华南理工大学

South China University of Technology

The Experiment Report of Deep Learning

School: School of Software Engineering

Subject: Software Engineering

Author:

Aiqing Wen

Supervisor:

Mingkui Tan

Student ID:

201720145037

Grade:

Graduate

December 14, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract—**Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent. This report aims to introduce the basics of SGD, NAG, Adadelta, RMSprop and Adam. The experiments part will show the result of logistic regression and linear classification using different optimization algorithms.**

i. INTRODUCTION

Gradient descent optimization algorithms, while increasingly popular, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent. These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by [1].

This experiment aims provide the intuitions towards the behavior of different algorithms for optimizing gradient descent. The first part comes to the introduction of the most common optimization algorithms, content includes the motivation to resolve challenges and how those algorithms lead to the derivation of their update rules. To demonstrate the strengths and weaknesses of different optimization algorithm, a variety of experimental results and analysis that compares those five optimization algorithms with each other will be shown in experiments part. Finally, we will give a conclusion to summarize this experiment.

ii. METHODS AND THEORY

A. Stochastic Gradient Descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example x^i and label y^i:

$$\theta = \theta - \nabla_\theta J(\theta; x^i; y^i)$$

Batch gradient descent performs redundant computations for large datasets, as it recomputed gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Fig. 1.

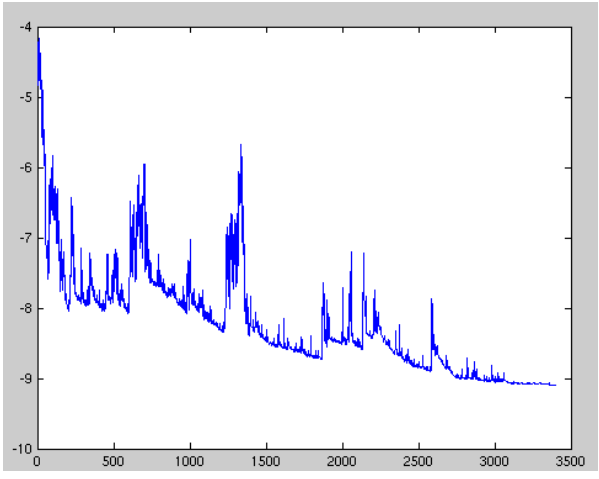


Fig.1. SGD fluctuation

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behavior as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively. Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example.

B. Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another [2], 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 as in Fig. 2.

Fig.2. SGD without momentum

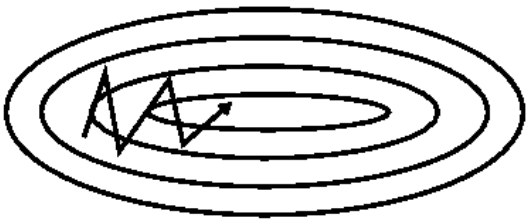


Fig.3. SGD with momentum

Momentum [3] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Fig. 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

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

$$\theta = \theta - v_t$$

the momentum term γ is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

C. Nesterov Accelerated Gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Nesterov accelerated gradient (NAG) [4] is a way to give our momentum term this kind of prescience. We know that we will use our momentum term γv_{t-1} to move the parameters θ. Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

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

$$\theta = \theta - v_t$$

Again, we set the momentum term γ to a value of around 0.9. While Momentum first computes the current gradient (small blue vector in Fig. 4) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector), NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks [5].

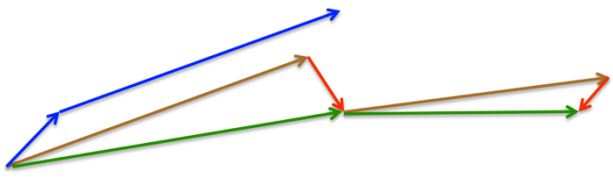


Fig.4. Nesterov update

D. Adadelta

Adadelta [6] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

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

We set γ to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\Delta\theta_t$:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t$$

We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[G]_t} g_t$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \varepsilon}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

E. RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

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

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

F. Adam

Adaptive Moment Estimation (Adam) [7] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients s_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients v_t, similar to momentum:

$$v_t = \beta_1 v_{t-1} + (1-\beta_1) g_t$$

$$s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2$$

v_t and s_t are estimates of the first moment (the mean) and the second moment (the decentered variance) of the gradients respectively, hence the name of the method. As v_t and s_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t}$$

$$\hat{s}_t = \frac{s_t}{1-\beta_2^t}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - - \frac{\eta}{\sqrt{\hat{s}_t} + \varepsilon}\hat{v}_t$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

iii. EXPERIMENTS

A. Dataset

Experiment uses a9a of LIBSVM Data, including 32561/16281(valuation) samples and each sample has 123/123 (valuation) features.

B. Implementation

a) Hyper-parameters

Hyper-parameters and their chosen values:

- Epoch = 5, means that we used the whole training data set 5 times to perform the gradient decent. The training data is shuffled at beginning of each epoch.
- Batch size = 256, 256 samples are used in each iteration.
- Learning rate = 0.05
- Regularization parameter = 0.01
- Threshold = 0.5
- Epsilon = 1e-8
- NAG gamma = 0.9, as the value 0.9 reach the best result compares with 0.8 and 0.95.
- Adadelta gamma = 0.95, as the value 0.95 reach the best result compares with 0.8 and 0.9.
- RMSprop gamma = 0.8, as the value 0.8 reach the best result compares with 0.9 and 0.95.
- Adam beta1 = 0.9, as the value 0.9 reach the best result compares with 0.95. Adam beta2 = 0.99, as the value 0.99 is always chose.

b) Logistic regression

The sigmoid function is used in this experiment as score function while L2 norm is chosen to be the loss function.

c) Linear Classification

The SVM is used to implement this Linear classifier.

C. Results of the algorithms on logistic regression

The loss of different optimization algorithms on the valuation set is shown in Fig. 5.

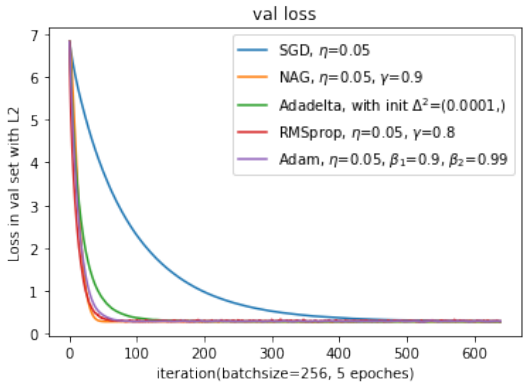
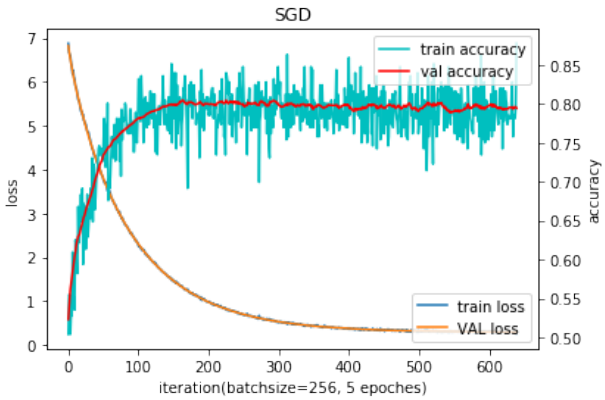


Fig.5. valuation loss (logistic regression)

Fig. 6-10 show the result of SGD, NAG, Adadelta, RMSprop and Adam, which include the loss of the training set and valuation set, and the accuracy of the training set and valuation set.



F Fig.6. SGD (logistic regression)

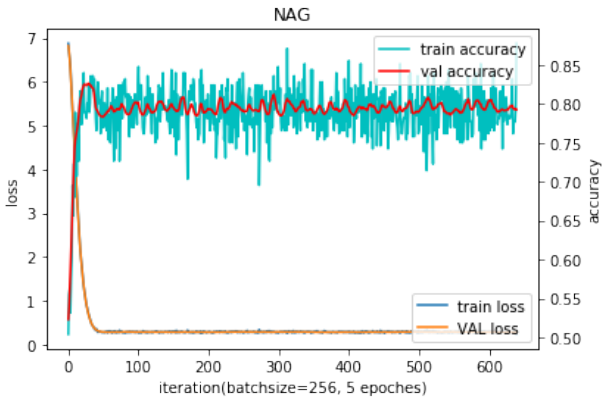


Fig.7. NAG (logistic regression)

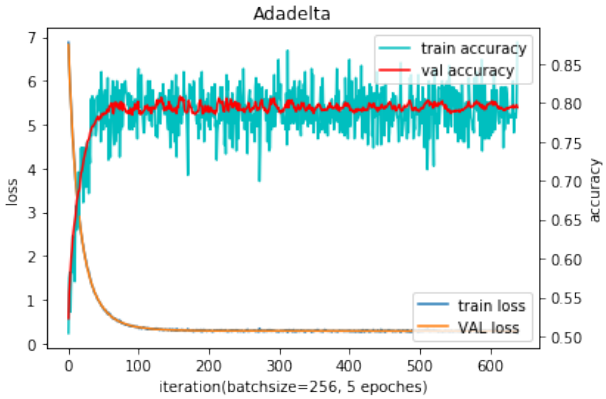


Fig.8. Adadelta (logistic regression)

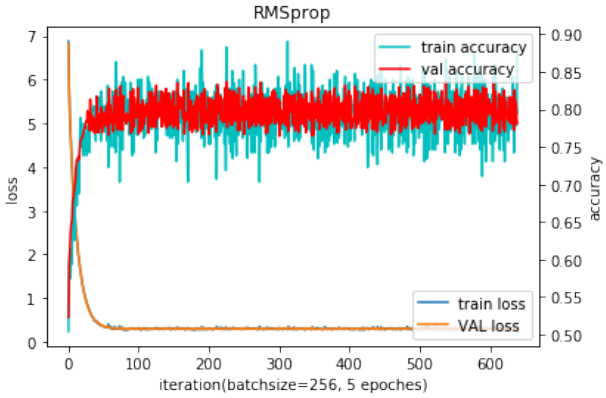


Fig.9. RMSprop (logistic regression)

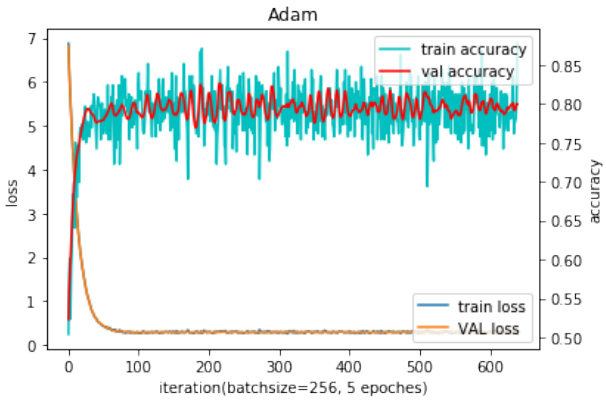


Fig.10. Adam (logistic regression)

D. Results of the algorithms on linear classification

The loss of different optimization algorithms on the valuation set using a linear classifier is shown in Fig. 11.

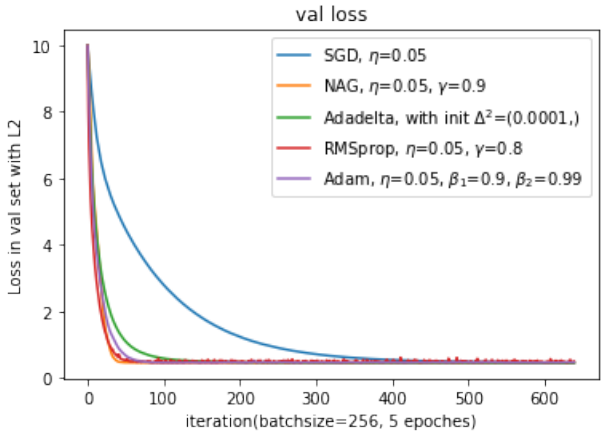


Fig.11. valuation loss (linear classification)

Fig. 12-16 show the result of SGD, NAG, Adadelta, RMSprop and Adam, which include the loss of the training set and valuation set, and the accuracy of the training set and valuation set.

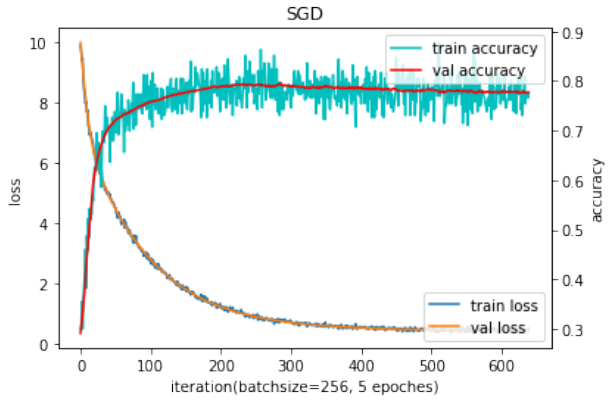


Fig.12. SGD (linear classification)

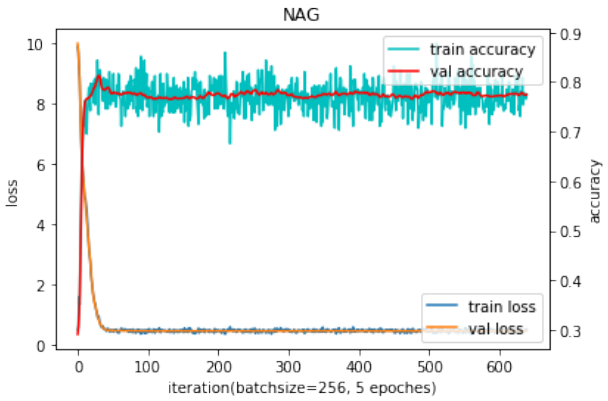


Fig.13. NAG (linear classification)

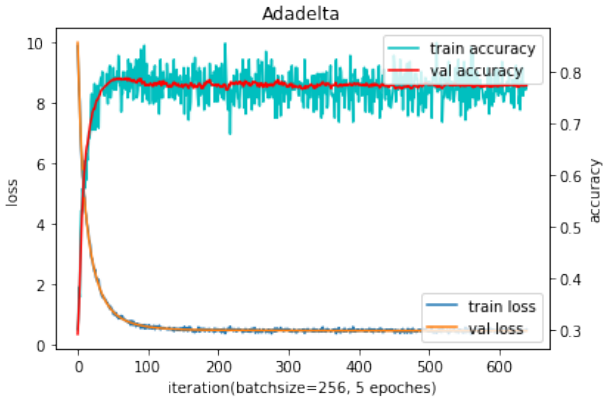


Fig.14. Adadelta (linear classification)

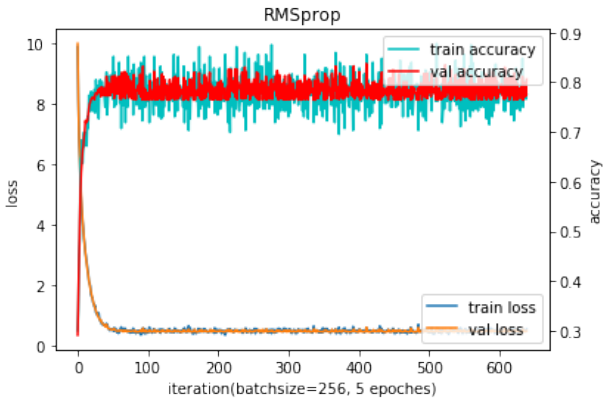


Fig.15. RMSprop (linear classification)

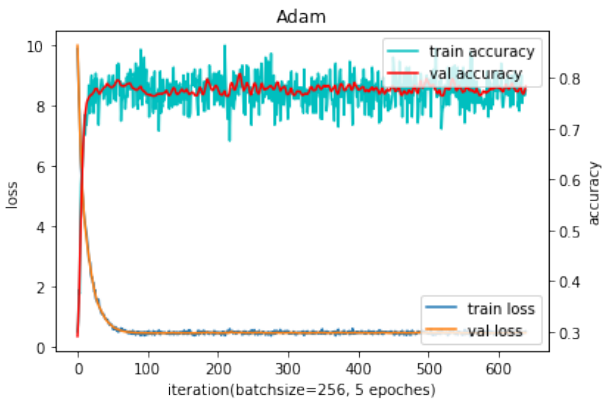


Fig.16. Adam (linear classification)

E. Analysis

The result of the experiments shows that SGD usually achieves to find a minimum, but it might take significantly longer time. If care about optimization speed, it is better to use adaptive learning rate methods or SGD with momentum.

It is also observed that among all adaptive learning rate methods, Adam seems performs the best, though there is no strong evidence show Adam is always the best.

iv. CONCLUSION

This experiment aims to understand the different between logistic regression and linear classification, moreover, to learn 5 optimization algorithms: SGD, NAG, Adadelta, RMSprop and Adam.

Logistic regression seems very similar to classification, but the logistic regression pays more attention on its regression operation where the regression object is sigmoid function. Logistic regression and SVM are both very widely used linear classifier.

SGD, NAG, Adadelta, RMSprop and Adam are all very widely used methods to optimize gradient decent. SGD is the most simple method that can always find a minimum, but with a longer time. Methods with adaptive learning rate or NAG is always the better choice if care about speed. And it seems that Adam performs the best.

There are many additional strategies for optimizing, for instance, batch normalization and early stopping. It is worth further study and research.

References

[1] Ruder S. An overview of gradient descent optimization algorithms[J]. 2016.

[2] Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. Proc. 8th Annual Conf. Cognitive Science Society.

[3] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks; The Official Journal of the International Neural Network Society, 12(1), 145–151.

[4] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

[5] Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. (2012). Advances in Optimizing Recurrent Networks. Retrieved from http://arxiv.org/abs/1212.0901

[6] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701

[7] Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13.