helped us along our way. We connected to them via *Slack* and *Twitter*.
As the project was quite big, it required a fair amount of time. One of the biggest problems we encountered was the lack of official information and other resources. The field started to grow and develop not very long ago which is why is is mainly based on blog posts and the open source projects of the Machine Learning/Deep Learning community. The work was divided between the group members. We have been working separately during the week and meeting for further discussions and improvements during the weekends.
We used some material from the online courses of Coursera (by the way, RMSprop algorithm was introduced by Geoff Hinton, an instructor of a Coursera class).
Overall, the process of researching the topic was interesting and the goal of deeper exploring neural networks and the math behind it was met.