FYS-STK4155, Autumn 2019, Project 2

Aksel Kristofer Gravir (akselkg)

November 13, 2019

Introduction

The project code can be found at my github: `https://github.com/Akselkg/FYS-STK4155-2019/Project2`

The main goal of the project is to create an implementation of a multilayer perceptron neural network and compare it to "ordinary" regression methods for both regression and classification problems. In this case, OLS and logistic regression respectively. I used the Wisconsin Cancer data as the logistic regression problem, and the FrankeFunction from Project 1 as the regression problem.

The project source code is split into three main python files. neuralnetwork.py implements the class NeuralNetwork, an object oriented implementation of the neural network based on the implementation in the lecture notes. This class is used in cancer_logreg.py and franke_regression.py, each setting up the relevant data structures, runs the regression and the neural networks.

a)

The breast cancer dataset contains 569 instances of 30 attributes and one binary classifier. The attributes are real positive values generated from an image of a breast tissue sample, and the classifier asserts whether the sample is a malignant tumor or not.To process the data I split the data set into training and test data, then the data is scaled using sklearn's StandardScaler. I found this scaling to be neccesary for my gradients not to blow up.

b)

I implemented standard gradient descent to solve the Logistic regression problem. The regression parameters β are initialized with random values drawn from a normal distribution, then iteratively changed by the gradient of the problem iteration. The gradient represents the direction of greatest improvement in β. The size of the change is the gradient scaled by a parameter η, corresponding to the learning rate. Too small of an η will lead to the iteration not converging in time, and high values of η can lead to instabilities. I landed on a value of $\eta = 0.01$ after testing both higher and lower values. This value does not appear to converge totally in 100000 iterations as can be seen from the cost function values as the scheme progresses, and the norm of the gradient, which remains at 0.001. In 1000000 iterations the gradient norm is 0.0001, ten times smaller.

I then set up sklearns default logistic regression for the same data(first in my code). My implementation got an accuracy score of 0.944 using $\eta = 0.01$ and 100000 iterations. sklearn gets a score of 0.958, slightly higher.

c)

My neural network code is structured fairly similarly to the code in the lecture slides, but implementing custom hidden layers and activation functions. I still used the sigmoid function as my activation function, which seemed to perform well for me. With my choice of weights I got an average accuracy score of 0.966, a bit higher than the logistic regression scores. This score is even higher than the score i got running sklearn's MLPClassifier, averaging at 0.961, which surprised me! I averaged the scores over 50 runs since i noticed the scores differing from run to run by 0.03. In total the neural networks seems to perform better than logistic regression on this data set.

As for my parameters I chose fairly simple values of one hidden layer of 50 nodes, 100 epochs of iterations with batch size of 100, a learning rate $\eta = 0.1$ and a regularization parameter $\lambda = 0.01$. My biggest problem in evaluating these parameters was that very many choices of them seemed to run very well, giving roughly the same scores. This I feel is an inherent problem with this data set. Since the accuracy rating already is so high, one changed prediction leads to a major change in the accuracy score due to the relatively small size of the (test) set. Using more hidden layers worked well up to four hidden layers, where i started getting meaningless results. One layer did seem to perform slightly better than two and three. For the number of epochs I got a decent result of 0.94 with only one epoch, slowly improving with more. As long as the total number of iterations including mini-batches are at least three or so, I got meaningful results with the other parameters as above. Using a lower learning rate however would increase the need for more epochs. As an example using $\eta = 0.001$ and five epochs I got an accuracy of 0.85. I got seemingly good results with and without using the λ parameter.

d)

For the regression problem I based myself on code from the first project for OLS regression on the Franke function. However I split the data set into a training and test set in the same way I do for the neural network regression. The wellness of the fit is in this case evaluated by the mean square errors of the fit and the function value, but also by plotting the resulting fits.

For my neural network I did try using the sigmoid function, which worked in some sense of the word. The resulting plot had the main "hump" present but was 0 elsewhere. I changed to the ReLU function, which worked quite well! I also tried a leaky ReLU variant which seemed to be roughly equal to the ReLU.

As my design matrix I used the same one as for the OLS $[1, x, y, xy, x^2, ..., y^5]$. This is questionable, it could perhaps be seen as guiding the network into a polynomial solution? It did work quite well though. Parameters were decided as in the classification problem, but i did find it useful here to increase the number of epochs and lower the training rate. Additionally I was unable to get reasonable results with more than one hidden layer. I could not tell if this was due to a bug in my code or inherent to the problem. Possibly more parametres needed adjusting. The parametres and the resulting function is plotted in Fig. 2.

Comparing the plots visually there are some notable differences. The OLS plot is much smoother than the NN plot, in that sense like the original function. I would however say the NN captures the smaller bump better, and in general has a more alike shape. The result of the OLS seems to skew a bit. This is reflected in the cost functions: MSE= 0.00234 for OLS, MSE= 0.000848 for NN. With an added error $\epsilon = 0.1$: MSE= 0.0131 for OLS, MSE= 0.0120 for NN. Overall better, not incredibly much so in the case with the added error. The plots look similar with the error added.

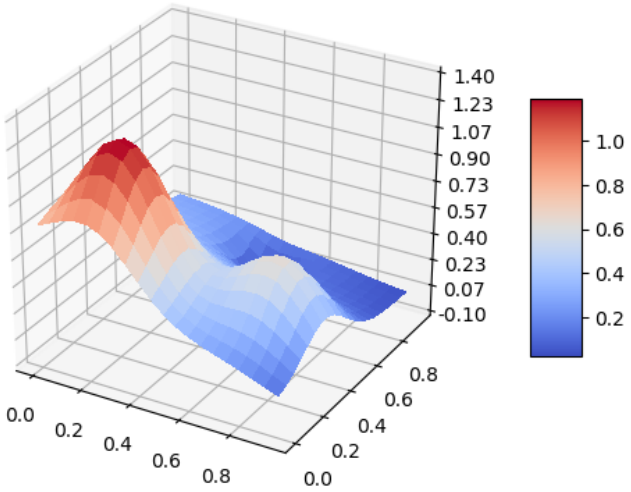

Figure 1: Franke function on $[0, 1] \times [0, 1]$

Conclusion

Overall, all my implemented NN ran better than the respective regression algoritms. The differences were not major, but seemed significant. The datasets used here were also quite small, which in theory should favour the non-NN algorithms. That being said I did not quite get a sense of scale in how expensive the algorithms were to run. For the logistic regression i stopped my gradient descent at an arbitrary point since i could not get a good convergence. For the regression on the Franke function the OLS is certainly much lighter to run.

For the Franke function, the MSE favor my NN, but the qualitative differences could go either way depending on whether how smooth you want the resulting curve. I still prefer the NN.

It would have been interesting to look at other data sets in both cases to get a feeling for how much the results were shaped by the data.

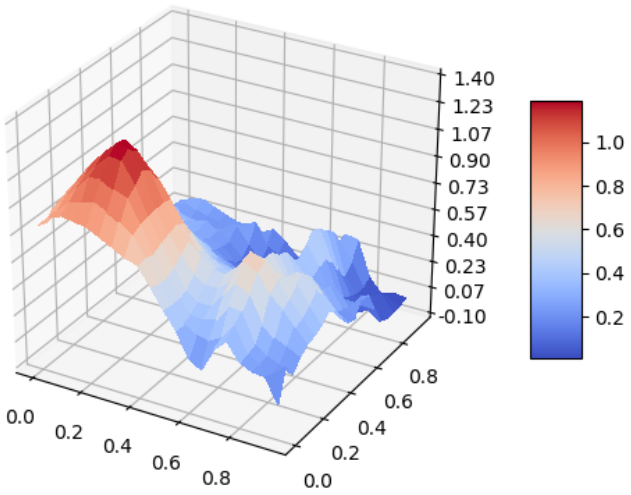

Figure 2: Approximate Franke function from implemented neural network. $n = 40 \times 40$, $\epsilon = 0$ One hidden layer of 100 neurons, 1000 epochs, batch size 100, $\eta = 0.001$, $\lambda = 0.0001$

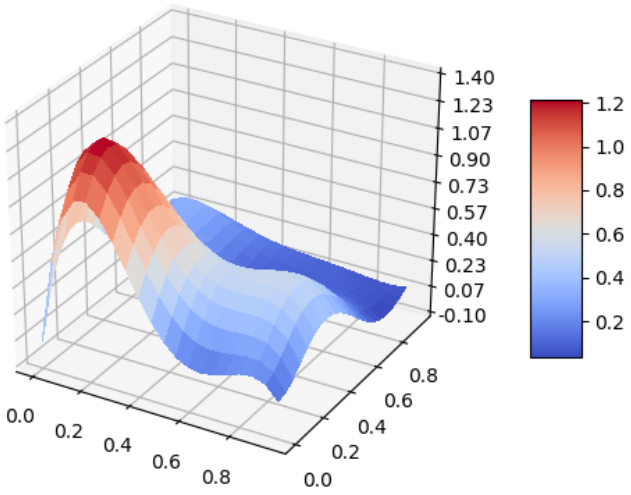

Figure 3: Approximate Franke function from OLS. $n = 40 \times 40$, $\epsilon = 0$.