# Advanced Concepts of Machine Learning: Implementing Backpropagation

Kirill Shcherbakov (i6248484), Shashank Subramanya(i6257859)

November 04, 2020

## 1 Introduction and general information

In this assignment we implemented the basic version of backpropagation for a small neural network and explored the sensitivity of its hyperparameters. The neural network was designed as a class in Python. Various initialization solutions for the weights and bias terms of our network were tried and the best result was shown by the following initialization: Xavier initialization for the weights initialization (because these serve as good starting points for initialization and mitigate the chances of exploding or vanishing gradients. They set the weights neither too much bigger than 1, nor too much less than 1. So, the gradients do not vanish or explode too quickly. This helps to avoid slow convergence, also ensuring that we do not keep oscillating off the minima.) and zeroes for bias terms.

Following the difference of the backpropagation explanation by Professor Kurt and Andrew Ng, two functions were tried as cost function:

- Quadratic cost (Mean squared error (MSE))
- Cross-entropy cost

It can be noted that using Cross-entropy cost our model requires significantly fewer iterations (almost 5 times less) than when we use Quadratic cost (Mean squared error). However, all subsequent experiments that will be described were done using the MSE cost function.

## 2 Software description and how to use it

### 2.1 Prerequisites

The following dependencies must be installed to run the code:

1. Python 3.x
2. Google Colaboratory service/ Jupyter Notebook
3. NumPy 1.18.5
4. Matplotlib 3.2.2

### 2.2 How to launch the code?

- Firstly, you should open the corresponding notebook in Google Colaboratory service or locally on your computer with Jupyter Notebook.
- In the notebook the code is arranged in a sequential way. In order to reproduce the corresponding results, you need to open a notebook in Google colab (preferably using GPU runtime) and run each cell of the code sequentially, then wait for the result to run.
- The following training parameters are available for modification: the number of iterations ("iterations"), the learning rate ("learning rate"), weight decay ("weight decay").

## 3 Description of the learning performance

We can change and examine the following hyperparameters of our network to achieve the best performance: the learning rate ($\boldsymbol{\alpha}$) and the regularization parameter ($\boldsymbol{\lambda}$).

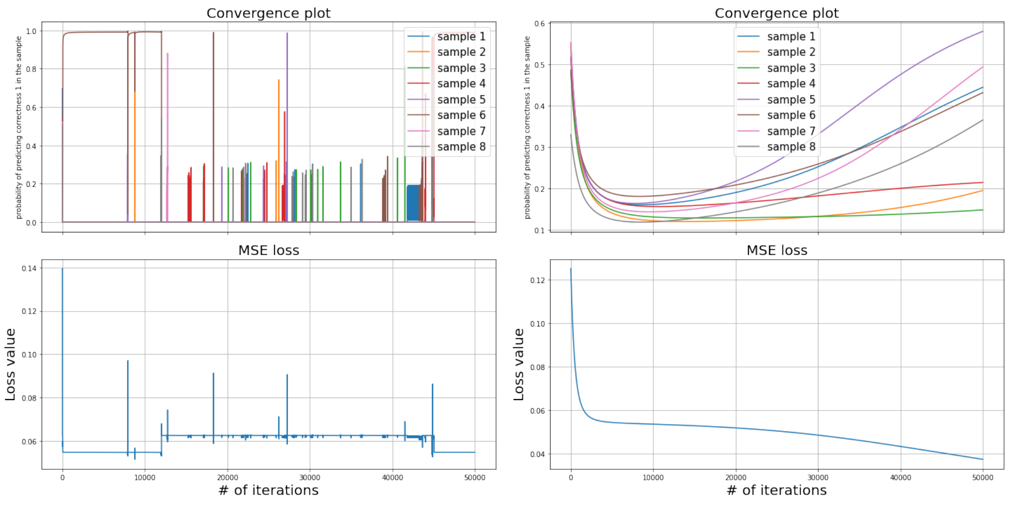
At extremes, a learning rate $\boldsymbol{\alpha}$ that is too large will result in weight updates that will be too large and the performance of the model (such as its loss on the training dataset) will oscillate over training epochs. Oscillating performance is said to be caused by weights that diverge (are divergent). A learning rate that is too small may never converge or may get stuck on a suboptimal solution.

The regularization parameter $\lambda$ determines how you trade off the original cost with the large weights penalization.

### 3.1 Understanding the impact of the Learning Rate on Neural Network Performance

Figure 1(a) shows a case when the learning rate $\alpha$ is extremely large. Larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution. But it might “jump” over the best configurations and “step over” local minimas in the gradient descent and it can lead to divergence. And this is what we see on Figure 1(a). The first figure shows the convergence of each object in the training sample, or rather shows the probability of correctly predicting one in each training object, which is equivalent to the convergence of each object, because as long as one in each sample has high enough probability others will be 0 after setting the threshold (we chose the threshold for the sigmoid output to be equal 0.5). We can see that at the initial stage, gradient descent manages to optimize the weights so that some predicted values for training objects converge to the correct value. However, due to the large learning rate coefficient, high weights adapt too strongly and rapidly, the gradient descent jumps very much and “step over” the optimal value, and we observe oscillation. This can also be seen in the second graph, which shows the MSE error, where we see a sharp drop, and then we see local jumps of the error, which means that our algorithm tries to optimize the cost and eventually changes the weight too much.

A learning rate that is too low will take a long time to converge. And we can see that on Figure 1(b). This is especially true if there are a lot of saddle points in the loss-space. We see that even 50 thousand iterations are not enough, that all the training objects have come together to the correct solution. Also, the MSE error graph shows that the error drops sharply after about 3000 iterations, after which it decreases slowly and slightly due to a small learning rate value.



(a) (b)

Figure 1: (a) $\boldsymbol{\alpha} = 100.0$ and $\boldsymbol{\lambda} = 0.00001$ (b) $\boldsymbol{\alpha} = 0.01$ and $\boldsymbol{\lambda} = 0.00001$

On Figure 2(a) we can see that the change to the learning rate is not linear. Based on this behavior and the influence of the learning rate on our MSE we can also conclude about the optimal value and choose it to be equal to 10. Also, usually, the learning rate and the weight decay are independent of each other and we cannot make a conclusion about the weight decay value knowing the optimal value of the learning rate.

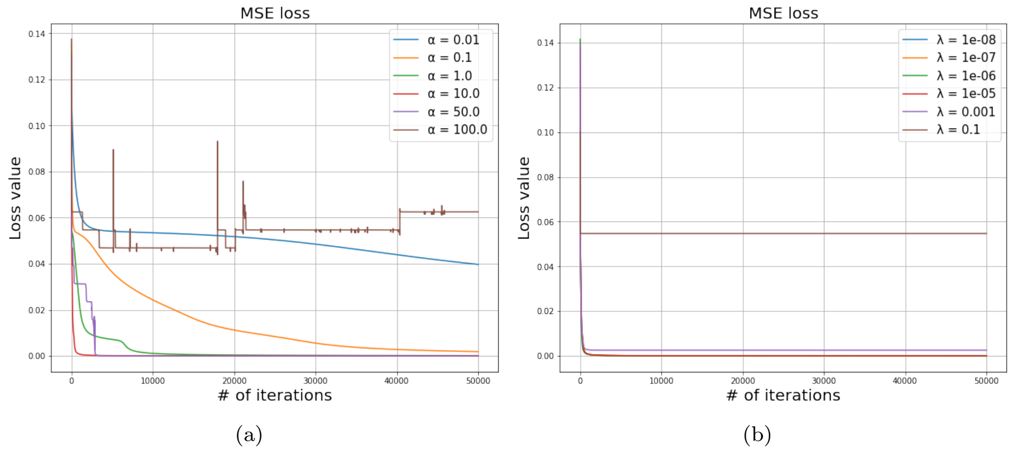


Figure 2: (a) Comparison of the performance for different $\boldsymbol{\alpha}$ with the constant $\boldsymbol{\lambda} = 0.00001$ (b) Comparison of the performance for different $\boldsymbol{\lambda}$ with the constant $\boldsymbol{\alpha} = 10.0$

### 3.2 Understanding the impact of the Regularization Parameter on Neural Network Performance

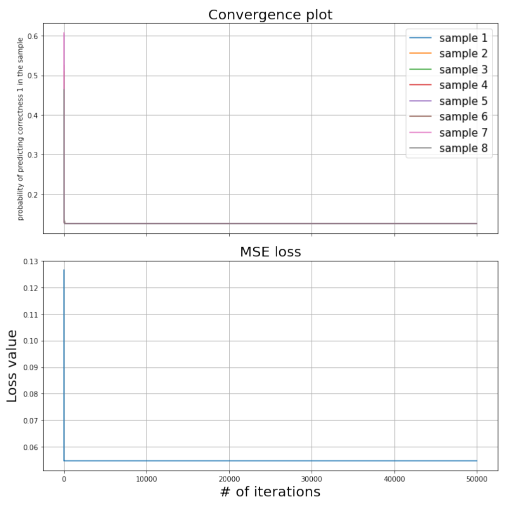
By adding the weight decay we penalize complexity. We will still use a lot of parameters, but we will prevent our model from getting too complex. The parameter weight decay controls the relative importance of the two parts of the error function. If we set it to a high value, the network does not care so much about correct predictions on the training set and rather keeps the weights low, hoping for good generalization performance on the unseen data. And that is what we see in Figure 3(a). Here the learning rate of 10 and weight decay equal to 0.1 is used. We can also see that no training object was predicted correctly and they did not converged. And it is easy to explain. With weight decay we are in a competition between the values of weights and the cost term in our lost function. By setting a large value to the weight decay, we give great importance (breaking the competition by helping the regularization member in the loss function) to the weight of the regularization term in the loss function, it practically does not pay attention to the difference between the predicted values and the real ones, but only tries to make the weights small.

Putting extremely small values means that we practically remove the regularization of weights and do not penalize them. On Figure 3(b) we see our training samples converge and the MSE decrease with a very low value of weight decay.

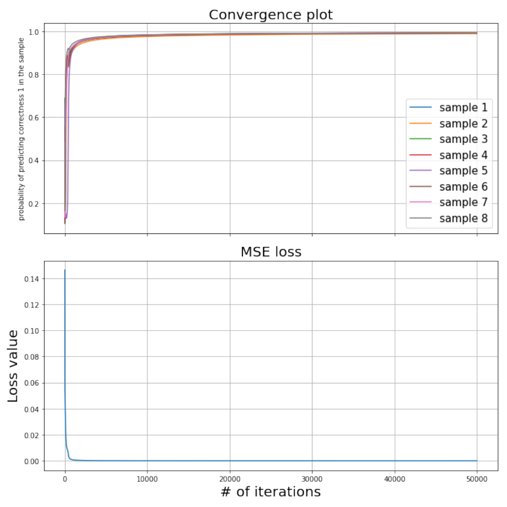
## 4 Interpretation of weights of the network

Based on the results obtained during the experiments with the learning rate and the weight decay, the final model was chosen with $\boldsymbol{\alpha} = 10.0$ and $\boldsymbol{\lambda} = 0.00001$ as it converged relatively quickly in about 3000 iterations and had the lowest cost value of 0.000081 among the parameters we experimented with(Figure 4).

To understand the functioning of the weights, let's analyze the activation values of the hidden layer $a^{(2)}$ and output layer $a^{(3)}$. Figure 5 provides the values of the activation nodes(rounded off to 2 decimal places) for each of the 8 inputs. The neural network accurately predicts the input value through the output of $a^{(3)}$. In $a^{(2)}$, if we consider 0.5 as the threshold value, we observe that each input value is represented as a unique code using the 3 nodes. For example, input [1,0,0,0,0,0,0,0] is represented as *low,high,high*. This is similar to



(a)



(b)

Figure 3: (a) $\boldsymbol{\alpha} = 10.0$ and $\boldsymbol{\lambda} = 0.1$ (b) $\boldsymbol{\alpha} = 10.0$ and $\boldsymbol{\lambda} = 0.00000001$

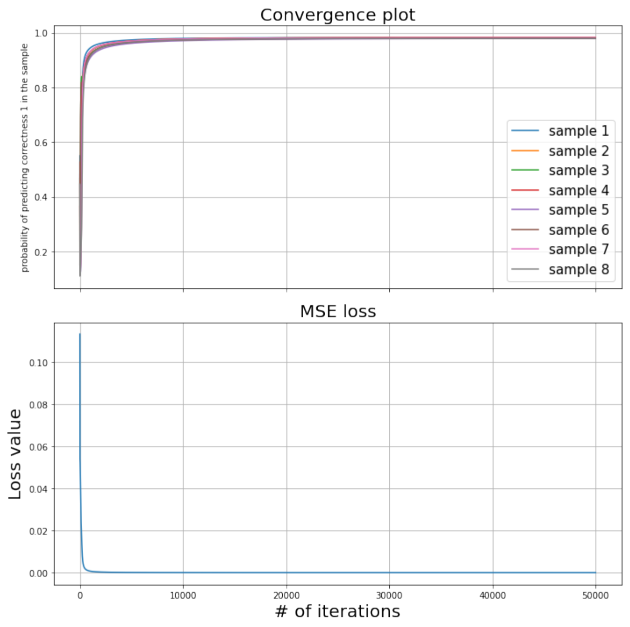


Figure 4: $\boldsymbol{\alpha} = 10.0$ and $\boldsymbol{\lambda} = 0.00001$

a binary representation of numbers 0 to 7 in 3 bits, except that this neural network has arbitrarily assigned one value to each input.

The weights in the network are responsible for encoding the input value to the hidden layer and then decoding them back to the original value in the output layer. Thus, it is likely that there is a similar pattern in the weights $W^{(1)}$ and $W^{(2)}$. And, from Figure 6 (weights rounded off to 1 decimal place) it can be observed that the sign of weight from the input layer to the activation node in the hidden layer is the same as the weight from the same node to the corresponding output layer in all cases except one. A similar pattern was observed in the values of bias $b^{(1)}$ and $b^{(2)}$.

| Input | Activation $a^{(2)}$ | | | Activation $a^{(3)}$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10000000 | 0.02 | 0.99 | 0.97 | 0.98 | 0.00 | 0.01 | 0.00 | 0.01 | 0.02 | 0.00 | 0.00 |
| 01000000 | 0.01 | 0.02 | 0.09 | 0.00 | 0.98 | 0.02 | 0.00 | 0.01 | 0.00 | 0.02 | 0.00 |
| 00100000 | 0.01 | 0.05 | 0.98 | 0.02 | 0.01 | 0.98 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 |
| 00010000 | 0.98 | 1.00 | 0.32 | 0.00 | 0.00 | 0.00 | 0.97 | 0.01 | 0.02 | 0.02 | 0.00 |
| 00001000 | 0.01 | 0.89 | 0.02 | 0.02 | 0.01 | 0.00 | 0.02 | 0.98 | 0.00 | 0.00 | 0.00 |
| 00000100 | 0.99 | 0.60 | 1.00 | 0.01 | 0.00 | 0.00 | 0.02 | 0.00 | 0.97 | 0.00 | 0.01 |
| 00000010 | 0.88 | 0.34 | 0.00 | 0.00 | 0.01 | 0.00 | 0.02 | 0.00 | 0.00 | 0.97 | 0.02 |
| 00000001 | 0.90 | 0.00 | 0.54 | 0.00 | 0.00 | 0.01 | 0.00 | 0.00 | 0.02 | 0.02 | 0.97 |

| Legend | |
|---|---|
| 🟩 | > 0.5 |
| 🟥 | <= 0.5 |

Figure 5: Activation values for $\boldsymbol{\alpha} = 10.0$ and $\boldsymbol{\lambda} = 0.00001$

| Weight $W^{(1)}$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| -4.0 | -4.3 | -4.2 | 4.1 | -4.4 | 4.7 | 2.1 | 2.2 |
| 4.6 | -3.8 | -2.6 | 5.6 | 2.4 | 0.6 | -0.4 | -5.4 |
| 3.4 | -2.5 | 3.5 | -1.0 | -4.1 | 5.3 | -5.9 | -0.06 |

| Weight $W^{(2)}$ | | |
|---|---|---|
| -5.5 | 8.6 | 7.6 |
| -7.0 | -10.0 | -8.8 |
| -5.7 | -8.9 | 9.5 |
| 7.9 | 12.0 | -4.2 |
| -6.8 | 8.5 | -9.5 |
| 8.1 | 1.5 | 12.0 |
| 8.8 | -4.6 | -17.3 |
| 9.2 | -17.0 | 2.9 |

| Legend | |
|---|---|
| 🟩 | > 0 |
| 🟥 | < 0 |

Figure 6: Weight matrix values for $\boldsymbol{\alpha} = 10.0$ and $\boldsymbol{\lambda} = 0.00001$