Numerical Optimization and its Applications: Final Project

Shahaf Finder 204037618
Roy Uziel 203398854

July 12, 2018

1 Project Overview

The goal of this project is to implement a simple neural network for classification of small dimention data.

The objective function will be softmax regression.
The network will have L residual layers (L is a parameter) of the shape:

$$
\begin{aligned}
&f(W_1, W_2, b, x) = x + W_2\sigma(W_1x + b)\\
&W_1 \in \mathbb{R}^{n\times n}\\
&W_2 \in \mathbb{R}^{n\times n}\\
&b \in \mathbb{R}^{n}
\end{aligned}
$$

2 Objective Loss Function

- We've computed Softmax and Loss in **softmax.m** and **loss.m**.
- The derivatives for the Softmax are in **loss_grad_theta.m** (gradient by weights) and **loss_grad_x.m** (gradient by X).
- **test_loss.m** is the gradient test for the loss function.
- **loss_SGD.m** is an SGD variant for the loss function.
- **Q3.m** is a script used to run the SGD.

We didn't notice a significant difference using different batch size and learning rate.

Results for learning rate: 0.005, batch size: 100

Figure 1: Peaks Validation Data

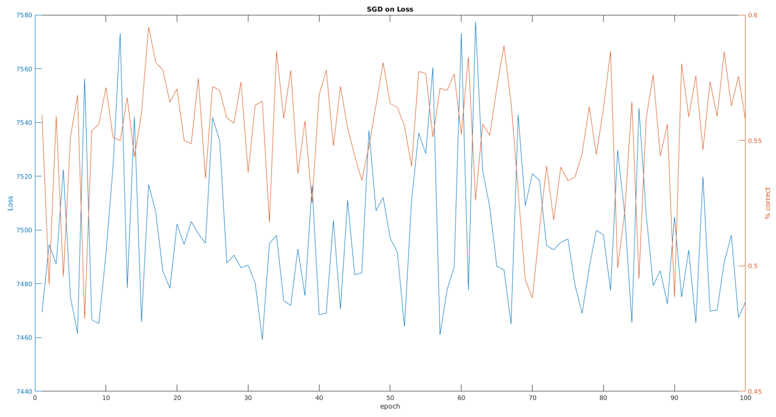
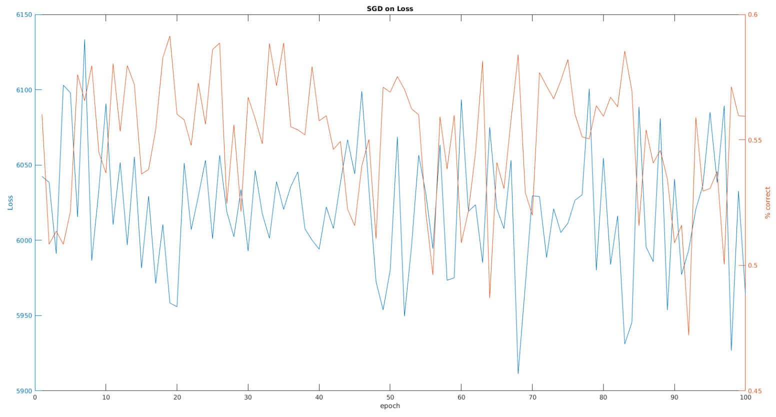


Figure 2: Peaks Training Data



3 Residual Neural Network

3.1 Jacobian Transpose Times Vector

ResNN_jac_theta_t_mul.m and **ResNN_jac_x_t_mul.m** are the jacobian transpose times vector for each layer.

Both can receive multiple samples X (as an n times m matrix), and their respectives V (as an n times m matrix) to multiply by.

- **ResNN_jac_theta_t_mul.m** returns the average result among all samples.

- **ResNN_jac_x_t_mul.m** returns the multiplication of each sample with it's v (as an n times m matrix).

These functions are used in the back propagation process:

$$\nabla_{\theta^{(l)}} f_l = \frac{\delta f_l}{\delta \theta^{(l)}}^T \nabla_{x^{(l+1)}} f_{l+1}$$
$$\nabla_{x^{(l)}} f_l = \frac{\delta f_l}{\delta x^{(l)}}^T \nabla_{x^{(l+1)}} f_{l+1}$$

In order to test those functions we've created two auxiliary functions ResNN_jac_theta_mul.m ResNN_jac_x_mul.m
These auxiliary functions compute:

$$\frac{\delta f}{\delta \theta} v, \frac{\delta f}{\delta x} v$$

We used the gradient test to make sure these functions are correct, and then used the jacobian transpose test with the required functions.
Those tests are in **test_resnn.m**.

3.2 Forward Pass and Back Propogation

forward_pass.m and **back_propogation.m** are the implementations.

To test them we created test_net.m that uses forward pass and back propogation to compute the gradient,
then each iteration we used forward pass to compute the loss value after moving to a direction.

3.3 SGD and Conclusions

ResNN_SGD.m is the SGD variant for the entire network.
run_net.m is a script used to run the SGD.

We've tested different network lengths and batch sizes on different data sets.
We got to the following conclusions:

- Increasing batch size increased the training time, and above 50 it had little effect on results.
- Each dataset required different network lengths for good results.
- Lowering the learning rate resulted in a more steady but slower convergence. We found out that, generally, 0.01 is a good balance between speed and steadiness.

Figure 3: GMM validation Data - 3 layer

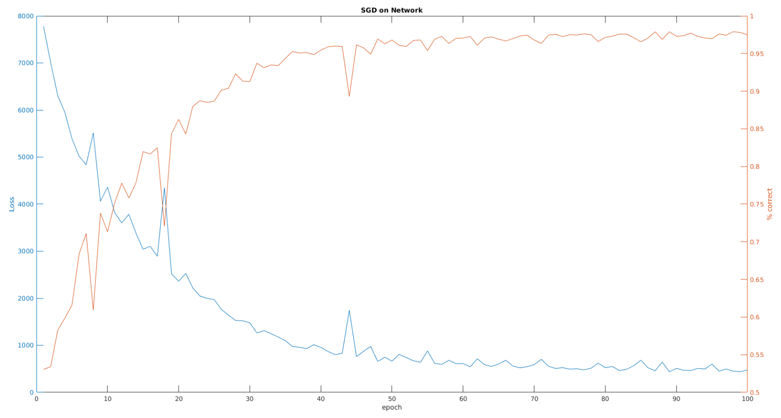
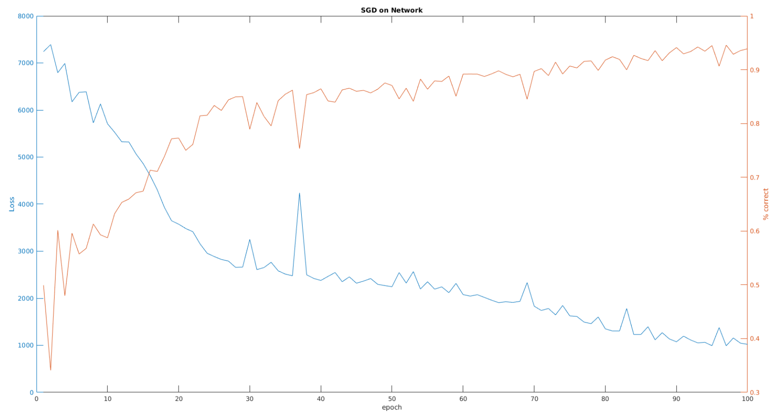


Figure 4: Peaks validation Data - 10 layer

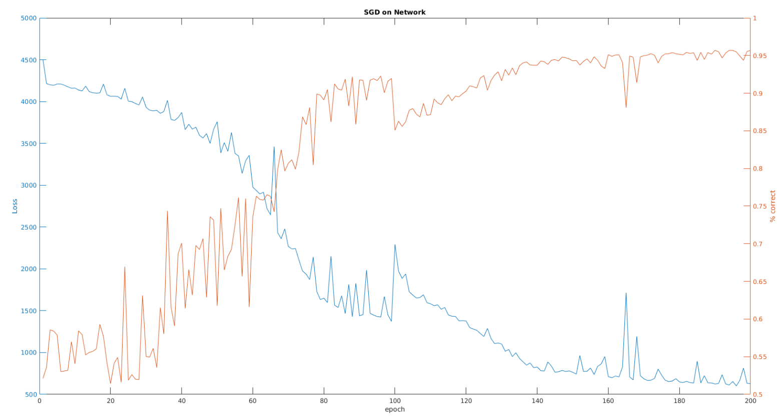


4 Improving on SGD

In seek to improve the results of SGD, we tried the following:

- Decreasing alpha according to iteration
  After 100 iterations we calculated a new alpha by: $learning\_rate * 5/sqrt(i)$. This is in order to half the learning rate and keep decreasing as the iterations continue.
  The method gave minor improvement.
- Adding momentum (as seen in class)
  The method got much better results in convergence speed, but was un-

Figure 5: SwissRoll validation Data - 15 layer



steady in the long run.
So in order to deal with the unsteadiness, after 100 iterations we calculated a new gamma using: $moment\_gamma * 10/sqrt(i)$.
This decreases the momentum as itereations continue.
The results were a fast convergence in a more steady way.

- Taking only the rejection part of the momentum.
  While trying to optimize even further, we tried to take only the projection of the momentum over the gradient.
  We calculated it by: $m\_proj = g * m' * g/(norm(g)^2)$.
  We found out that by adding only the projection part, the results got worse, so we tried adding the rejection part.
  Calculated by: $m\_rej = m - m\_proj$.
  The results were usually better, or at worse - the same, than the normal momentum.

Using these 3 imrovements made us able to reduce the number of layers without hurting the results too much.

Comparison between the 3 improvements.

All of the following figures are with the following parameters - batch size: 50, learning rate: 0.01, gamma (only for momentum): 0.7
Each figure shows 3 different run with each of the 3 improvements, made with the same initial weights (chosen randomly) and same seed for randperm (chosen randomly).

Blue - Using decreasing alpha only.
Orange - Using decreasing alpha, and decreasing momentum.
Yellow - Using all of the improvements specified above.

Figure 6: GMM Data - 1 layer

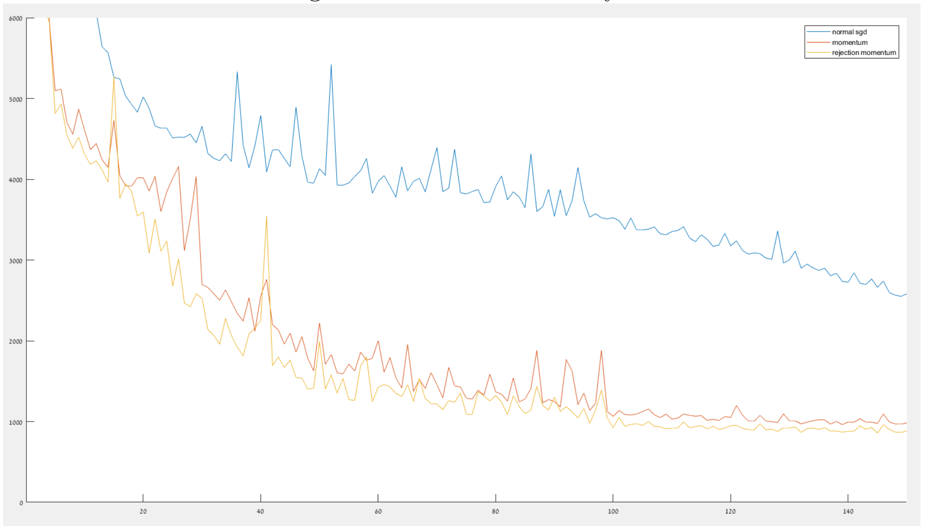


Figure 7: GMM Data - 2 layer

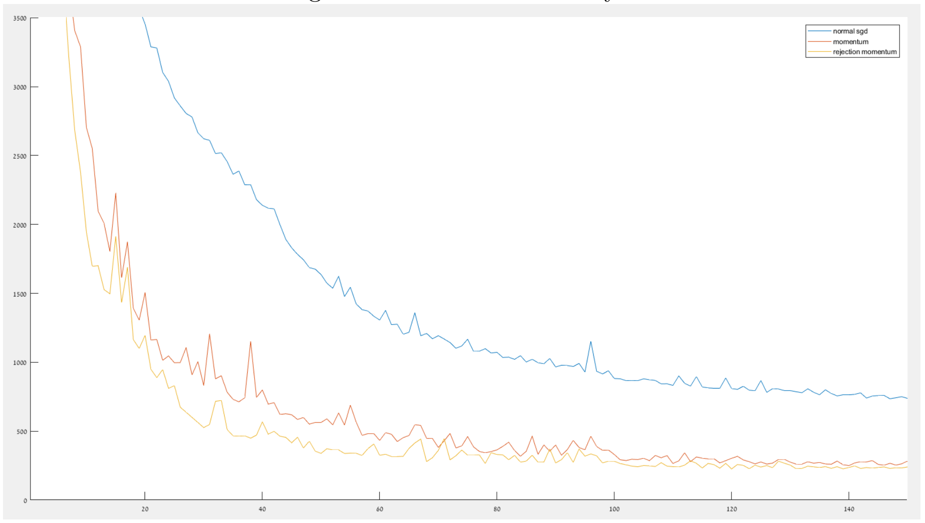


Figure 8: Peaks Data - 4 layer

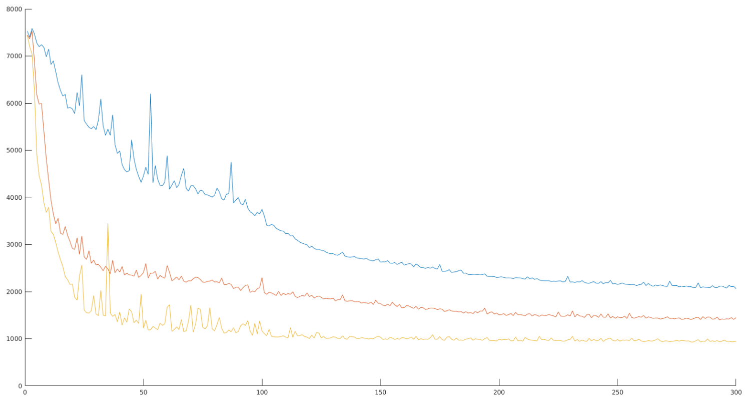


Figure 9: Peaks Data - 6 layer

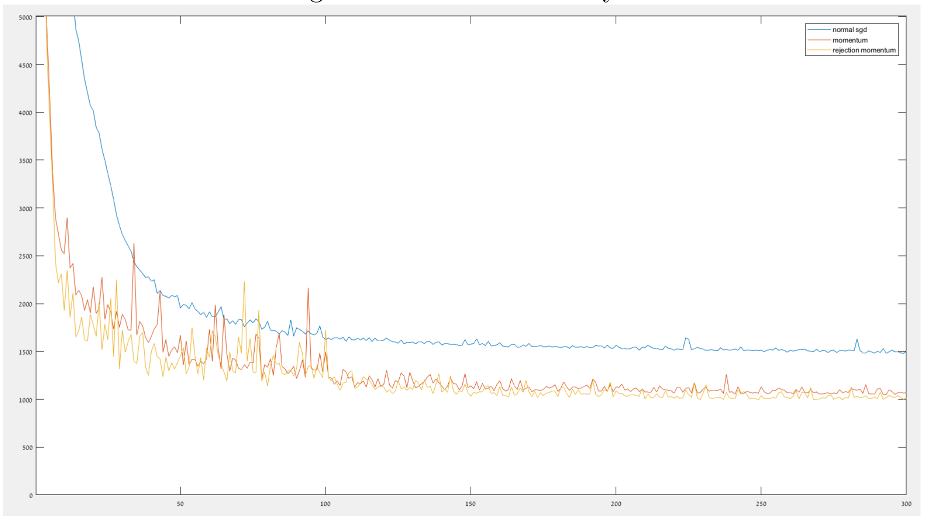


Figure 10: SwissRoll Data - 4 layer

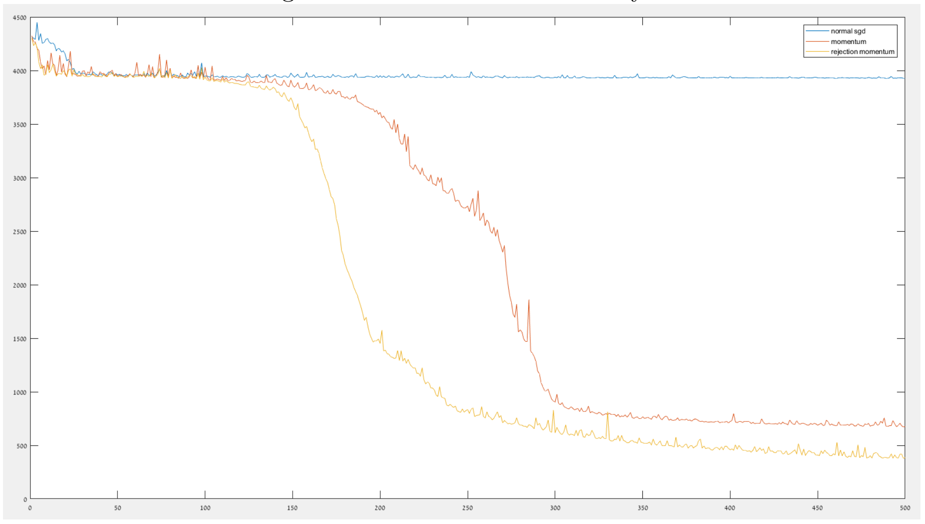
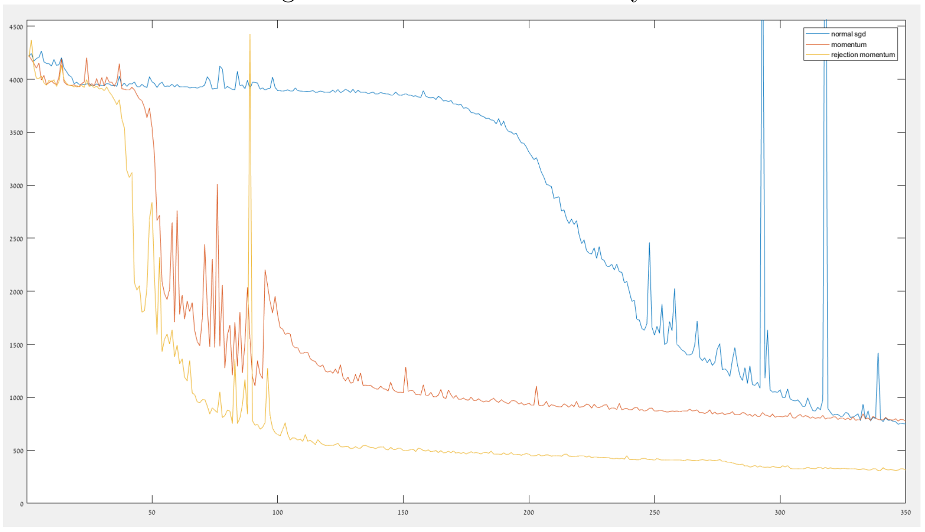


Figure 11: SwissRoll Data - 5 layer


normal sgd
momentum
rejection momentum