American University of Armenia

College of Science and Engineering

Optimization

# Optimization Algorithms for Neural Networks

*Team:*
Grigor Bezirganyan
Varlam Mkrtchyan
Marina Hambaryan

Spring, 2018

**Abstract**

The aim of the project is to study some of the first order optimization algorithms used in Neural Networks and compare their behaivour on the same data in terms of accuracy, loss and convergence time on a simple feed-forward Neural Networks. In particular, the project will study Nesterov Accelerated Gradient Descent, AdaGrad, RMSprop and Adam optimization algorithms. Two Neural Networks models will be introduced: one using the Keras library, the other using a self-written NN library.

# Contents

# 1 Introduction

## 1.1 Problem Setting and Description

We are going to create two Neural Networks models using different first order optimization algorithms to solve the same problem, which is: the prediction of customers' future behavioural tendency in bank, according to the customer information. At the end. the results of different optimization algorithms will be compered and a conclusion will be made.

## 1.2 The Structure of the Project Paper

The structure of the project is the following.
**Chapter 2** is the theoretical part where we introduce certain concepts that are mandatory to know before proceeding into the machine learning optimization problem solving. The experiments done for studying the algorithms as well as the conclusions will be done in **chapter 3**.

# 2 Theoretical Background

## 2.1 Machine Learning

Machine learning is a field of computer science which is aimed to make computers work without being explicitly programmed. Machine Learning tasks are classified into three main categories: supervised, unsupervised and reinforcement learning.

Supervised learning is the process of training the machine learning model on existing input and the correct output variables (X, y). The aim of supervised learning is to get a model trained on existing data so good, that later, provided only input data it can predict the output with minimal error.

In unsupervised learning we do not have an opportunity to measure the correctness of the predcition (we don't have the output variables (y)). Here tha machine learning algorithms tries to find patterns data structure or distribution based on hidden features of the data.

In reinforcement too we don't have the correct outputs, and the agent learns from experience by trial-and-error method trying to make the long-term reward as big as possible.

## 2.2 ANN

ANN (artificial neural network) is one of the supervised machine learning approaches. Its model is based on how biological neural networks are structured, hence the name Artificial Neural Network is derieved.

In ANNs the basic units are Neurons, which have an "Activation function" and are connected to other neurons. The connections between newurons are called "synapses". The input synapses are called dendrites and the output ones are called axons. Synapses have weights associated with them. Typically in ANN the neurons are organized in layers and each neuron is connected to some of[1] neurons in the next layer. Dhe data parameters are provided to ANNs in inpt layer, which, then, are passed to hidden layers. The last layer is the output layer, where we get the predicited values(Figure 1).

In order to calculate the output value of a neuron, the weight of each dendrite is multiplied with the outoput of connected neuron and sumed up. The final result is passed through function called **"activation function"**. There are different activation functions, such as Sigmoid, ReLU or Identity. The prpose of activation functions is to add non-linearity into the network.

The basic form of artificial neural networks are feedforward, although there are also recurrent algorithms. These are different from each other by the neuron connection structure.

**Feedforward connection** was represented in an overall description of the artificial neural network architecture (information moves in a forward direction: input layer, hidden layer and output layer).

On the other hand, in **Recurrent Neural Networks** (RNN) the signals can travel not only forward, but also backwards. In this way, previous calculations are recirculated, hence creating some kind of memory for the network. This memory can help to deal with dependant data variables, such as predicting the next word in the sentence.

After the data is passed through neural network and the outoput of neural network is derieved, the accuracy of prediction needs to be calculated usig the cost or error function.

[1] In fully-connected ANNs each neuron in a layer is connected to every neuron in the next layer

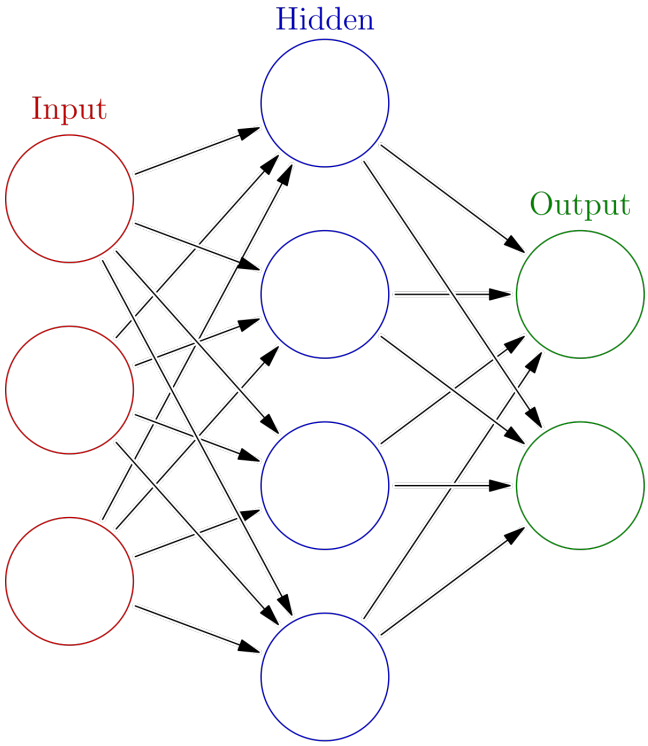



Figure 1: Neural network [8]

## 2.3 Training Neural Networks

*Training Neural Networks = Minimizing Cost/Loss function*

As it was mentioned, a neuron is a value: an activation function. Loss function represents a variance: error, difference from the expected value. The aim of the training is to minimize the variance. One of the useful techniques for optimizing the cost function is gradient descent. The algorithm for computing the gradient efficiently with respect to the neural network's weights and updating the wights to minimize the loss is called **backpropagation**.

The training process starts with initializing random weights to the synapses, which, obviously will not give the correct result. The improvement of the network is happening due to updating the weights to minimize the loss.

To minimize the cost we calculate the gradient of loss function and choose the descent direction. (hence, the term "**Gradient Descent**")

There are three variants of gradient descent.

### 2.3.1 Batch Gradient Descent

The first basic type of gradient decent algorithms is Batch Gradient Descent where the whole dataset is being used for gradient computation. Hence this algorithm is slow and requires lots

of memory. However, the accuracy is higher than in others.

$$w = w - a\nabla J(w) \tag{1}$$

w - weight, a – step size, $\nabla J(w)$ – gradient of the loss function

#### 2.3.2 Stochastic Gradient Descent

For faster computations Stochastic Gradient Descent shall be used where only one row from the dataset is being used for each step to compute the gradient of the cost function. This provides better performance and requires less resources for training on large datasets. This method, however, usually does not achieve the accuracy of the Batch Gradient Descent.

$$w = w - a\nabla J(w; x^{(i)}; y^{(i)}) \tag{2}$$

#### 2.3.3 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent is the most widely used among other algorithms. It mostly gives better results than in other two methods. In this case instead of taking one or all rows from the dataset, specified number of rows are being used for each gradient computation. This gives more stable convergence and more efficient gradient computation. Stochastic gradient descent is basically the Mini-Batch Gradient Decent when batch size is 1, hence, currently, in most stochastic gradient descent implementations batch size is being given, and by the term Stochastic Gradient Descent, often Mini-Batch Gradient descent is being meant. In our research as well, we will use Mini-Batch Gradient descent under the term Stochastic gradient decent.

$$w = w - a\nabla J(w; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3}$$

### 2.4 Back propagation

Back propagation = backward propagation of errors

In order not to calculate the gradient of the loss function, we can either approximate each partial derivative or use the easier method: the back propagation method. It is called backward because firstly we calculate the gradient of the last layer and in the end the gradient of the first layer. The computations of each layer depends of the calculations from the previously calculated ones.

$$\frac{\partial L}{\partial w_{ij}} = e_j \cdot x_i \tag{4}$$

where $\frac{\partial L}{\partial w_{ij}}$ is the derivative of the loss function with respect to the weight (synapse connecting the i-th neuron of the current layer to the j-th neuron of the next).
$e_j$ is the error obtained by subtracting the expected output value of the next neurons' j-th value from the real output value of the same neuron.
$x_i$ is our current neurons' value (activation function is applied).

### 2.5 Gradient Descent with Momentum

As we already know SGD is one of the best variants of gradient descent algorithms where, we have, stable convergence and more efficient gradient computation. For making SGD algorithm converge

more efficiently, a "momentum" is being added to classic SGD algorithm. SGD with momentum is Gradient decent method which helps gradient vectors to accelerate to the right direction efficiently. Here, instead of using original data, we find some moving average that brings us closer to the original function. For finding moving average exponentially weighted averages are being used. This is considered as classical momentum schemes. There are two main reasons why SGD with momentum works better than classic SGD. First of all in classic SGD we do not compute the exact derivative of loss function, instead we calculate errors batch by batch. Hence, chance of going in non optimal direction is higher. SGD with momentum provide us better estimation for derivative of loss function. The second reason why SGD momentum is better than classic one is because of level curves. In area where level curves change steeply, classic SGD has troubles navigating there.

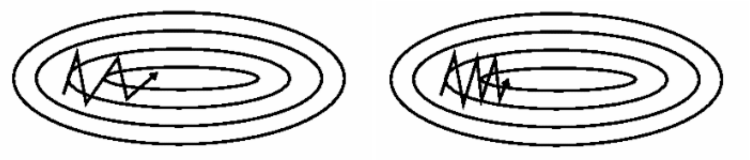


Figure 2: SGD with momentum vs classic SGD [7]

From Figure 2 we can see that Momentum helps SGD to accelerate gradients in the right direction and overall convergence is faster than in classic SGD.

## 2.6 Nesterov Accelerated Gradient Descent

The Nesterov Accelerated Gradient (NAG) is optimization method for improving stability and convergence of classic gradient decent. NAG consists of a gradient descent step, followed by momentum term. Here momentum is not the same as in classical approach. To understand what is the main difference let's have a look at Figure 3

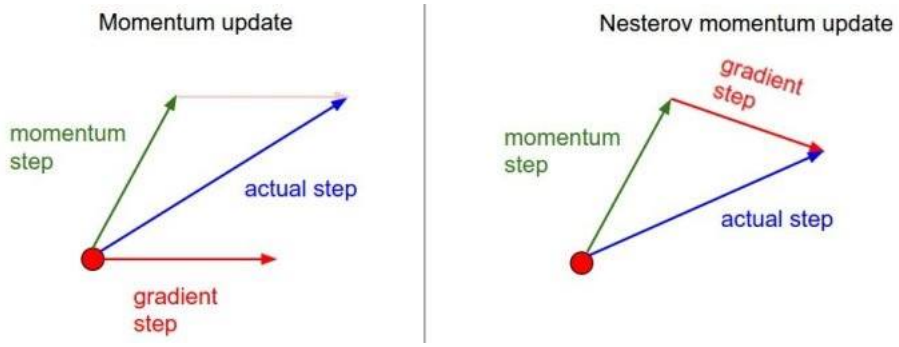



Figure 3: Momentum vs Nesterov [7]

In classical momentum update we can see that we have Gradient step vector and momentum step vector, based on these two vectors we do our actual step. In Nesterov momentum update we do momentum step, end position of which is considered as our net initial point for calculating Gradient step vector. After that we do our actual step based on momentum and gradient step

vectors. The Nesterov Accelerated Gradient is adapting updates to the slope of the error function and it is speeding up stochastic gradient decent.

## 2.7 Adagrad

The Adaptive Gradient Algorithm or Adagrad is modified stochastic gradient decent with "per-parameter" (adaptive) learning rate. The main reason for developing this algorithm is changing the learning rate of parameters. It is increasing learning rate for infrequent/rare parameters and decrease rate for frequent ones. It is mostly used when dealing with sparse data. This kind of strategy mostly improves convergence rate of classic stochastic gradient decent mainly in language processing, image recognition and computer vision problems, where sparse parameters have more valuable information. Nevertheless, the leanring rate of AdaGrad has monotonically decreasing behaivour, which affects the converge time and the accuracy of the algorithm.

## 2.8 RMSProp

RMSProp or Root Mean Square Propagation algorithm is another method used in neural network training. In this method, again learning rate is adapted for each of the parameters: We calculate the parameter learning rates based on the first moment average. The algorithm tries to resolve the problem of AdaGrad's decreasing learning rate. The main idea of RMSProp is to divide the learning rate for a weigh by a running average of the magnitudes of recent gradients for that weight.

## 2.9 Adam

Another adaptive gradient descent algorithm, which tries to fix the dedcreasing learning rate problem of AdaGrad is the Adam algorithm. Adaptive Moment Estimation or Adam is described as combination of two NN training optimization algorithms which are Adaptive Gradient Algorithm and Root Mean Square Propagation. Adam takes benefits from both algorithms and gives us better result. As we already know In RMSProp algorithm we adapt the parameter learning rates based on the average first moment, Adam is using both average of the first moment and the average of second moment of the gradient. Adam optimization algorithm is using another important addition from Adagrad which is increasing learning rate for infrequent/rare parameters and decrease rate for frequent ones.

# 3 Experiments

## 3.1 Models and configuration

For the experiments two different models with same configurations are being used. One of the models is created using the Keras library, the other, using our own NNlibPy library.

- Input Parameters: 11
- Input size: 10,000
  - train set size: 8000
  - test set size: 2000
- Loss function: Mean Squared Error
- Initial weights: Random from normal distribution
- Batch-size: 32
- Epochs: 70 or 250 (depending on optimizer)
- Layers
  - Input layer: 1
    - Neurons: 11
    - Activation function: Identity
  - Hidden layers: 2
    - Neurons in each layer: 6
    - Activation function: ReLU
  - Output layer
    - Neurons: 1
    - Activation function: Sigmoid
- Optimizers
  - SGD
  - SGD with Momentum
  - Nesterov Accelerated Gradient Descent
  - Nesterov Accelerated Gradient Descent with Momentum
  - AdaGrad
  - RMSprop
  - Adam

## 3.2 Data and problem

For experiments a dataset of customers of a bank will be used. The dataset can be found in this website.

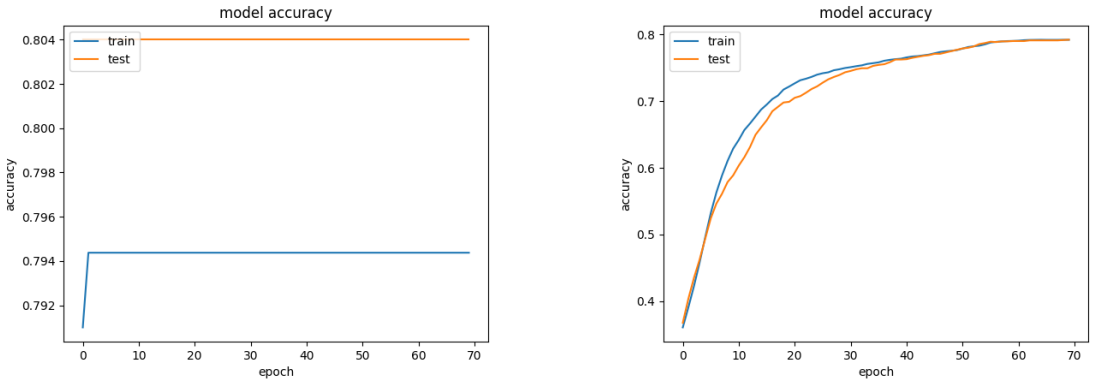
The problem is to train a model which, based on inputs can predict whether the customer will leave the bank or no.

## 3.3 Results

We start the experiments by training the models with same optimizers, and afterwards we compare the results.
Please note that the problem is to compare the convergence rates of loss and accuracy functions and not the actual values of the functions. In many of examples the Keras library finds a better minimum than the NNlibPy. Since it is enough to compare the algorithms we didn't concentrate on getting out from local minimum while developing the NNlibPy.

### 3.3.1 Experiment 1

- Optimizer: Stochastic Gradient Descent
  - Learning rate: 0.01
- Epochs: 70



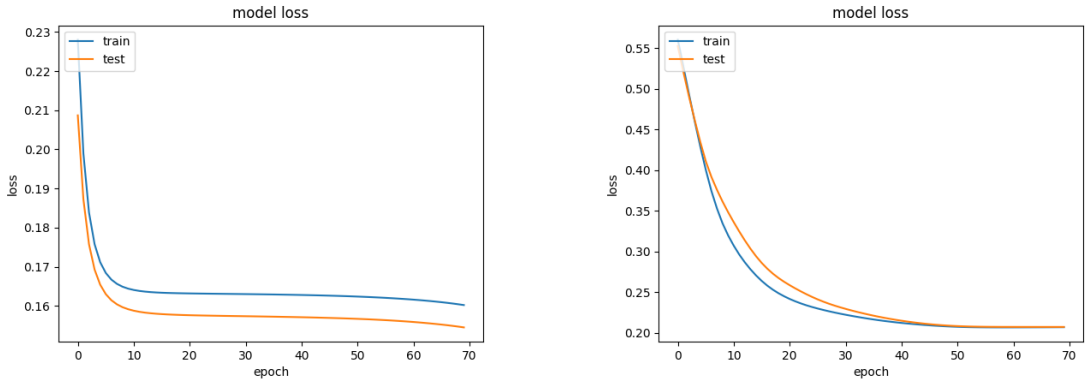

(a) Keras

(b) NNlibPy

Figure 4: SGD Accuracies

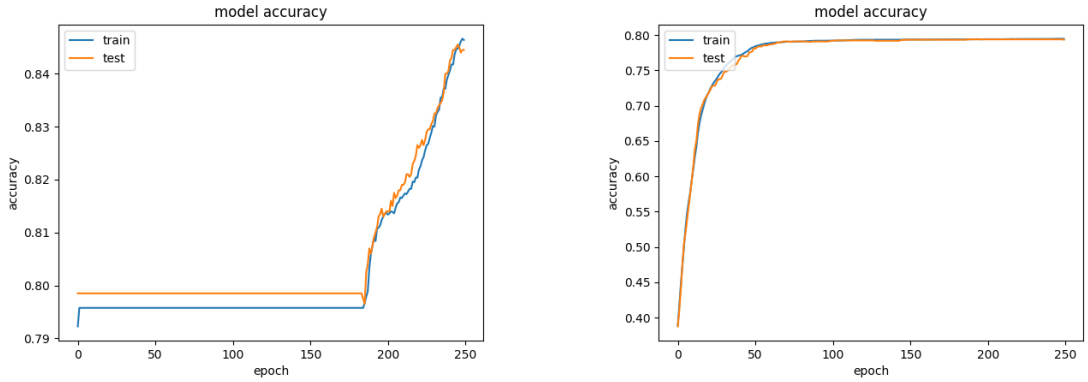
From Figure 4 it can be seen that in NNlibPy's results the accuracy increases and seems to converge starting from around epoch 55. On the other hand, in Keras' results accuracy rapidly increases and then stays constant. To understand this behaviour we look at loss functions' graphs in Figure 5. It can be seen that the loss of NNlibPy starts to converge from around epoch 50, whereas the loss of Keras keeps decreasing. This gives reason to assume, that since the loss keeps decreasing, later, the accuracy may increase. To test the assumption we do the training one more time with 250 epochs.

From Figure 6 we can see that Keras' accuracy starts to increase rapidly from around epoch 180, while NNlibPy's accuracy stays the same. There is possibility for NNlibPy to increase its accuracy later, after many more epoches, but since the aim of the project is to compare the algorithms, we can already register the resuls after 70 and 250 epoches. In fact, most ptobably the NNlibPy just gets stuck in one of its local minimum, hence the accuracy and the loss seems to converge. There are several ways to test this hypothesis, nevertheless, since converging to local optimum is enough to compare algorithms, we will not concentrate on getting out from local minimum trap.




(a) Keras (b) NNlibPy

Figure 5: SGD Losses



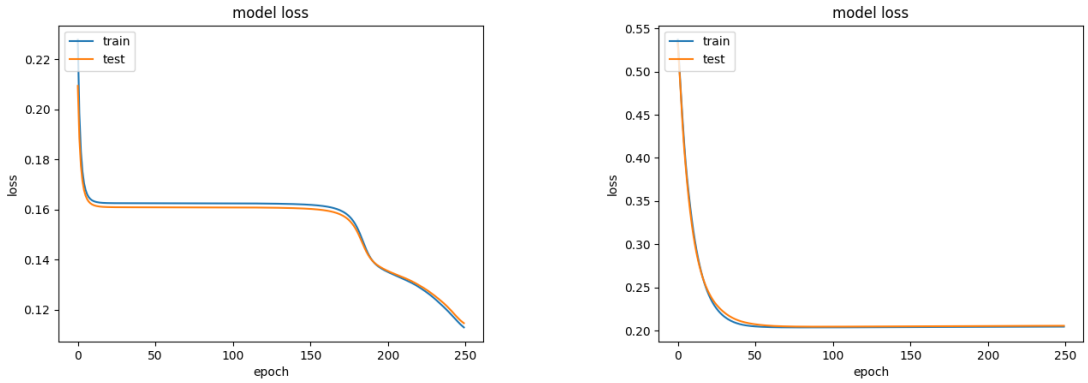

(a) Keras (b) NNlibPy

Figure 6: SGD Accuracies (250 epochs)

### 3.3.2 Experiment 2

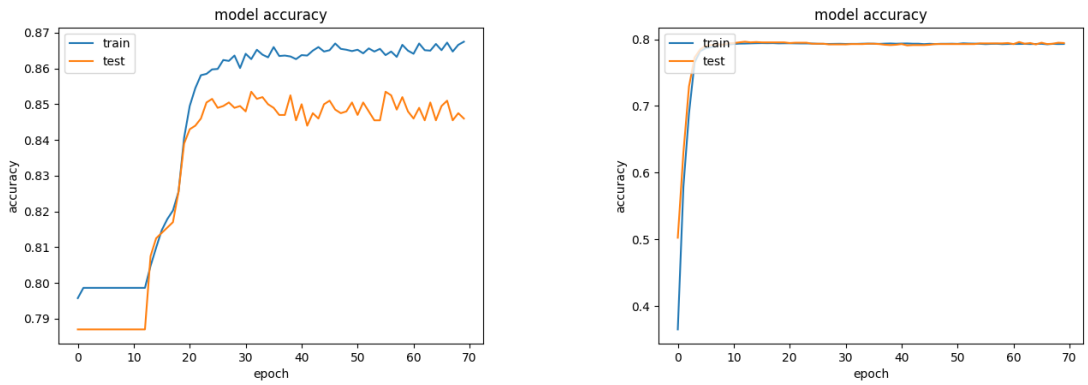
- Optimizer: Stochastic Gradient Descent with momentum
  - Learning rate: 0.01
  - Momentum: 0.9
- Epochs: 70

In this experiment we give momentum to our previous configuration. It is evident that with momentum the accuracies and the losses tend to their optimums a lot faster than without it. As described in theory this is the expected way, since the momentum kind of "pushes" down to optimum.




(a) Keras (b) NNlibPy

Figure 7: SGD Losses (250 epochs)




(a) Keras (b) NNlibPy

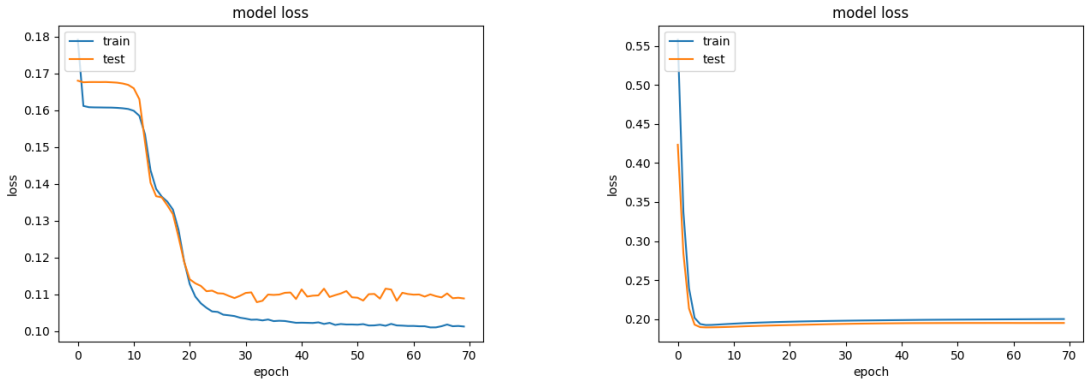
Figure 8: SGD with momentum Accuracies

Please note that in graphs the NNlibPy's curve decreases faster, but this is because the initial loss value starts from a higher point (0.55) compared to Keras (0.20)[1].

### 3.3.3 Experiment 3

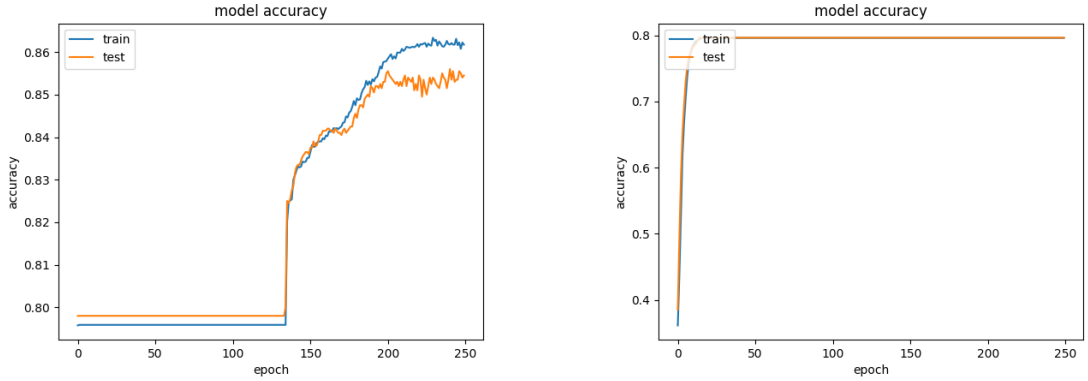
- Optimizer: Nesterov Accelerated Gradient Descent
  - Learning rate: 0.01
  - Momentum: 0
- Epochs: 250

[1]This is done intentionally to make the converging process more visible in NNlibPy, by setting that standard deviation to 1 instead of 0.1 during weight initialization




(a) Keras (b) NNlibPy

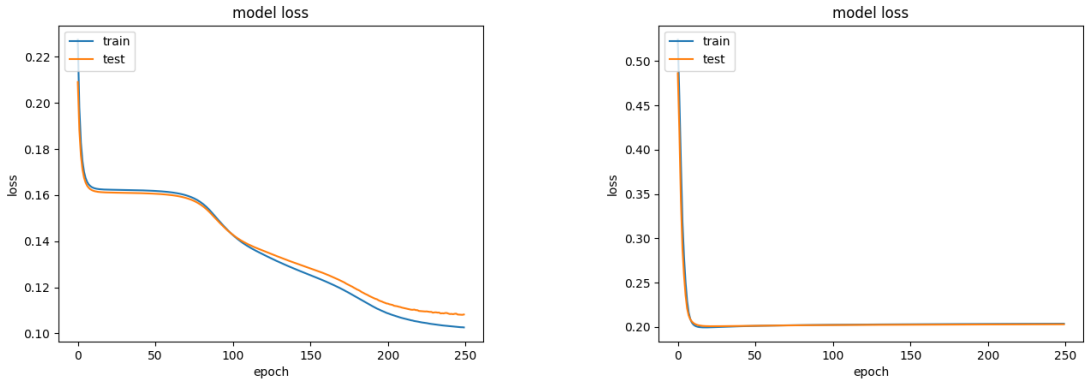
Figure 9: SGD with momentum Losses




(a) Keras (b) NNlibPy

Figure 10: Nesterov Accelerated GD Accuracies

Here (Figure 10) we apply Neterov Accelerated Gradient Descent without moemntum. We can see that Nesterov converges to local opimum faster. With Keras the functions converge to their optimum faster (starting from around epoch 140, compared to SGD which converged starting from around epoch 180) compared to SGD. NNlibPy, nevertheless, converges to its local optimum pretty fast, but fails to get out from it.

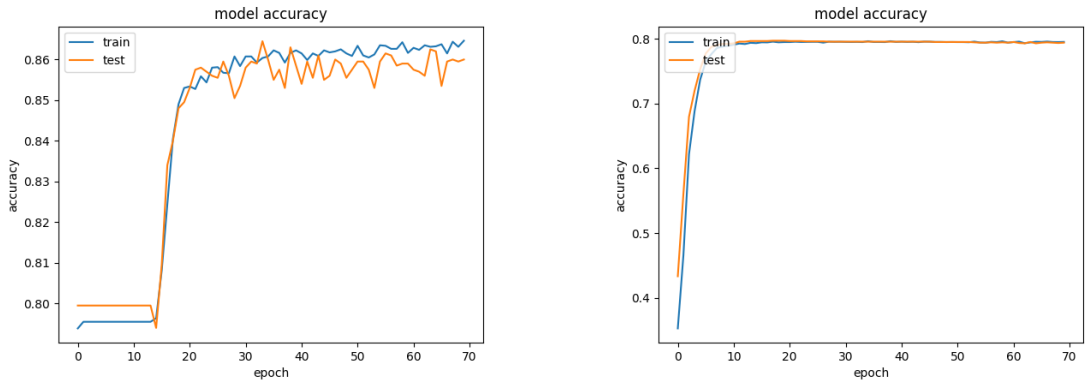



(a) Keras (b) NNlibPy

Figure 11: Nesterov Accelerated GD Losses

### 3.3.4 Experiment 4

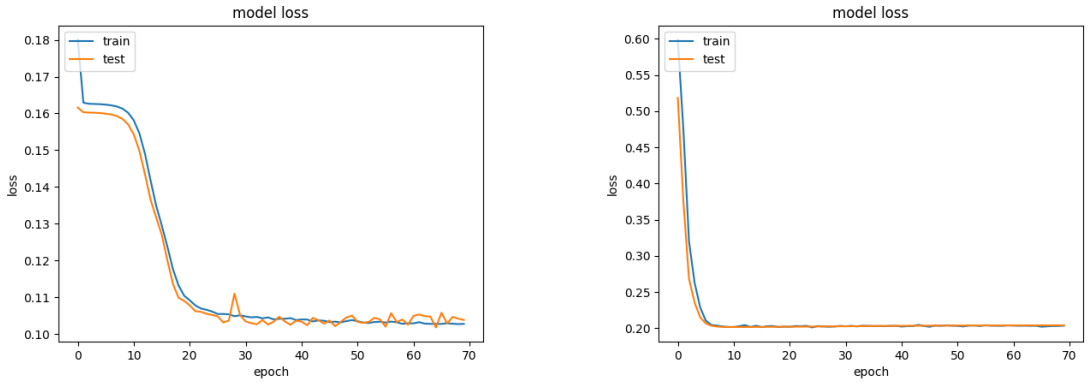
- Optimizer: Nesterov Accelerated Gradient Descent with momentum
  - Learning rate: 0.01
  - Momentum: 0.9
- Epochs: 70




(a) Keras (b) NNlibPy

Figure 12: NAG with momentum Accuracies

In this experiment (Figures 12, 13) we apply Neterov Accelerated Gradient Descent with moemntum. As expected moemntum "pushes" the loss downwards (and hence, accuracy upwards) to their optimums and they converge a lot faster compared to NAG without momentum. In Keras again we see that after finding a local minimum from around epoch 15 a better optimum is found.

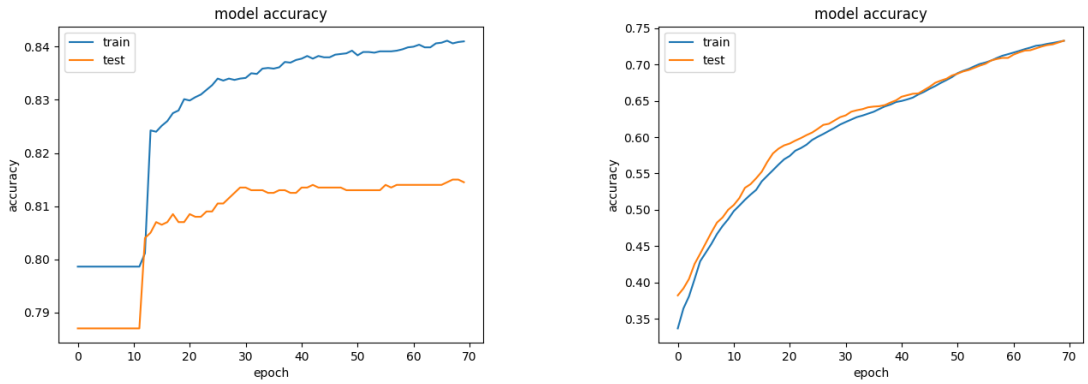



(a) Keras (b) NNlibPy

Figure 13: NAG with momentum Losses

### 3.3.5 Experiment 5

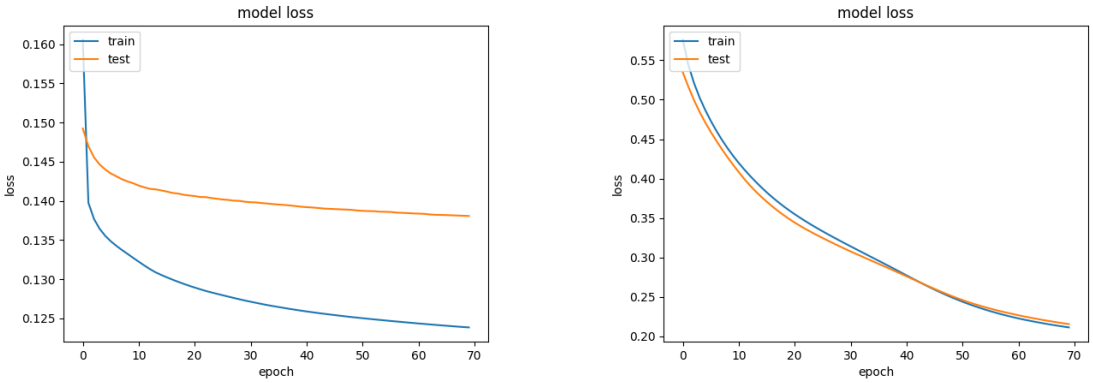
- Optimizer: Adagrad
  - Learning rate: 0.01
- Epochs: 70




(a) Keras (b) NNlibPy

Figure 14: Adagrad Accuracies

In this experiment (Figures 14, 15) we apply Adagrad algorithm. The main strenghth of this algorithm was adaptive learning rate. Nevertheless, we can see that in NNlibPy it converges slower than previous algorithms and in Keras the functions don't accuire their previous bests. This is because of monotonically decreasing learning rate in Adagrad's algorithm, which was fixed in Adadelta and RMSprop ( 3.3.6).

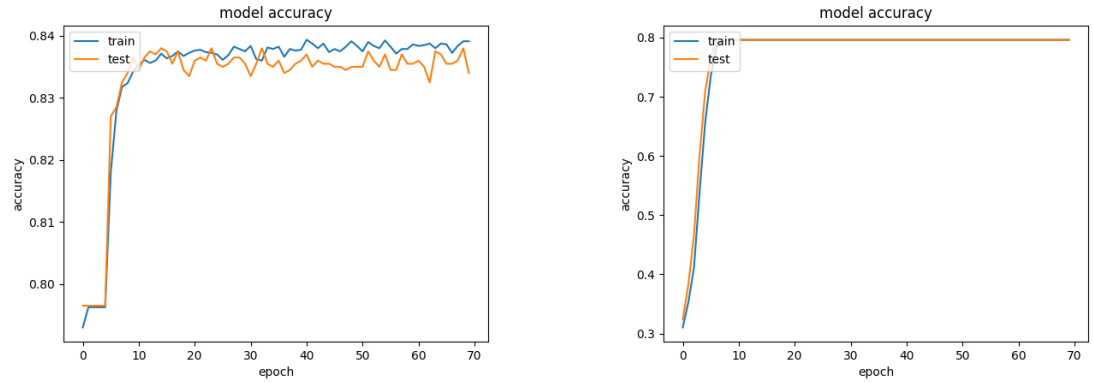



(a) Keras (b) NNlibPy

Figure 15: Adagrad Losses

### 3.3.6 Experiment 6

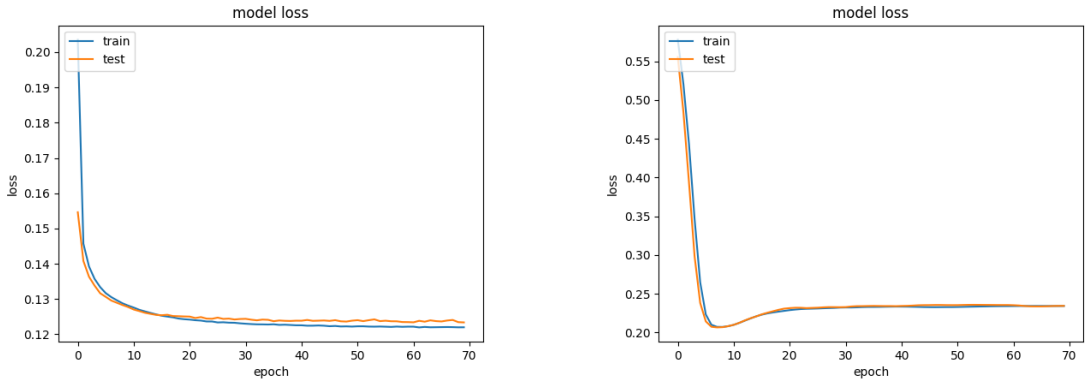
- Optimizer: RMSprop
  - Learning rate: 0.001
  - Gamma: 0.9
- Epochs: 70




(a) Keras (b) NNlibPy

Figure 16: RMSprop Accuracies

Here (Figures 14, 15) we apply RMSProp algorithm which intends to fix the gradually decreasing learning rate problem of Adagrad. As we can see in Figures 16 and 17 in our example RMSprop successfully completed its task and in both models converged pretty fast.

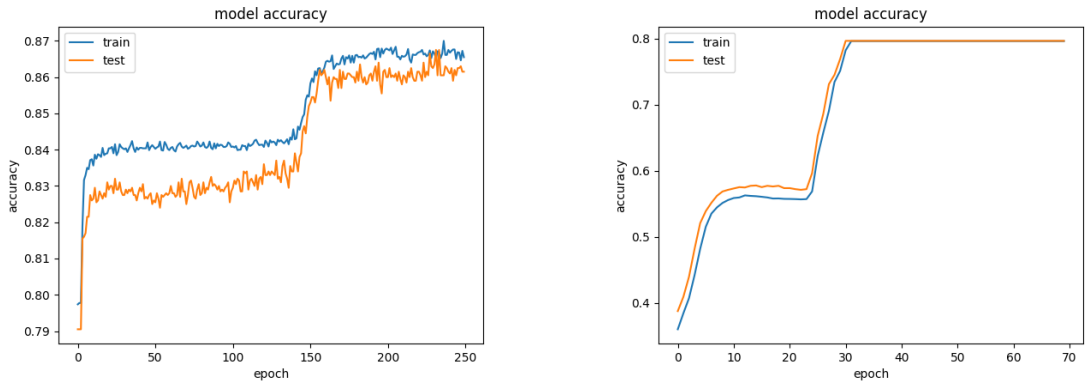



(a) Keras (b) NNlibPy

Figure 17: RMSprop Losses

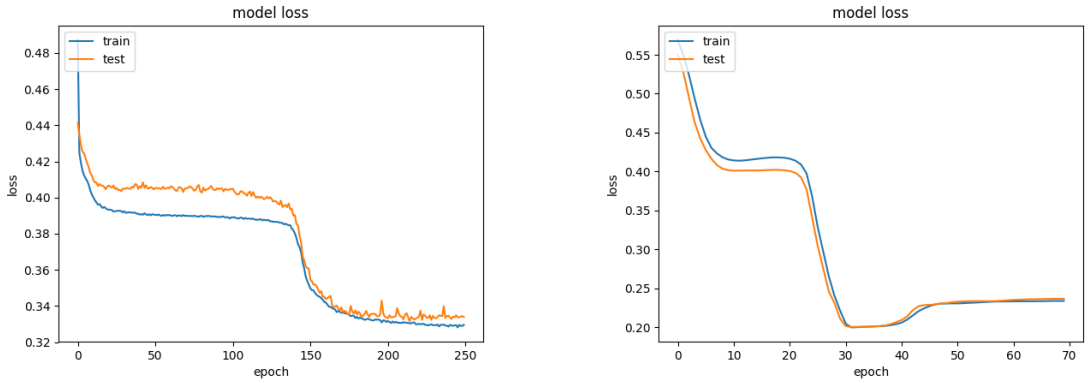
### 3.3.7 Experiment 7

- Optimizer: Adam
  - Learning rate: 0.001
  - Beta 1: 0.9
  - Beta 2: 0.99
- Epochs:
  - Keras: 250
  - NNlibPy: 70




(a) Keras (250 epoches) (b) NNlibPy (70 epoches)

Figure 18: Adam Accuracies

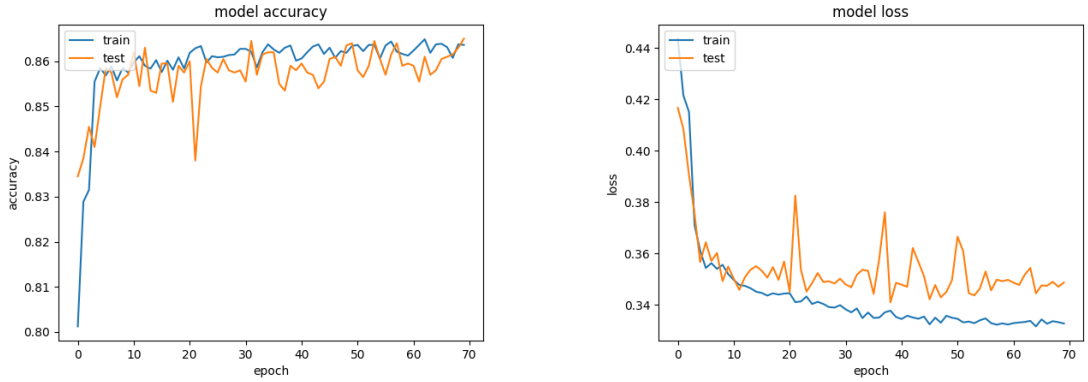



(a) Keras (250 epoches)

(b) NNlibPy (70 epoches)

Figure 19: Adam Losses

From Figures 18 and 19 can see that Adam optimizer has similar convergence rate in both models. At first it finds a local minimum, than converges to a better one. Although the NNlibPy's curves converge faster, the Keras' model obtains better optimum. Let's note in both both we took the default parameters suggested in Adam's paper. Since most of previous algorithms were run with bigger lr (0.01), we tested the Adam optimizer on Keras' model second time with lr = 0.01 (Figure 20).



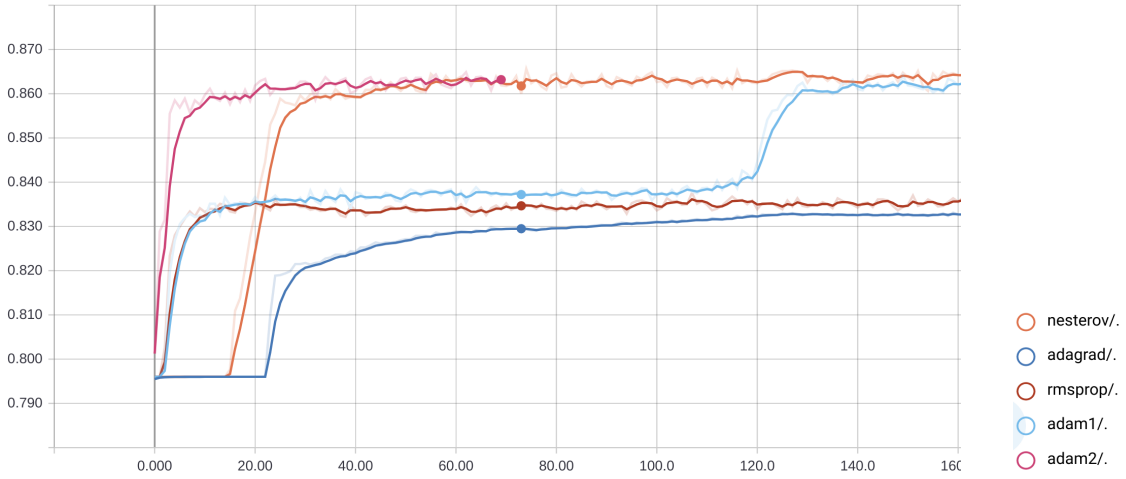

(a) Accuracy

(b) Loss

Figure 20: Keras, Adam optimizer (lr: 0.01)

Here we get faster convergence, but we can see also less smoother curve, with high varience. This is because higher learning rate causes our function to pass the optiumum value several times. To sum up, although Adam has adaptive learning rate, it is still benficial to try different values and choose the most optimal one fur the problem.

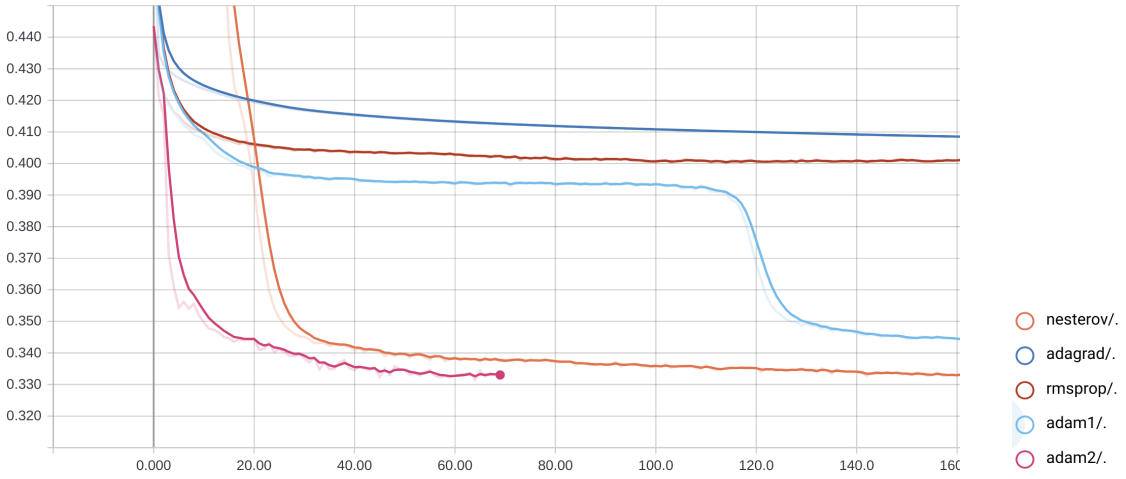
### 3.3.8 Summary




(a) nesterov, adagrad, rmsprop, adam(lr = 0.001), adam2(lr = 0.01)

Figure 21: Accuracies

From Figures 21 and 22 we can see the comparitive graphs of optimzers descusssed above[1]. We left out some of the optimizers like SGD or DAG without momentum, to keep the plot cleaner, and included the ones which give more important information. We can see that The fatsest converging optimizers with better optimums are the Nesterov Accelerated Gradient Descent and Adam with 0.01 learning rate.




(a) nesterov, adagrad, rmsprop, adam(lr = 0.001), adam2(lr = 0.01)

Figure 22: Losses

[1]Please note that although the results are similar to the ones mentioned in other papers, based on training data and the compiled model the results will vary from our results.

The RMSProp converges pretty fast, but fails to obtain the optimum obtained by Adam or NAG optimizers. Adam optimizer with 0.001 learning rate converges late, but obtains better maximum value, while, on the other hand, adagrad, both converges late and doesn't achieve good maximum. As descussed before, this is the expected behaivour for adagrad, since the monotonically decreasing learning rate doesn't let adagrad to converge faster and to better optimum. This problem was intended to be fixed in RMSProp and Adam. If we take into account only the convergence speed, it can be seen from experiments and from the final plot that both Adam and AdaGrad succeed to complete that task. Nevertheless, the RMSProp doesn't manage to obtain the optimum got by NAG and Adam on our example, and repeats the optimum value of AdaGrad.

If we compare the results of Keras' and NNlibPy's models, we can see that with convergence speed they have very similar behaivour. The only difference between them is the optimum value, in which, evidently Keras is the winner [1]. From experiments it is also visible the importance of experimanting with learning rate, since even with adaptive algorithms, changing learning rate may give better results. From experiments it is also evidet the role of momentum, since in SGD and NAG the results were hugely improved after adding momentum.

[1] As mentioned before, while developing the NNlibPy, we didn't concentrate on maximizing the accuracy, since the primary objective of our study is the behaivour of the algorithms and their converggance rate

# References


[1] Yu. Nesterov | Gradient methods for minimizing composite functions
https://doi.org/10.1007/s10107-012-0629-5

[2] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159.
http://stanford.edu/~jduchi/projects/DuchiHaSi10_colt.pdf

[3] Geoff Hinton. RMSProp, Curseria Lecture 6a, Overview of mini-batch gradient descent
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[4] Sebastian Ruder. An overview of gradient descent optimization algorithms
https://arxiv.org/abs/1609.04747

[5] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015
http://neuralnetworksanddeeplearning.com/

[6] Gradient Descent Algorithm and Its Variants – Towards Data Science, 2018.
https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

[7] Vitaly Bushaev, Stochastic Gradient Descent with momentum, 2017
https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d

[8] Wikipedia, Artificial Neural Networks, Retreived May 2018 https://en.wikipedia.org/wiki/Artificial_neural_network

[9] R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996 | The Backpropagation Algorithm
https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf