FRTN50 hand in 3

Karl Tengelin & Daniel Jogstad

October 2019

1 Introduction

Task 1

For the SVM problem from the previous hand in, implement the accelerated proximal gradient method above. Use the hyper-parameters that gave the best performance in the previous hand-in. Compare ordinary proximal gradient with (3) and one of (1) and (2). Use the same initial point x^0 and step size $\gamma = \frac{1}{L}$ for all methods where L is the smoothness-constant of f. For μ in (3) you could either calculate the strong-convexity parameter directly or tune it until you get convergence.

Plot $\left\|x^k - x^\star\right\|$ where x^* is the solution. Find x^* by simply solving the problem to high precision before generating your plot. Is the convergence rate improved over the non-accelerated method? Can you comment on the behaviour seen in the plot?

For task 1, we first solve the same problem as in the last assignment, using proximal gradient. The hyper parameters used was $\lambda = 0.0001$ and $\lambda = 0.5$. Then we resolve it by using acceleration with β:s set to

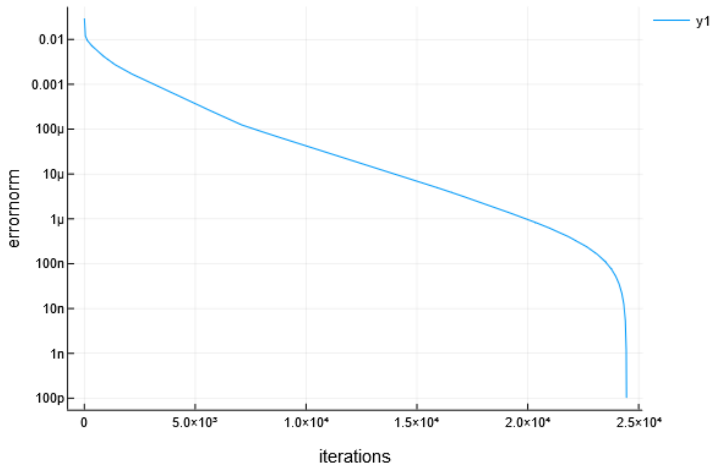
$$
\begin{gathered}
\beta 1^k = \frac{k-2}{k+1} \\
\beta 2^k = \frac{t^k - 1}{t^{k+1}}, \quad t^0 = 1, \quad t^{k+1} = \frac{1 + \sqrt{1 + 4\left(t^k\right)^2}}{2} \\
\beta 2^k = \frac{1 - \sqrt{\mu\gamma}}{1 + \sqrt{\mu\gamma}}
\end{gathered}
\tag{1}
$$

where μ was tuned to give the best results.

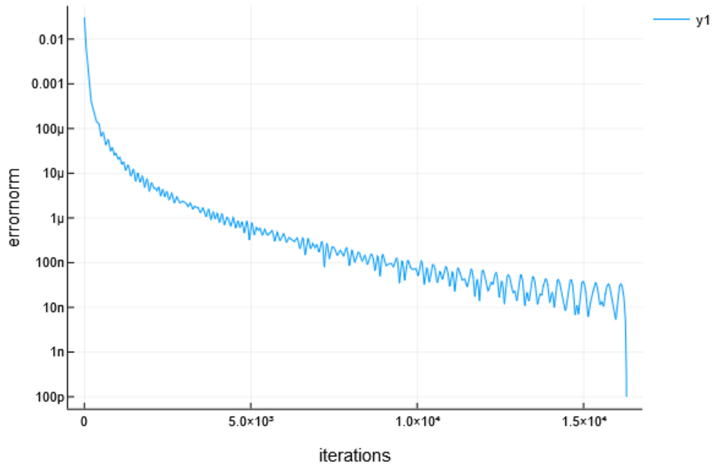
Proximal gradient method with acceleration can be done in several ways, but the one that was given in the assignment instruction was

$$\begin{aligned} x^{k+\frac{1}{2}} &= x^k + \beta^k \left(x^k - x^{k-1}\right) \\ x^{k+1} &= \operatorname{prox}_{\gamma g}\left(x^{k+\frac{1}{2}} - \gamma \nabla f\left(x^{k+\frac{1}{2}}\right)\right) \end{aligned} \tag{2}$$

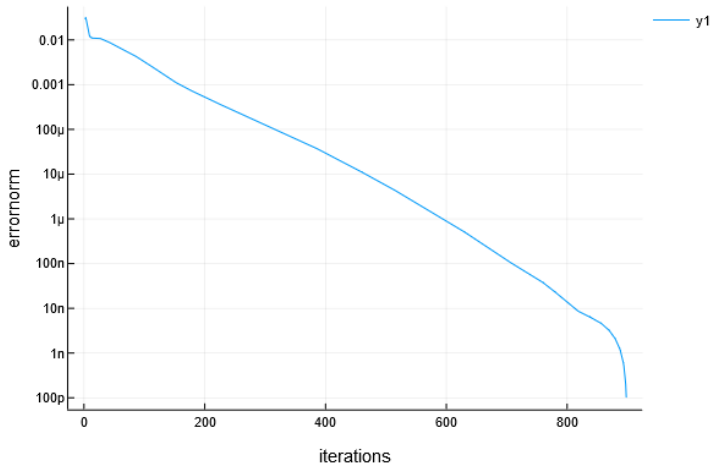
In this assignment we only compare the original solution with $\beta 3$ and $\beta 1$. We define the term errornorm $= \left\|x^k - x^\star\right\|$ and plot it for all iterations:



(a) Errornorm over iterations for proximal gradient without acceleration.



(b) Errornorm over iterations for for proximal gradient with acceleration using $\beta 1$.



(c) Errornorm over iterations for for proximal gradient with acceleration using $\beta 3$ and $\mu = 99$.

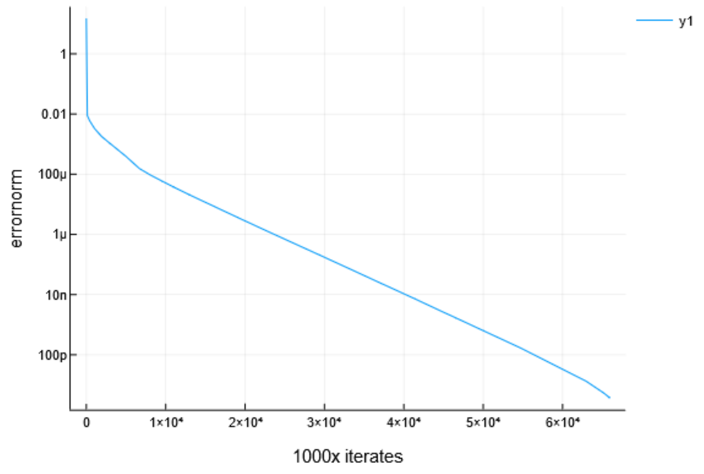
As we can see, the most inefficient method by far is the old method without acceleration. Also clear is that $\beta 3$ is much more efficient than $\beta 1$. The plots for the old method and the new one with $\beta 3$ are very similar in appearance, with the number of iterations being the big difference. On the other hand, by using $\beta 2$ we get a very distinct graph with errornorms oscillating as the method progresses. We can also see that the oscillations get bigger with higher iterations. This method is dependent on which iteration number k we are on as seen in equation 1. As k goes to infinity, $\beta 1$ approaches one and as a result, equation 2 will give more weight to the term x^{k-1} thus being more and more dependent on where it was on the last step and oscillating more.

Task 2

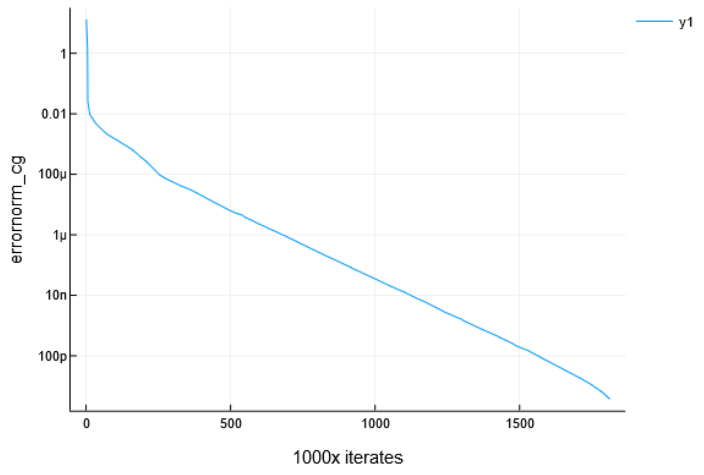
For the SVM problem from the previous hand in, implement the coordinate proximal gradient method. Use the hyper-parameters that gave the best performance in the previous hand-in. Compare the coordinate method to the ordinary and accelerated proximal gradient methods from Task 1. Use the same initial point x_0 for all methods and use the same algorithm-parameters for the ordinary and accelerated proximal gradient as in Task 1. For coordinate gradient, compare both the coordinate-wise step-size of $\gamma_i = \frac{1}{Q_{ii}}$ and the uniform choice $\gamma_i = \frac{1}{L}$ from ordinary gradient descent. Q_{ii} is here the i:th diagonal element of the Q matrix of the function f. Plot $\|x^k - x^\star\|$

Which method required the least amount of computational effort? Which method required the least amount of iterations? Which was fastest in real time? Can you comment on the similarities/differences between real time performance and number of iterations needed? How fair is it to compare real time performance? Can it be easily affected?

As described above, the coordinate proximal gradient method was implemented and used with different step-sizes and compared to the ordinary and accelerated proximal gradient methods. To measure convergence, the errornorm was measured as errornorm $= \|x^k - x^\star\|$ where x^k is the iterates from the coordinate-wise method and $x^\star$ is the solution from the standard one. This resulted in the following figures and tables:



(a) Errornorm over iterations for coordinatewise proximal gradient with $\gamma = 1/L$



(b) Errornorm over iterations for for coordinatewise proximal gradient with $\gamma_i = 1/Q_{ii}$

Method	iterations	time [s]
coord-wise: $\gamma = 1/L$	32990201	130.106376
coord-wise: $\gamma = 1/Q_{ii}$	906745	4.041286
standard PGM	65321	6.026112
accelerated PGM: $\beta 1$	16291	1.432103
accelerated PGM: $\beta 3$	905	0.072532

Table 1: Comparing iterations and execution times for different methods

As we can see from the table, the absolutely most efficient method is the accelerated PGM with $\beta 3$. This makes it preferable to use, but it also demands an extra input parameter μ that might be hard to tune or calculate. Other than that we can see that while the number of iterations greatly varies between the methods, the execution times are pretty close for some of them. Whichever is faster will probably be largely dependent on the effectiveness of our implementations.

If we look at the first graph, we have a very logical behaviour of the coordinate wise PGM: first, the errornorm of the randomised starting point will decrease rapidly with each coordinate that gets adjusted for the first number of times. Then the decrease slows down, but steadily improves until finally the method converges.

The second graph is similar to the previous one, with the most apparent difference being the big difference in number of iterations. Also noticeable is that this graph does not give the same impression of a steady exponential decrease (linear decrease in the plotted log scale) as the previous one. The reason for both of these is the change in step-size. It varies, leading to different rates of decrease. Also, it is much better fitted to each single coordinate-wise step than a uniform step size which results in a much faster convergence.

Compared to the standard proximal gradient method in the original graph 1a we can say that while the appearance of the graphs are quite similar (up to the point where convergence starts to happen in the original one), the graphs for the coordinate proximal gradient method both take many more iterations to converge. This is expected, since one iteration now only changes one coordinate and not all of them. However, when studying table 4 the execution time for the coordinate-wise proximal gradient method with adaptive step size is actually a little faster than the original one and the one with non adaptive step size is much larger. Instead of comparing execution times between the methods another comparison can be made: since we have 500 coordinates in each solution we can approximate that the coordinate-wise proximal gradient method (with constant γ) should take about 500 iterations more than the standard proximal gradient method. When comparing the iteration numbers, we find that if we multiply them for the standard one with 500 we get $500 \cdot 65321 = 32660500$ which is quite close to the number of iterations for the coordinate-wise one (with non-changing γ) of 32990201. Iteration-wise, the methods are equally fast.

However, as we can see in the table, they are not equally fast in time, probably due to the aforementioned implementation differences. In theory, if one could implement a perfect coordinate proximal gradient method, they should be equally fast also time-wise, but this might be hard to do in reality. Instead, with adaptive step-size where the number of iterations needed is much lower, the time needed for convergence was about level to that of the original method.

Hence the adaptive step size coordinate-wise method is probably to be preferred among the mentioned three methods (coord-wise with adapt. γ, coord-wise and standard PGM).

Task 3

We now want to train the network to approximate some function f. To make sure that everything works as expected, run the Task 3 code in backprop.jl to make sure that the pre-defined network can approximate the function $f(x) = \min\left(\|x\|_2^2, 3\right)$. Training over all the data 100 times should result in an average loss of less than 0.001. Plot the result and make sure it looks reasonable. Compare average loss on training data and the test data, how do they compare? If you continue training, how does these values change? With 2000 data-points and a decently optimised network, one pass over the data-set takes roughly 0.2s on a 3.6GHz processor and allocates less than 20MiB

Firstly we plot the function values produced by $f(x) = \min\left(\|x\|_2^2, 3\right)$ in the interval -4 to 4 (which is the same range as our ADAM model has been trained on) along with the values produced by the model in the same range after the model has been trained 100 times. The results are illustrated in 3

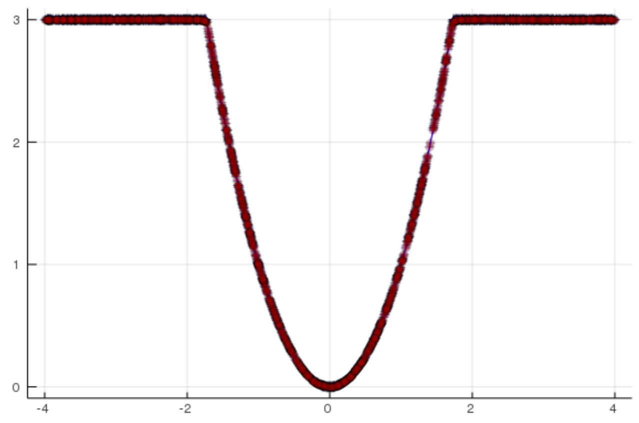


Figure 3: ADAM model trained for 100 training iterations on the interval -4 to 4, projected on the interval -4 to 4

Here it is clear that the model follows the function very well, which is also what is to be expected on the interval in which the model is trained.

We then check how the loss for the training data and test data changes depending on how many iterations we train the ADAM model.

Training iterations	Average loss train	Average loss test
100	1.1779499051754562e-4	1.234785944480224e-4
200	3.4054058532077513e-6	3.4824253402634653e-6
500	2.1603953944837563e-6	2.2486696713372876e-6

Table 2: Average losses for training and test data respectively for our ADAM model

Here we can see that the longer we train the model the smaller the average loss, which is also what is to be expected. However the increase in iterations from 200 to 500 only lowers the average loss with about $1e - 6$ for both the training data and the test data. This leads us to believe that the model has more or less converged and training it further will not decrease the average loss.

We can then check how well the model performs outside of the interval in which it has been trained. We do this by plotting the function f in the interval -8 to 8 as well as the values produced by the model. Results are shown in 4

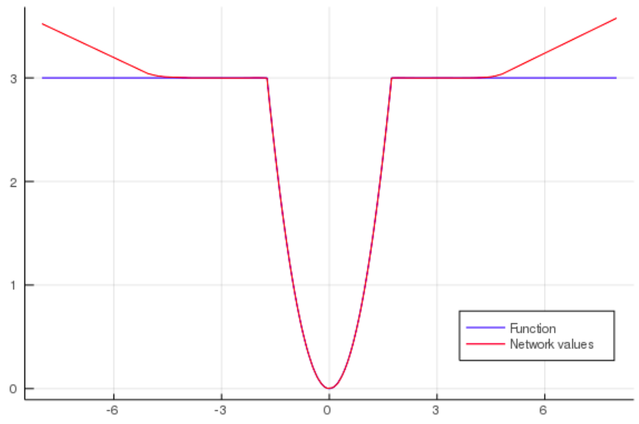


Figure 4: Adam model trained 100 training iterations on the interval -4 to 4, projected on the interval -8 to 8

Here we can see that the interval from -4 to 4 in which the model has been trained the model performs very well. However outside of the interval the model appears to "believe" that the characteristics of the quadratic parts lives on even outside the interval. It projects the information it knows about how the model looks like within the interval to outside the interval and therefore we see these "horns" from -8 to -4 and 4 to 8.

Task 4

Add noise to the training data using the following line

ys = [fsol(xi).+ 0.1.*randn(1) for xi in xs]

and retrain the network. Remember to redefine the layers, network and ADAMTrainer so that you start from scratch. What error do you get on the training vs test data now? Explain the differences and what they mean.

Firstly we check out the function and the ADAM model looks inside the interval -4 to 4.

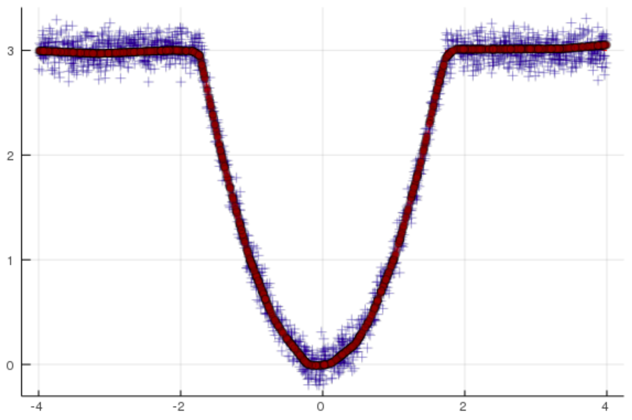


Figure 5: ADAM model trained for 100 training iterations on noisy data on the interval -4 to 4, projected on the interval -4 to 4

Here we can see that the model approximates the function by taking an average of the points. This is clear by looking in 5 where the red line, representing the model output, is drawn as a average over the noisy data.

This means that the the model approximates the training data worse than it did before but since the average is close to the original function (which we also use as test data) it approximates the test set well. This is clear when looking at the average loss for the training and test data sets:

Training iterations	Average loss train	Average loss test
100	1.0393679060162378e-2	5.157088918645537e-4
200	1.0364599137060906e-2	5.353247280348141e-4
500	1.0275568583830278e-2	5.642885776222946e-4

Table 3: Average losses for training and test data respectively for our ADAM model, trained on noisy data

We can see that the model approximates the train data poorly but the test data quite well, which is due to the fact that both the model and the test data is effectively a mean over the train data hence the bigger resemblance between the model and test data compared to the model and train data. Interestingly we don't see the phenomenon here which we did in the previous task where the average loss gets lower when number of training iterations is increased. This is probably due to that the training data is quite rough so the model can only extract so much information from it.

If we plot the model on the interval -8 to 8 as we did in the previous task one can see that the model is a bit worse when trained on noisy data:

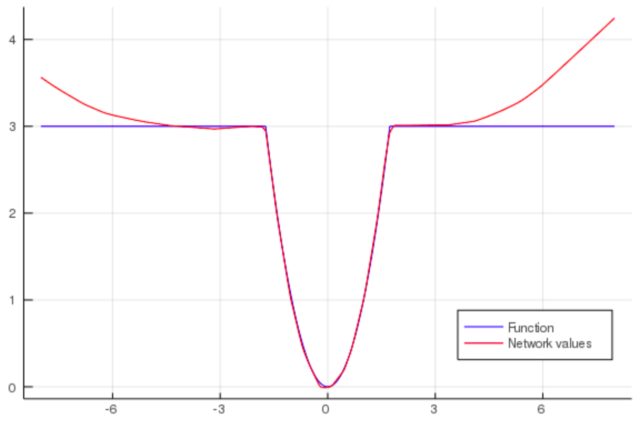


Figure 6: Adam model trained 100 training iterations on noisy data on the interval -4 to 4, projected on the interval -8 to 8

The "horns" appear to be a bit steeper and also they show a tendency to grow quadratically rather than linearly within the intervals -8 to -4 and 4 to 8 (contrary to the behaviour shown in task 3). An intuitive explanation to this might be that when the data isn't noisy the model approximate a function that goes straight through each data point, producing a line. But when the data is noisy it produces a function that weaves more in between the data points, producing more of a quadratic behaviour rather than linear. And when the model tries to guess how the function looks like outside of the training set the quadratic behaviour is accounted for.

Task 5

Decrease the number of points in the training data from 2000 to 30, and redo Task 4. Since we have less data you have increase the number of times you train over the data significantly. How does the results change compared to Task 4?

Going down from 2000 training points to 30 means that we can increase the number of training iterations and still train the model within the same time as when using 2000 data points, so for this task we use 10000 training iterations instead of 100.

Firstly we plot the training points together with the function and model outputs.

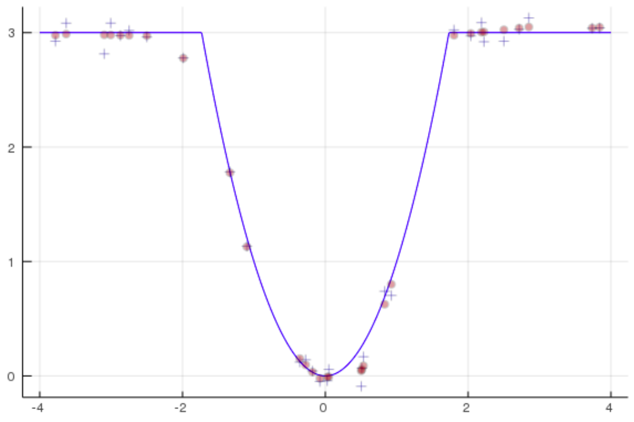


Figure 7: ADAM model trained for 1000 training iterations on noisy data on the interval -4 to 4, projected on the interval -4 to 4 and using only 30 training points (blue crosses represent training data, red dots model output and the blue line the function f)

Here we can see that the model has a much harder task trying to recreate the function f using only the 30 noisy training points.

We are also interested in the average losses for training and test data:

Training iterations	Average loss train	Average loss test
5000	8.48039729011454e-3	5.909065193508755e-3
10000	4.721395181450446e-3	1.059682015755587e-2

Table 4: Average losses for training and test data respectively for our ADAM model, trained on 30 noisy training points

Here we can see that for 5000 iterations the model appears to fit the training data and test data with approximately the same loss. However when using

10000 training iterations the model performs better on the training set than the test set with a factor 10. This indicates that the model has been overfitted for the training data.

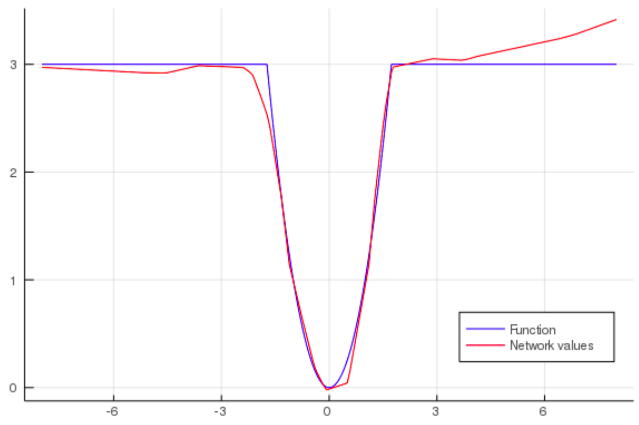


Figure 8: Adam model trained 10000 training iterations on 30 noisy data points on the interval -4 to 4, projected on the interval -8 to 8

When looking at the model projected onto the interval -8 to 8 we see that it behaves more sporadically than in the other tasks. This is probably due to that we now use way less information on which the model can base its outputs. However looking at the results one could argue that they are better than the results using more data points.

Task 6

Change back to 2000 data-points in the training set. We now want to approximate $f(x) = \min\left(0.5\sin\left(\|x\|_2^2\right), 0.5\right)$ with $x \in \mathbb{R}^2$ Modify the network so that the first layer has 2 inputs and update the data generation accordingly. We again want to train the network with 2000 data-points uniformly in [-4, 4] x [-4,4], without noise. What is the lowest error you are able to get without changing the network structure? Report the number of iterations you used. What happens if you resume the training with a lower learning rate $\gamma = 10^{-5}$ instead of 10^{-4}? Why?

The average loss for training and testing data sets for a given number of training iterations and for different gamma-values are illustrated in table 6.

Training iterations	γ	Average loss train	Average loss test
100	10^{-4}	0.06843078390746409	0.05344274321680003
200	10^{-4}	0.01657401672260799	0.010897716001133387
500	10^{-4}	0.014272278112105923	0.005811570734538932
600	10^{-5}	0.009866931600459709	0.0016426569573715082
700	10^{-5}	0.010055718597120102	0.001954026386771292

Table 5: Average losses for training and test data respectively for our ADAM model for different training iterations and γ-values

Here we can see that from 100 to 200 iterations the average loss is improved but from 200 to 500 the gain is marginal. However when we change γ and train another 600 iterations we can see that the loss is improved further. Going from 600 to 700 we cannot see any further improvement.

This behaviour is probably due to that when having a step size that is too large we overshoot the minima each iteration. So by first having a larger step size we get faster convergence but in order to fine tune our solution we need to use a smaller step size to get as close as possible to the minima.

We can also look at a graphical representation of the model outputs and function outputs 9.

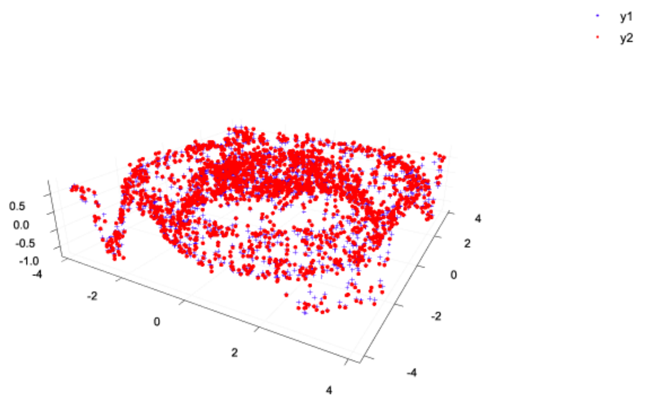


Figure 9: The model outputs (blue) and function outputs (red) after 700 training iterations on the LeakyReLu network

Here the model outputs seem to follow the desired function correctly.

Task 7

Reset the learning rate to 10^{-4}. Make a quick try to redo the training with ReLu instead of LeakyReLu as activation functions. What happens? Give some plausible explanation.

In theory using ReLu as opposed to LeakyReLu means that some of the neurons in the network may become inactivated due to that the backwards gradient is zero if the input value is negative. This means that the output from that neuron to the layers below will always be zero and thus it doesn't help with the calculations. However this should in theory speed up the calculations since less of the neurons are activated, but the downside is that you might have only a half-functioning network.

Training iterations	γ	Average loss train	Average loss test
100	10^{-4}	0.25281060984968007	0.2299525550716516
200	10^{-4}	0.25318336322687485	0.23066949332771056
300	10^{-4}	0.2527813539389329	0.23003385202991264
500	10^{-4}	0.25291147709072226	0.23019955518118
1000	10^{-4}	0.10091096335744408	0.09280476129845472

Table 6: Average losses for training and test data respectively for our ADAM model for different training iterations and γ-values, using ReLu

What we can see from our results is that it takes far more iterations for the ReLu network to converge to a solution. This might be because we have many dead neurons in our network so that we only use a portion of our available nodes, meaning that we have to do more iterations i.e more passes through the network to get out as much information as we would get when using LeakyReLu.

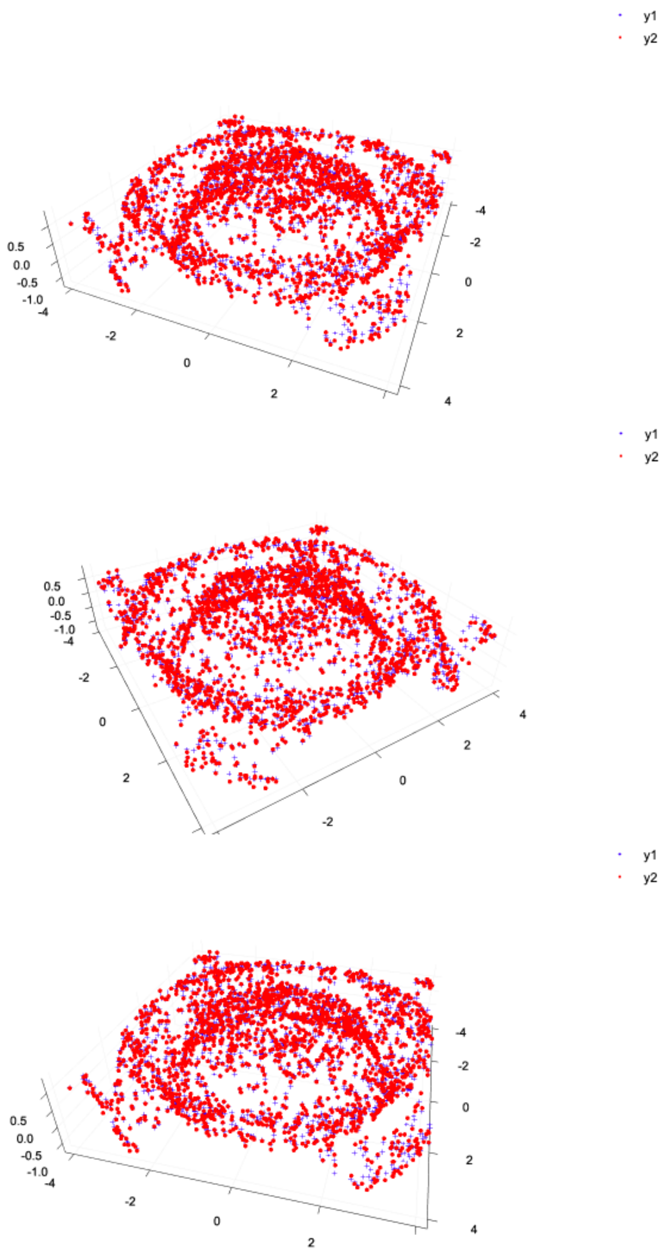


Figure 10: The model outputs (blue) and function outputs (red) after 1000 training iterations on the ReLu network

When looking at the graphs in 10 we can see that the ReLu networks outputs are similar to the LeakyReLu network output (9). However as we established when looking at the average losses in 6 we see that the LeakyReLu performs better in our case.