American University of Armenia

College of Science and Engineering

Optimization Project

Optimization Algorithms for Neural Networks

Team: Arusyak Hakobayn, Emma Saroyan, Lilia Mamikonyan

Spring, 2018

Abstract

This paper discusses optimization algorithms for neural networks, specifically the Nesterov Accelerated Gradient algorithm, the Adaptive Gradient Descent, the Root Mean Squared Propagation and the Adaptive Moment Estimation algorithms. We first give an introduction to Gradient Descent and Stochastic Gradient Descent in brief to prepare the ground for the four main optimization algorithms discussed in this paper. Then we consider their applications, provide in-depth explanations, examples and Matlab implementations.

Keywords: gradient, gradient descent, stochastic gradient descent, blackbox optimization, momentum, nesterov accelerated gradient, adagrad, adam, adaptive moment estimation, root mean squares propagation, rmsprop, applications of fractals

Contents

Abstract **i**

1 Introduction **4**

2 Theoretical Background **5**

2.1 Blacbox Optimization 5

2.2 Linear Regression 5

2.3 About Gradient Descent 6

2.4 Stochastic Gradient Descent 7

2.5 Problem Setting 8

3 Momentum and Nesterov Accelerated Gradient Descent **11**

3.1 Momentum: Explained 11

3.2 Nesterov Accelerated Gradient Descent: Explained 12

3.3 Nesterov Accelerated Gradient Descent: Problem Implementation 13

3.4 Nesterov Accelerated Gradient Descent: Pros and Cons . . 14

4 Adaptive Gradient Descent **16**

4.1 Adagrad applications 16

4.2 Adagrad: Explained 16

4.3 Adagrad: Pseudo-code and Problem Implementation 17

4.4 Adagrad: Pros and Cons 19

5 Root Mean Squared Propagation **20**

5.1 RMSProp: Explained 20

5.2 RMSProp: Pseudo-code and Problem Implementation 21

6 Adaptive Moment Estimation **23**

6.1 Adaptive Moment Estimation: Explained 23

6.2 Adaptive Moment Estimation: Pseudo Code and Problem Implementation 24
6.3 Adaptive Moment Estimation: Pros and Cons 25

7 The Results and Conclusion 27

8 Appendices 28
8.1 Algorithm Implementation in Matlab 28
8.1.1 Matlab Code for Gradient Descent 28
8.1.2 Matlab Code for Stochastic Gradient Descent 28
8.1.3 Matlab Code for Nesterov Accelerated Gradient Descent 29
8.1.4 Matlab Code for Adaptive Gradient Descent 30
8.1.5 Matlab Code for Root Mean Squared Propagation . . 30
8.1.6 Matlab Code for Adaptive Moment Estimation 31
8.1.7 Matlab Code for Computing Cost 32
8.1.8 Matlab Code for Plotting Data 32
8.1.9 Matlab Code for Runner 32

List of Figures

2.1 Linearly dependent data 6
2.2 SGD fluctuation (Source: Wikipedia) 8
2.3 (a) Scatterplot of data (b) Surface plot of data (c) Training data 10

3.1 (a) SGD without momentum (b) SGD with momentum 11
3.2 Momentum and NAG updates 13
3.3 Contour of NAG 14
3.4 Results of NAG 14
3.5 Convergence rates for GD and NAG 15

4.1 Contour of Adagrad 18
4.2 Results of Adagrad 18

5.1 Contour of RMSProp 22
5.2 Results of RMSProp 22

6.1 Contour of Adam 24
6.2 Results of Adam 25

Chapter 1

Introduction

Optimization algorithms help to minimize (maximize) a given objective function. Some objective functions are defined analytically and have different parameters which belong to a particular set. Others may not be defined analytically but be generated as a result of an experimental or a simulated data. In both cases we need efficient optimization algorithms which converge to a real solution with the minimum error. In this paper we will explore four widely used optimization algorithms for neural networks - Nesterov Accelerated Gradient (NAG), Adaptive Gradient Descent (Adagrad), Root Mean Squared Propagation (RMSProp) and Adaptive Moment Estimation (Adam). All the mentioned algorithms are examples of Stochastic Gradient Descent, however one is more efficient than the other, or one is converging to a local extrema faster than the other. In this paper we will explain how these algorithms work, what are their drawbacks or advantages over one another. After covering the theoretical part, we will implement a regressive task in order to visualize the algorithm performances and how they converge to a local/global solution. All the sources used during the research are provided at the back of the paper.

Chapter 2

Theoretical Background

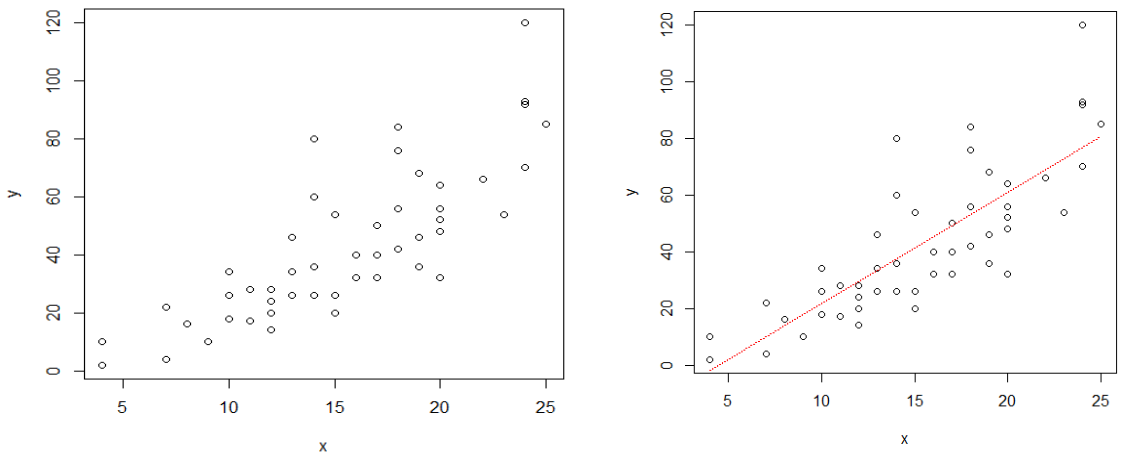
2.1 Blacbox Optimization

Suppose we are dealing with some data, containing variables x and y, which are somehow related to each other. In minimization problems we are usually given the relation pattern of these variables - the objective function $J(\theta)$, and a set Ω where x is defined. However, when the objective function and the feasible set are results of a computer code (a black box), we are dealing with Blackbox optimization, where the problem lacks the algebraic representation of $J(\theta)$ and Ω [ASS16].

Blackbox system is formed when we have simulated or experimental data. Different statistical and machine learning techniques are used to handle these types of data. The optimization methods discussed in the next chapters are representations of Blackbox optimization.

2.2 Linear Regression

Suppose that the variables x and y from a given data are following a linear pattern as in Figure 2.1a. This implies that the relationship between x



(a) (b)

Figure 2.1: Linearly dependent data

and y is linear and has the form

$$y = w \cdot x$$

where w is the slope of the red line in Figure 2.1b. Our problem becomes fitting the red line in the data in a way that it describes the data (i.e. $x - y$ relationship) in the best possible way [Rai16]. Here we define a loss function l, which is the difference between a point on the red line and the corresponding y. Eventually our goal becomes to minimize the loss function $l(y, f(x))$ which is the error of f on an example (x, y)

$$Minimize \quad l(y, f(x)).$$

Gradient Descent method will be used for the minimization of the loss function.

2.3 About Gradient Descent

Gradient descent (GD) is one of many optimization algorithms used to find the solution to a minimization problem when it cannot be obtained analytically. GD algorithm follows the below stated general steps to minimize an objective function $J(\theta)$:

1. Calculate descent direction by taking the opposite of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. θ

2. Set a learning rate η

3. Update parameters according to $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$.

Please refer to the Appendix for Gradient Descent Matlab implementation.[1]

Taking into consideration the amount of data we have, the parameter update accuracy and the time for performing this update we may choose one the following GD types:

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent

In the next chapter we will observe the behavior of the Stochastic gradient descent algorithm, its various implementations and advantages. This will prepare the ground for further discussions about the Nesterov accelerated gradient (NAG), Adagrad, Adadelta and Adam optimization algorithms.

2.4 Stochastic Gradient Descent

While Batch gradient descent (BGD) algorithm computes $\nabla_\theta J(\theta)$ w.r.t. θ for the entire training dataset, Stochastic gradient descent (SGD) updates the parameters one at a time. In other words, SGD changes the descent direction for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Due to frequent updates SGD outperforms BDG in terms of time and potentially better local minimum point. Moreover, by gradually decreasing the learning rate η in convex optimization problems, SGD algorithm almost surely converges to the global minimum point. It is worth to note that high variance updates can cause fluctuations in the objective function $J(\theta)$ as in Figure 2.2.

SGD has a very good performance when it comes to training time and it is successfully applied to many machine learning problems which are

[1]You can find the code of GD in Appendix

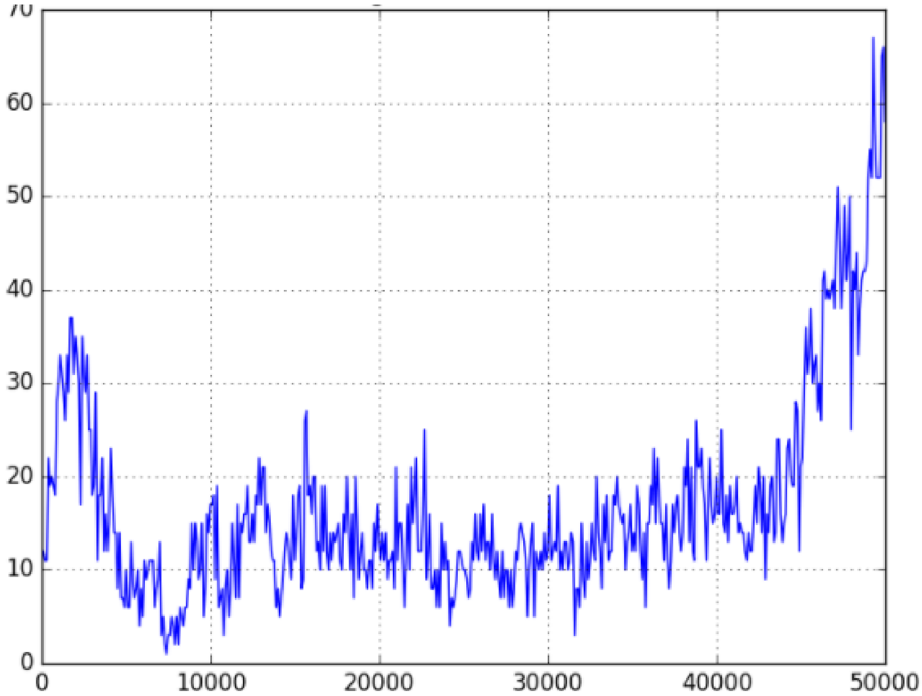


Figure 2.2: SGD fluctuation (Source: Wikipedia)

sparse and large-scale. However, there are drawbacks connected with this algorithm, one of which being the learning rate adaptation. In addition, as we want to decrease the learning rate, we will have to tune hyper parameters which might complicate the convergence [Bot12].

Please refer to the Appendix for Stochastic Gradient Descent Matlab implementation.[2]

There are different algorithms which are used to improve the above mentioned drawbacks of SGD to some extent. In later chapters we will see what exactly those algorithms suggest and what are the expected results.

2.5 Problem Setting

In this project we will implement linear regression (one variable) task. The task will be to predict profits for a food truck. For opening up a new outlet, there are many city choices. However we are interested in the one promising the highest possible profit. Let's say the chain has trucks in different cities and we have available data for profits and populations

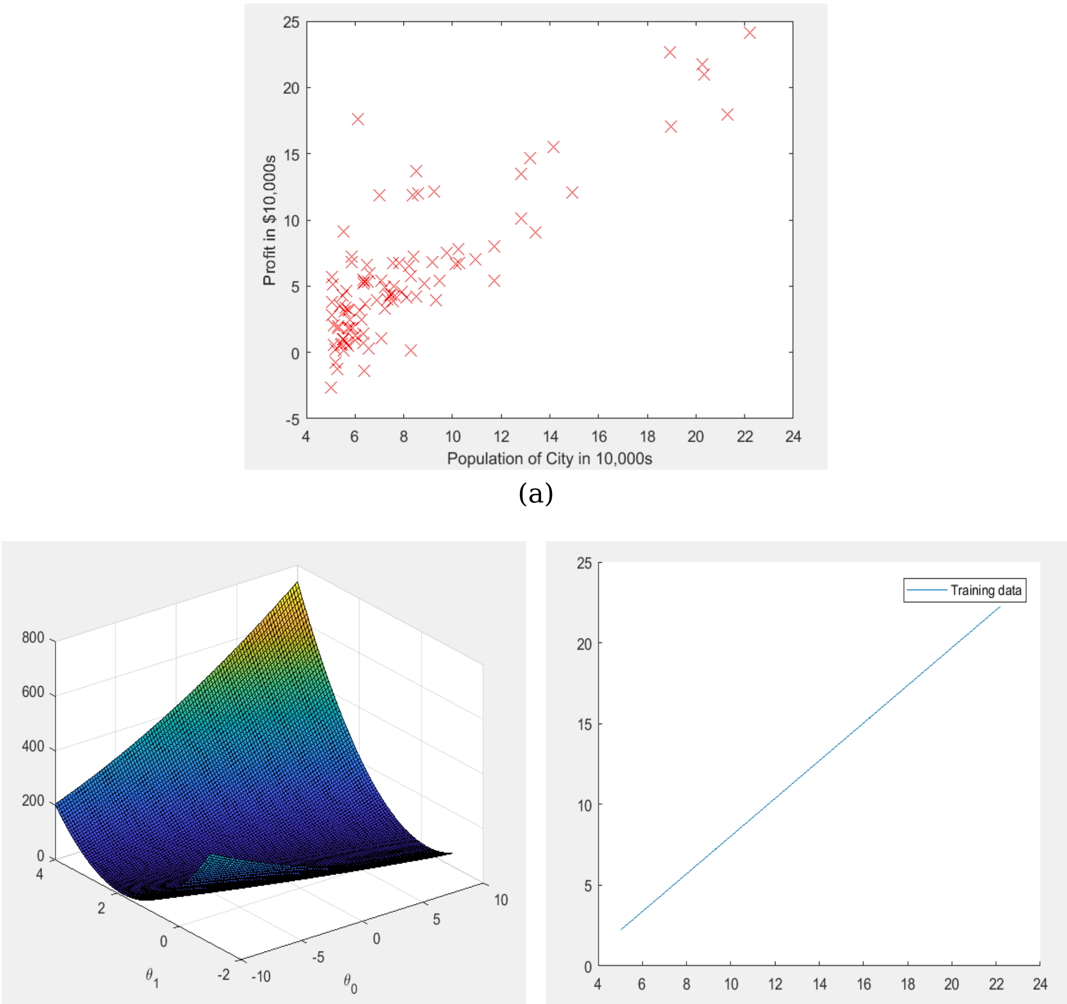
[2]You can find the code of SGD in Appendix

from those cities. Our goal is to make this data useful for selecting a city for later successful expansion. We will be using the optimization algorithms discussed in this paper to implement this task, and we will observe the differences in the outputs and overall performance.

The dataset for our one variable linear regression problem is given in the Appendix.[3] The first column is the population of a city and the second one is the profit of a food truck in that city (negative value for profit means loss).

Follow the plots in Figure 2.3 for a better visualization of the input data. As you can see, the data is indeed in linear form, thus we can use simple Linear Regression model and develop our optimization algorithms accordingly.

[3]You can find the input dataset for Matlab codes in Appendix



(a)

(b)

(c)

Figure 2.3: (a) Scatterplot of data (b) Surface plot of data (c) Training data

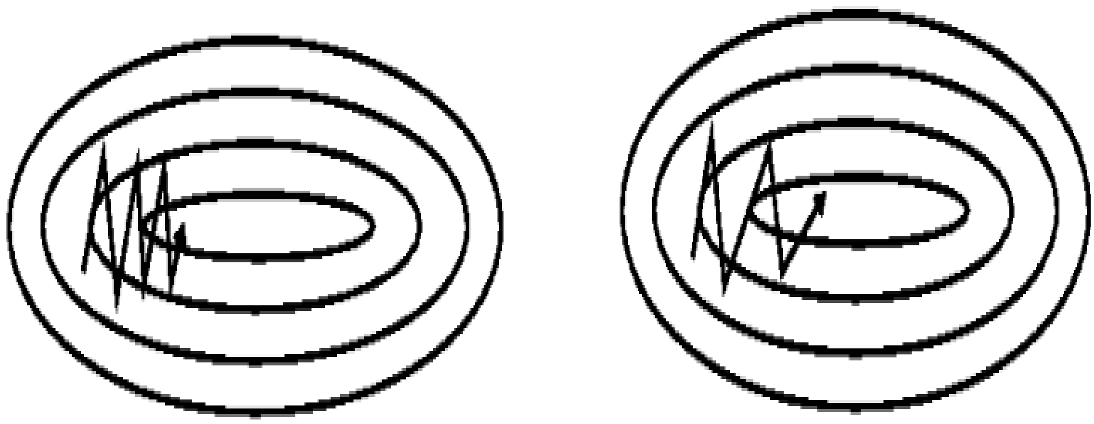
Chapter 3

Momentum and Nesterov Accelerated Gradient Descent

We have already described the Stochastic Gradient Descent algorithm, its implementation, and challenges. There are methods that are designed to help improve the performance of SGD, and in this chapter we will outline two methods that help to accelerate SGD. These methods are Momentum and Nesterov Accelerated Gradient (NAG).

3.1 Momentum: Explained

As can be seen in the figure below, in the second scenario, Momentum helps to accelerate SGD in the relevant direction by damping oscillations. How it does this? It does this by adding a fraction γ of the update vector



(a) (b)

Figure 3.1: (a) SGD without momentum (b) SGD with momentum

of the past time step t to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

The γ above is called a momentum term and is usually set to 0.9 or a close value.

According to a popular story about momentum, Gradient Descent is a man walking down a hill. This man follows the steepest path downwards which means that his progress is really slow though steady and directed to a "minima point". And now, compared to the Gradient Descent, this new Momentum is a heavy ball rolling down the same hill. The added inertia (which in the case of Momentum algorithm is the momentum term) acts both as a smoother and an accelerator, damping oscillations and causing it roll smoothly towards a local minima [Goh17]. This standard story is a common intuition behind the algorithm explained above.

More rigorously, this is what happens to our parameter updates when using Momentum:

1. The momentum term increases for dimensions whose gradients point in the same directions.

2. The momentum term reduces updates for dimensions whose gradients change directions.

As a result, the method of Momentum gives faster convergence and reduced oscillation.

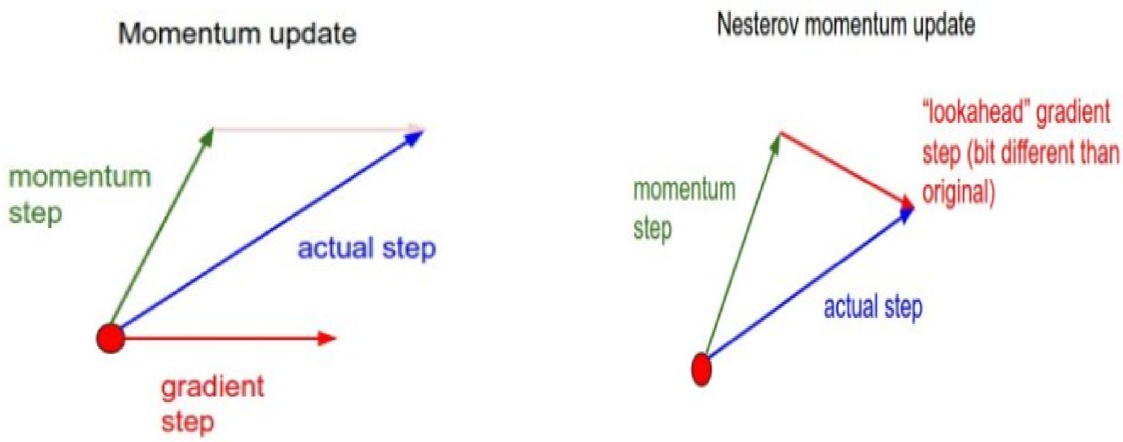
3.2 Nesterov Accelerated Gradient Descent: Explained

As we have seen, the Momentum gives faster convergence and reduced oscillation or, in other words, we have a ball that accelerates and rolls down a hill, following the slope. However, "blindly" following a slope is not that satisfactory because, for some functions that are non-convex, it may fail to converge. This is where the Nesterov Accelerated Gradient Descent (NAG) method comes at play, which solves the problem by finding an algorithm achieving the same acceleration as that of the method of Momentum, but can be shown to converge for some general cases of convex functions.

Nesterov Accelerated Gradient Descent is proven to be the optimal method among all gradient based algorithms. It handles general types of convex functions.

NAG is a way to give a precision to the momentum term. We will use the momentum term γv_{t-1} to move the parameter θ. By computing $\theta - \gamma v_{t-1}$ we obtain an approximation of the next position of the parameters. And the key is that in this algorithm we calculate the gradient to the approximate future position of our parameters as done below:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$



(a) (b)

Figure 3.2: Momentum and NAG updates

3.3 Nesterov Accelerated Gradient Descent: Problem Implementation

In Chapter 1 we discussed the problem of predicting profits for a food truck. We solved this problem using NAG algorithm.[1] In Figure 3.3 you can see the contour of NAG.

[1]You can find the code of NAG in Appendix

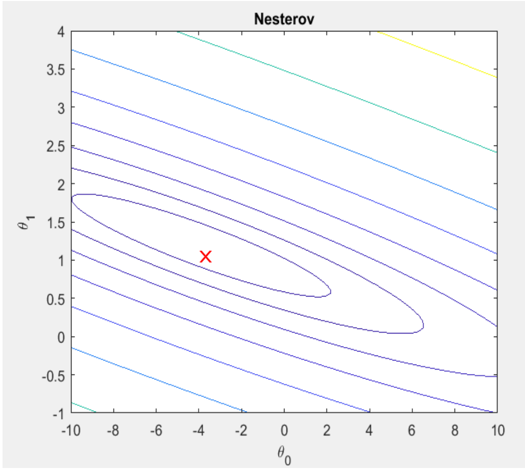


Figure 3.3: Contour of NAG

The results of our solution using NAG algorithm are in Figure 3.4 which is exactly the output we received after running NAG algorithm on Matlab.

```
Theta found by nesterov: -3.674755 1.045245
For population = 35,000, we predict a profit of -163.973648
For population = 70,000, we predict a profit of 36419.602749
```

Figure 3.4: Results of NAG

We can infer that whenever the population is around 35000 people, we are not likely to make profit based on our training data. However, when the population doubles we indeed receive some good profit.

3.4 Nesterov Accelerated Gradient Descent: Pros and Cons

NAG is the optimal gradient based approach for smooth and strongly convex functions NAG enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistently works slightly better than standard momentum.

Class of Function	GD	NAG
Smooth	$O(1/T)$	$O(1/T^2)$
Smooth & Strongly-Convex	$O\left(exp\left(-\frac{T}{\kappa}\right)\right)$	$O\left(exp\left(-\frac{T}{\sqrt{\kappa}}\right)\right)$

Figure 3.5: Convergence rates for GD and NAG

Chapter 4

Adaptive Gradient Descent

The name ”Adaptive Gradient Descent” already claims about its adaptive behavior. Adaptive Gradient Descent (Adagrad) is an optimization algorithm used to improve the strength of SGD. More precisely, it adapts the learning rate η at each step for every parameter θ.

4.1 Adagrad applications

In August 2008 Stefan Klein, Josien P.W. Pluim and Marius Staring published an article at Springerlink.com. The article introduced the Adagrad optimization algorithm for image registration by predicting the adaptive step size η. Their algorithm developed an image driven mechanism to select proper values for the most important free parameters of the method [KPS08].

In 2012 Jeffrey Dean used Adagrad algorithm at Google to train large-scale neural nets which learned to recognize cats in YouTube videos [Rud16]. In addition, Jeffrey Pennington used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones [Rud16].

4.2 Adagrad: Explained

The advantage of Adagrad over other optimization algorithms is that it provides feature-specific adaptive learning rate η by performing larger updates for infrequent features and smaller updates for frequent features, where infrequent features indicate high variance and frequent features indicate low variance. In other words, this algorithm adapts the learning rate to the size of the gradient.

The steps below comprise the algorithm of Adagrad.

1. Calculate the gradient of the cost function J w.r.t. the parameter θ_i at time step t, which will actually be our descent direction (opposite of the gradient). Let's denote this by $g_{t,i}$:

$$g_{t,i} = -\nabla_{\theta_t} J(\theta_{t,i})$$

2. Update the parameter θ_i at each time step t:

$$\theta_{t+1,i} = \theta_{t,i} + \eta \cdot g_{t,i}$$

   Note: η is the general learning rate which will be modified at each time step t. See the next step.

3. Calculate the following constant:

$$G_{t,ii} = \sum_{i=1}^{t} g_i g_i^T = g_1 g_1^T + g_2 g_2^T + ... + g_t g_t^T,$$

   where G_t is a diagonal matrix in which every diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t.

4. Update:

$$\theta_{t+1,i} = \theta_{t,i} + \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i},$$

   where ϵ is just a very small (usually, something like 10^{-8}) addition in the denominator to avoid division by zero.

The general learning rate η most of the time is kept constant 0.01.

4.3 Adagrad: Pseudo-code and Problem Implementation

See below the pseudo-code for Adagrad optimization algorithm.

Algorithm 1 Adagrad

procedure MyProcedure
    eta $\leftarrow$ 0.01
    epsilon $\leftarrow$ 0.000001
    diagMatrixG $\leftarrow$ 0
    theta $\leftarrow$ *randn*
while not converged:
    grad $\leftarrow$ *computeGrad(costFunction)*
    diagMatrixG $\leftarrow$ $grad^2$
    adjustedGrad $\leftarrow$ *grad* $\div (epsilon + sqrt(diagMatrixG))$
    theta $\leftarrow$ *theta* $-$ *eta* $\times$ *adjustedGrad*

In Figure 4.1 you can see the contour of Adagrad.

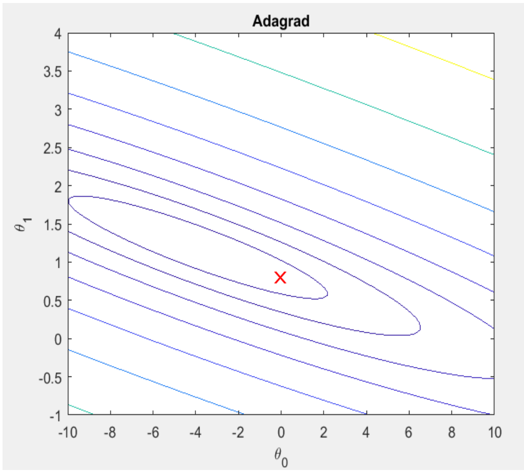


Figure 4.1: Contour of Adagrad

Now we are solving the same problem stated in Chapter 1, this time using Adagrad algorithm.[1]

The results of our solution using Adagrad algorithm are in Figure 4.2 which is exactly the output we received after running Adagrad algorithm on Matlab. As compared to NAG algorithm, we receive much better re-

```
Theta found by adagrad: -0.026181 0.804007
For population = 35,000, we predict a profit of 27878.443318
For population = 70,000, we predict a profit of 56018.698542
```

Figure 4.2: Results of Adagrad

sults using Adagrad. Why? Because Adagrad, is developed to adapt the

[1]You can find the code of Adagrad in Appendix

learning rate with each parameter, to perform necessary frequent and non-frequent updates, thus resulting in better output.

4.4 Adagrad: Pros and Cons

The advantage of Adagrad over other optimization algorithms is its adaptive learning rate. The latter continuously changes with each parameter. Those parameters which appear rarely receive higher learning rate, while those which are frequent are updated with lower learning rate. These updates result in accelerating the convergence of the problem.

The disadvantage of Adagrad is that the learning rate diminishes very fast. This is a result of having the sum of the squares of the gradients w.r.t. θ_i up to the time point t in the denominator. As denominator increases, the fraction becomes smaller and smaller and eventually decreases the overall learning rate monotonically [DHS11].

Chapter 5

Root Mean Squared Propagation

In the previous chapter we introduced Adagrad optimization algorithm where we learned that because of the accumulation of the sum of the gradients in the denominator the learning rate with time becomes infinitely small number. At this point the algorithm becomes no longer useful. In order to avoid such accumulation we will introduce a new algorithm called Root Mean Squared Prop (RMSProp) which will resolve this issue by restricting the growing sum to some fixed size w.

The name ”Root Mean Square” already claims about taking the square of an average under some root. More precisely, instead of taking the square root of the sum of the gradients (like in Adagrad), we take the square root of the mean of the recent gradients. This might seem a bit confusing, but in reality it helps to prevent the continuously increasing sum in the denominator.

5.1 RMSProp: Explained

Generally, RMSProp is one of the most used methods in Neural Networks. RMSProp uses a similar idea as Adagrad, with small betterment, while adjusting the learning rate for each feature. This algorithm was proposed by Geoff HintonIn stemming from the need to resolve Adagrad's radically diminishing learning rates [Rud16]. Around the same time, independently of each other, another algorithm called AdaDelta was developed which is very similar to RMSProp.

RMSProp is an optimizer which utilizes the sum of the gradients by taking their averages, and using this as part of the learning rate. Let $\nabla_{\theta_t} J(\theta_t)$ be the gradient of the cost function w.r.t. the parameter θ at time step t. Denote this by g_t

$$g_t = -\nabla_{\theta_t} J(\theta_t).$$

Afterwards, let's denote the running average of the recent gradients at time step t by $E[g^2]_t$, which obviously depends on the previous average $E[g^2]_{t-1}$ and the current gradient (squared) g_t^2. We also use a decay term γ to perform the update, where γ is usually set to 0.9:

$$E[g^2]_t = \gamma \cdot E[g^2]_{t-1} + (1-\gamma) \cdot g_t^2$$

Therefore, the update of the parameter θ at time $t+1$ becomes:

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

A good default value for η in this case is 0.001 and ϵ in the denominator again is a term to avoid division by 0. Therefore, the final update will look like

$$\theta_{t+1} = \theta_t + \frac{0.001}{\sqrt{0.9 \cdot E[g^2]_{t-1} + 0.1 \cdot g_t^2 + \epsilon}} \cdot g_t$$

5.2 RMSProp: Pseudo-code and Problem Implementation

See below the pseudo-code for RMSProp optimization algorithm.

Algorithm 2 Adagrad

```
procedure MyProcedure
    eta ← 0.001
    gamma ← 0.9
    epsilon ← 0.000001
    avgGrad ← 0
    theta ← randn
while not converged:
    grad ← computeGrad(costFunction)
    avgGrad ← gamma × avgGrad + (1 − gamma) × grad
    adjustedGrad ← grad ÷(epsilon + sqrt(avgGrad))
    theta ← theta − eta × adjustedGrad
```

In Figure 5.1 you can see the contour of RMSProp.

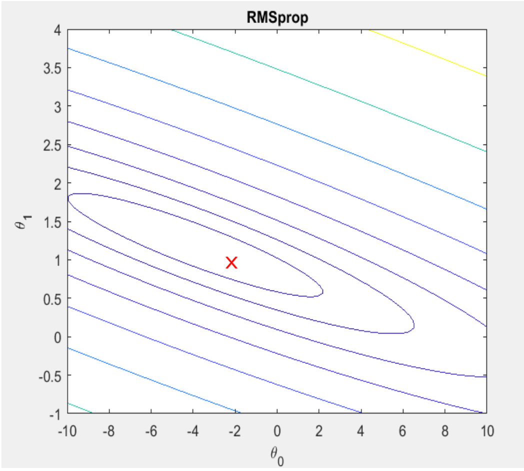


Figure 5.1: Contour of RMSProp

In this section we will again solve the same problem stated in Chapter 1, this time using RMSProp algorithm.[1]

The results of our solution using RMSProp algorithm are in Figure 5.2 which is exactly the output we received after running RMSProp algorithm on Matlab. As we can see, for a city with population 35000 people

```
Theta found by RMSprop: -2.187097 0.958108
For population = 35,000, we predict a profit of 11662.814601
For population = 70,000, we predict a profit of 45196.602274
```

Figure 5.2: Results of RMSProp

we predict a profit of about 12000, and for a city with population 70000 people profit is around 45000.

[1] You can find the code of RMSProp in Appendix

Chapter 6

Adaptive Moment Estimation

We described two optimization algorithms before - Adagrad and RMSProp. We learned that Adagrad was working well with sparse gradients, and RMSProp was working well in on-line and non-stationary settings. Now we will discuss a new algorithm called Adaptive Moment Estimation (Adam) which basically combines the advantages of these two methods and results in much better optimization. As we can infer from its name, Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

6.1 Adaptive Moment Estimation: Explained

Similar to Momentum, Adam keeps an exponentially decaying average of the past gradients. Let's denote the estimates of the first moment and the second moment of the gradients by m_t and v_t respectively, and let β_1 and β_2 be the decay rates. Then, m_t and v_t will be updated in the following way:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

As m_t and v_t are initially vectors of 0's, according to Adam observations, they become biased towards 0, especially during the initial time steps and in case β_1 and β_2 are small [Rud16]. For this reason new first and second moment estimates are formed to correct the bias:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \text{ (estimate of the first moment of the gradient)}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \text{ (estimate of the second moment of the gradient)}$$

Afterwards, we update the paramether θ using similar update rule as used during RMSProp:

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

During the update β_1 is usually set to 0.9, β_2 is 0.999 and ϵ is 0.00000001.

6.2 Adaptive Moment Estimation: Pseudo Code and Problem Implementation

See below the pseudo-code for Adam optimization algorithm.

Algorithm 3 Adam

procedure MyProcedure
  $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  $t \leftarrow 0$
while θ_t not converged do:
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta J_t(\theta_{t-1})$
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
  $\hat{m}_t \leftarrow m_t \div (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t \div (1 - \beta_2^t)$
  $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \hat{m}_t \div (\sqrt{\hat{v}_t} + \epsilon)$

In Figure 6.1 you can see the contour of Adam.

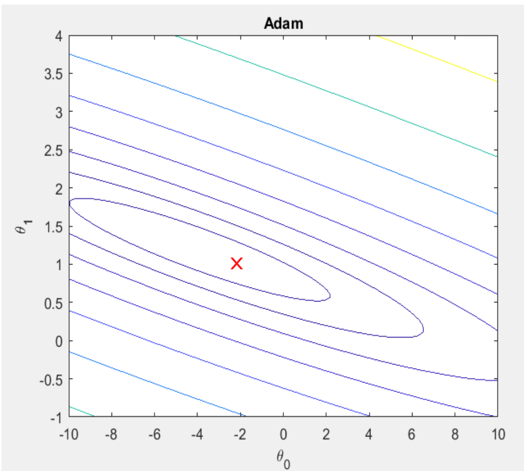


Figure 6.1: Contour of Adam

Now we will solve the same problem stated in Chapter 1 for the last time, using the last algorithm discussed in this paper - Adam algorithm.[1]

The results of our solution using Adam algorithm are in Figure 6.2 which is exactly the output we received after running Adam algorithm on Matlab. As we can see, for a city with population 35000 people we

```
Theta found by Adam: -2.181565 1.010481
For population = 35,000, we predict a profit of 13551.200704
For population = 70,000, we predict a profit of 48918.052905
```

Figure 6.2: Results of Adam

predict a profit of about 14000, and for a city with population 70000 people profit is around 49000.

6.3 Adaptive Moment Estimation: Pros and Cons

Some of the advantages of Adam include:

- Magnitudes of parameter updates are invariant to rescaling of the gradient,
- Step sizes are approximately bounded by the step size hyper parameter
- Does not require a stationary objective
- Works with sparse gradients
- Naturally performs a form of step size annealing

At this point it is good to emphasize some differences between the three algorithms discussed above. RMSProp with momentum generates its parameter updates using a momentum on the rescaled gradient, whereas Adam updates are directly estimated using a running average of first and second moment of the gradient. RMSProp also lacks a bias-correction term; this matters most in case of a value of β_2 close to 1, since in that

[1]You can find the code of Adam in Appendix

case not correcting the bias leads to very large step sizes and often divergence [KB15].

Chapter 7

The Results and Conclusion

The main focus of the paper - Optimization Algorithms for Neural Networks - was to introduce four widely used optimization algorithms for Neural Networks. More precisely, we talked about Nesterov Accelerated Gradient(NAG), Adaptive Gradient(Adagrad), Root Mean Squared Propagation(RMSPROP) and Adaptive Moment Estimation(Adam). Moving on, we provided detailed explanation on how the algorithms work, and how their performances differ from one another. Then we developed a regressive example and used it for our algorithms to see what outputs each one of them would produce. We also observed some of the modern applications of these algorithms. As a result, we have developed Matlab implementations of the above mentioned algorithms which can later be used in various datasets as inputs.

Chapter 8

Appendices

8.1 Algorithm Implementation in Matlab

8.1.1 Matlab Code for Gradient Descent

```
function [theta, J_history] = gradientDescent(x, y, theta,
   alpha, num_iters)

% Initialize some useful values
m = length(y); % number of training examples
X = [ones(m,1) x]; % Add a column of ones to x
% J_history = zeros(num_iters, 1);

for i = 1:num_iters
    hypothesis = X * theta;
    loss = hypothesis - y;
    gradient = (transpose(X) * loss) / m;
    temp = theta - alpha * gradient;
    theta = [temp(1); temp(2)];

    % Save the cost J in every iteration
%     J_history(i) = computeCost(X, y, theta);

end

end
```

8.1.2 Matlab Code for Stochastic Gradient Descent

```
function [theta] = stochasticGradientDescent(x, y, theta,
   alpha, num_iters)

%Initialize some useful values
m = length(y); % number of training examples
```

```
n = size(x,2); % number of features
X = [ ones(m,1) x]; % Add a column of ones to x
data = [X, y];

for i = 1:num_iters
%     data = data(randperm(size(data,2)),:); % data can be shuffled;
    for j = 1 : m
        % j - example number
        hypothesis = (data(j,1:2)*theta)*ones(1,n);
        loss = data(j,1:2);
        y_new = data(j,3)*ones(1,n);
        gradient = 1/m * ((hypothesis - y_new).*loss).';
        theta = theta - alpha * gradient;
    end

end

end
```

8.1.3 Matlab Code for Nesterov Accelerated Gradient Descent

```
function [theta] = nesterov_m(x, y, theta, velocity , alpha, gamma, num_iters)

%Initialize some useful values
m = length(y); % number of training examples
n = size(x,2); % number of features
X = [ ones(m,1) x]; % Add a column of ones to x
data = [X, y];

for i = 1:num_iters
%     data = data(randperm(size(data,2)),:); % data can be shuffled
    for j = 1 : m
        % j - example number
        theta = theta - gamma * velocity;
        hypothesis = (data(j,1:2)*theta)*ones(1,n);
        loss = data(j,1:2);
        y_new = data(j,3)*ones(1,n);
        gradient = 1/m * ((hypothesis - y_new).*loss).';
        velocity = gamma * velocity + alpha * gradient;
        theta = theta - velocity;
    end

```

```
end

end
```

8.1.4 Matlab Code for Adaptive Gradient Descent

```
function [theta] = adagrad(x, y, theta, grad_sum_square , alpha,epsilon, num_iters)

%Initialize some useful values
m = length(y); % number of training examples
n = size(x,2); % number of features
X = [ ones(m,1) x]; % Add a column of ones to x
data = [X, y];

for i = 1:num_iters
%     data = data(randperm(size(data,2)),:); % data can be shuffled
    for j = 1 : m
        % j - example number
        hypothesis = (data(j,1:2)*theta)*ones(1,n);
        loss = data(j,1:2);
        y_new = data(j,3)*ones(1,n);
        gradient = 1/m * ((hypothesis - y_new).*loss).';
        grad_sum_square = grad_sum_square + dot(gradient, gradient);
        delta = - alpha * gradient / sqrt(grad_sum_square + epsilon);
        theta = theta + delta;
    end

end

end
```

8.1.5 Matlab Code for Root Mean Squared Propagation

```
function [theta] = adagrad(x, y, theta, grad_sum_square , alpha,epsilon, num_iters)

%Initialize some useful values
m = length(y); % number of training examples
n = size(x,2); % number of features
X = [ ones(m,1) x]; % Add a column of ones to x
data = [X, y];

```

```
for i = 1:num_iters
%     data = data(randperm(size(data,2)),:); % data can be
    shuffled
    for j = 1 : m
        % j - example number
        hypothesis = (data(j,1:2)*theta)*ones(1,n);
        loss = data(j,1:2);
        y_new = data(j,3)*ones(1,n);
        gradient = 1/m * ((hypothesis - y_new).*loss).';
        grad_sum_square = grad_sum_square + dot(gradient,
            gradient);
        delta = - alpha * gradient / sqrt(grad_sum_square +
            epsilon);
        theta = theta + delta;
    end

end

end
```

8.1.6 Matlab Code for Adaptive Moment Estimation

```
function [theta] = adam(x, y, theta, mom_m, mom_v, alpha,
    epsilon, beta1, beta2, beta1_exp, beta2_exp, num_iters)

%Initialize some useful values
m = length(y); % number of training examples
n = size(x,2); % number of features
X = [ ones(m,1) x]; % Add a column of ones to x
data = [X, y];

for i = 1:num_iters
%     data = data(randperm(size(data,2)),:); % data can be
    shuffled
    for j = 1 : m
        % j - example number
        hypothesis = (data(j,1:2)*theta)*ones(1,n);
        loss = data(j,1:2);
        y_new = data(j,3)*ones(1,n);
        gradient = 1/m * ((hypothesis - y_new).*loss).';

        mom_m = beta1 * mom_m + (1.0 - beta1) * gradient;
        mom_v = beta2 * mom_v + (1.0 - beta2) * dot(gradient
            ,gradient);
        beta1_exp = beta1_exp * beta1;
        beta2_exp = beta2_exp * beta2;
        theta = theta - alpha * (mom_m / (1.0 - beta1_exp))
```

```
            / (sqrt(mom_v / (1.0 - beta2_exp)) + epsilon);


    end

end

end
```

8.1.7 Matlab Code for Computing Cost

```
function J = computeCost(X, y, theta)

% Initialize some useful values
m = length(y); % number of training examples

J = (1 / (2*m) ) * sum(((X * theta)-y).^2);

end
```

8.1.8 Matlab Code for Plotting Data

```
function plotData(x, y)

figure; % open a new figure window
plot(x, y, 'rx', 'MarkerSize', 10);
axis([4 24 -5 25]);
xlabel("Population of City in 10,000s"); % setting the x label as population
ylabel("Profit in $10,000s");            % setting the y label

end
```

8.1.9 Matlab Code for Runner

```
%% Initialization
clear ; close all; clc

%% ======================= Part 1: Data Info & Import =======================
% Info
% data is taken for population and profit problem
% x refers to the population size in 10,000s
% y refers to the profit in $10,000s

data = load('ex_data.txt');
```

```
x = data(:, 1);
y = data(:, 2);

%% ======================= Part 2: Plotting =======================
fprintf('Plotting Data ...\n')

% Plot Data
plotData(x, y);

fprintf('Program paused. Press enter to continue.\n');
pause;

%% =================== Part 3.1: Gradient descent ===================
fprintf('Running Gradient Descent ...\n')

theta_gradient = zeros(2, 1); % initialize fitting parameters

% Some gradient descent settings
iterations = 1500;
alpha = 0.01;

% compute and display initial cost
m = length(y);
X = [ones(m, 1), data(:,1)];% Add a column of ones to x
computeCost(X, y, theta_gradient)

% run gradient descent
theta_gradient = gradientDescent(x, y, theta_gradient, alpha, iterations);

% print theta to screen
fprintf('Theta found by gradient descent: ');
fprintf('%f %f \n', theta_gradient(1), theta_gradient(2));

% Plot the linear fit
hold on; % keep previous plot visible
plot(X(:,2), X*theta_gradient, '-')
legend('Training data', 'Linear regression')
hold off % don't overlay any more plots on this figure

% Predict values for population sizes of 35,000 and 70,000
predict1 = [1, 3.5] *theta_gradient;
fprintf('For population = 35,000, we predict a profit of %f\
```

```
n',...
    predict1*10000);
predict2 = [1, 7] * theta_gradient;
fprintf('For population = 70,000, we predict a profit of %f\n',...
    predict2*10000);

fprintf('Program paused. Press enter to continue.\n');
pause;

%% ============ Part 3.2: Stochastic gradient descent ==============
fprintf('Running Stochastic Gradient Descent ...\n')

theta_stochastic = zeros(2, 1); % initialize fitting parameters

% Some stochastic gradient descent settings
iterations = 1500;
alpha = 0.01;

% compute and display initial cost
m = length(y);
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
computeCost(X, y, theta_stochastic)

% run stochastic gradient descent
theta_stochastic = stochasticGradientDescent(x, y, theta_stochastic, alpha, iterations);

% print theta to screen
fprintf('Theta found by stochastic gradient descent: ');
fprintf('%f %f \n', theta_stochastic(1), theta_stochastic(2));

% Plot the linear fit
hold on; % keep previous plot visible
plot(X(:,2), X*theta_stochastic, '-')
legend('Training data', 'Linear regression')
hold off % don't overlay any more plots on this figure

% Predict values for population sizes of 35,000 and 70,000
predict1 = [1, 3.5] *theta_stochastic;
fprintf('For population = 35,000, we predict a profit of %f\n',...
    predict1*10000);
```

```
predict2 = [1, 7] * theta_stochastic;
fprintf('For population = 70,000, we predict a profit of %f\n',...
    predict2*10000);

fprintf('Program paused. Press enter to continue.\n');
pause;
%% ================ Part 3.3: Nasterov ======================

fprintf('Running Nasterov ...\n')

theta_nesterov = zeros(2, 1); % initialize fitting parameters

% Some nesterov settings
iterations = 1500;
alpha = 0.01;
velocity = zeros(2, 1);
m = length(y);
gamma = 0.9;

% compute and display initial cost
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
computeCost(X, y, theta_nesterov)

% run nesterov gradient descent
theta_nesterov = nesterov(x, y, theta_nesterov, velocity , alpha, gamma, iterations);

% print theta to screen
fprintf('Theta found by nesterov: ');
fprintf('%f %f \n', theta_nesterov(1), theta_nesterov(2));

% Predict values for population sizes of 35,000 and 70,000
predict1 = [1, 3.5] *theta_nesterov;
fprintf('For population = 35,000, we predict a profit of %f\n',...
    predict1*10000);
predict2 = [1, 7] * theta_nesterov;
fprintf('For population = 70,000, we predict a profit of %f\n',...
    predict2*10000);

fprintf('Program paused. Press enter to continue.\n');
pause;
```

```
%% ================ Part 3.4: Adagrad ======================

fprintf('Running Adagrad ...\n')

theta_adagrad = zeros(2, 1); % initialize fitting parameters

% Some stochastic gradient descent settings
iterations = 1500;
alpha = 0.01;

% compute and display initial cost
m = length(y);
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
computeCost(X, y, theta_adagrad)
grad_sum_square = 0;
epsilon = 0.0000001;

% run stochastic gradient descent
theta_adagrad = adagrad(x, y, theta_adagrad, grad_sum_square
    , alpha, epsilon, iterations);

% print theta to screen
fprintf('Theta found by adagrad: ');
fprintf('%f %f \n', theta_adagrad(1), theta_adagrad(2));

% Predict values for population sizes of 35,000 and 70,000
predict1 = [1, 3.5] *theta_adagrad;
fprintf('For population = 35,000, we predict a profit of %f\n',...
    predict1*10000);
predict2 = [1, 7] * theta_adagrad;
fprintf('For population = 70,000, we predict a profit of %f\n',...
    predict2*10000);

fprintf('Program paused. Press enter to continue.\n');
pause;

%% ================ Part 3.5: RMSprop ======================
fprintf('Running RMS_prop ...\n')

theta_RMS_prop = zeros(2, 1); % initialize fitting parameters

% Some RMS_prop settings
iterations = 1500;
```

```
alpha = 0.001;

% compute and display initial cost
m = length(y);
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
computeCost(X, y, theta_RMS_prop);
gamma = 0.9;
grad_expect = 0;
epsilon = 0.00000001;
% run stochastic gradient descent
theta_RMS_prop = rms_prop(x, y, theta_RMS_prop, grad_expect,
    alpha, gamma, epsilon, iterations);

% print theta to screen
fprintf('Theta found by RMSprop: ');
fprintf('%f %f \n', theta_RMS_prop(1), theta_RMS_prop(2));

% Predict values for population sizes of 35,000 and 70,000
predict1 = [1, 3.5] *theta_RMS_prop;
fprintf('For population = 35,000, we predict a profit of %f\
  n',...
    predict1*10000);
predict2 = [1, 7] * theta_RMS_prop;
fprintf('For population = 70,000, we predict a profit of %f\
  n',...
    predict2*10000);

fprintf('Program paused. Press enter to continue.\n');
pause;

%% ============== Part 3.6: Adam ====================
fprintf('Running RMS_prop ...\n')

theta_adam = zeros(2, 1); % initialize fitting parameters

% Some Adam settings
iterations = 1500;
alpha = 0.001;

% compute and display initial cost
m = length(y);
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
computeCost(X, y, theta_adam);
mom_m = 0;
mom_v = 0;
epsilon = 0.00000001;
```

```
beta1 = 0.9;
beta2 = 0.999;
beta1_exp = 1.0;
beta2_exp = 1.0;
% run Adam
theta_adam = adam(x, y, theta_adam, mom_m, mom_v, alpha,
    epsilon, beta1, beta2, beta1_exp, beta2_exp, iterations);

% print theta to screen
fprintf('Theta found by Adam: ');
fprintf('%f %f \n', theta_adam(1), theta_adam(2));

% Predict values for population sizes of 35,000 and 70,000
predict1 = [1, 3.5] *theta_adam;
fprintf('For population = 35,000, we predict a profit of %f\
    n',...
    predict1*10000);
predict2 = [1, 7] * theta_adam;
fprintf('For population = 70,000, we predict a profit of %f\
    n',...
    predict2*10000);

fprintf('Program paused. Press enter to continue.\n');
pause;

%% ============= Part 4: Preperation for Visualizing J(
    theta_0, theta_1) =============

% Grid over which we will calculate J
theta0_vals = linspace(-10, 10, 100);
theta1_vals = linspace(-1, 4, 100);

% initialize J_vals to a matrix of 0's
J_vals = zeros(length(theta0_vals), length(theta1_vals));

% Fill out J_vals
for i = 1:length(theta0_vals)
    for j = 1:length(theta1_vals)
        t = [theta0_vals(i); theta1_vals(j)];
        J_vals(i,j) = computeCost(X, y, t);
    end
end

% Because of the way meshgrids work in the surf command, we
    need to
% transpose J_vals before calling surf, or else the axes
```

```
    will be flipped
J_vals = J_vals';

%% Surface plot
fprintf('Visualizing J(theta_0, theta_1) ...\n')

figure;
surf(theta0_vals, theta1_vals, J_vals)
xlabel('\theta_0'); ylabel('\theta_1');

%% Contour plot
fprintf('Visualizing J(theta_0, theta_1) ...\n')

figure;
% Plot J_vals as 15 contours spaced logarithmically between
    0.01 and 100
contour(theta0_vals, theta1_vals, J_vals, logspace(-2, 3,
    20))
xlabel('\theta_0'); ylabel('\theta_1');
hold on;

% plot(theta_gradient(1), theta_gradient(2), 'rx', '
    MarkerSize', 10, 'LineWidth', 2);
% title('Gradient Descent');

% plot(theta_stochastic(1), theta_stochastic(2), 'rx', '
    MarkerSize', 10, 'LineWidth', 2);
% title('Stochastic Gradient Descent');

% plot(theta_nesterov(1), theta_nesterov(2), 'rx', '
    MarkerSize', 10, 'LineWidth', 2);
% title('Nesterov');

% plot(theta_adagrad(1), theta_adagrad(2), 'rx', 'MarkerSize
    ', 10, 'LineWidth', 2);
% title('Adagrad');

% plot(theta_RMS_prop(1), theta_RMS_prop(2), 'rx', '
    MarkerSize', 10, 'LineWidth', 2);
% title('RMSprop');

% plot(theta_adam(1), theta_adam(2), 'rx', 'MarkerSize', 10,
    'LineWidth', 2);
% title('Adam');

x = [theta_gradient(1) theta_stochastic(1) theta_nesterov(1)
```

```
    theta_adagrad(1) theta_RMS_prop(1) theta_adam(1)];
y = [theta_gradient(2) theta_stochastic(2) theta_nesterov(2)
    theta_adagrad(2) theta_RMS_prop(2) theta_adam(2)];
colors = ['r', 'g','r', 'g','r', 'g'];
for i=1:6
    scatter(x(i), y(i),'LineWidth',10,'MarkerFaceColor',
       colors(i));
end
title('All methods together');
% GD - dark_blue, SGD - red, nasterov - orange,
% adagrad - pink, rmsprop - green, adam - light_blue
```

Bibliography

[YLM98] Genevieve Orr Yann LeCun Leon Bottou and Klaus-Robert Müller. *Efficient BackProp*. Springer, 1998.

[KPS08] Stefan Klein, Josien P.W. Pluim, and Marius Staring. *Adaptive Stochastic Gradient Descent Optimization for Image Registration*. Springer, 2008.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*. Journal of Machine Learning Research, 2011.

[Bot12] Léon Bottou. *Stochastic Gradient Descent Tricks*. Springer, 2012.

[KB15] Diederik P. Kingma and Jimmy Lei Ba. *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. 2015.

[ASS16] Satyajith Amaran, Nikolaos V. Sahinidis, and Bikram Sharda. *Simulation optimization: a review of algorithms and applications*. Annals of Operations Research, 2016.

[Fer16] Arturo Fernandez. *AdaGrad: Automatic Learning Rates*. 2016.

[Rai16] Piyush Rai. *Learning as Optimization: Linear Regression*. 2016.

[Rud16] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. arXiv preprint arXiv:1609.04747, 2016.

[Goh17] Gabriel Goh. *Why Momentum Really Works*. 2017.