ML–DS I: Project IV Report

Sebastijan Trojer
ML-DS I 2024/25 , FRI, UL
st5804@student.uni-lj.si

I. Introduction

In this project we implemented fully connected neural network for regression and classification, with arbitrary number of hidden layers and arbitrary activation functions for each one.

II. Methodology and results

A. Classification neural network

We started by implementing a multilayer fully-connected artificial neural network (ANN) for classification. We used the sigmoid activation function between each layer, except the one before the loss function, where we used the softmax, to convert logits to probabilities. The implementation is object oriented, as we defined classes for different substructures, used in ANN. Each such class consists of 3 main methods – the constructor, forward and backward method, which are used for computing values in forward and backward pass. For classification, we were optimizing the categorical cross entropy loss, which has an additional advantage that, when combined with softmax, the derivative simplifies. For weight initialization we used the Xavier weights, and biases were initialized to 1.

Since we have a modular design, implementing backpropagation was simpler, as each object was designed such that it memorized values from the forward pass for that iteration, therefore we didn't need to recompute them or have other data structures for that. We verified the backpropagation is implemented correctly by comparing the gradients with their numerical values. We performed a forward pass, and then for each layer, we compared the gradients with their numerical values, and computed a relative error. We did that on a network with 3 hidden layers of size 5 on the doughnut dataset. The largest relative error we recorded was 4.76×10^{-8} which indicates that our gradients were computed correctly. The numerical gradients were computed using the approximation, where we change the value of the parameters slightly, and observe how the loss changes.

In the experiments we used sigmoid activation function, however we also implemented ReLU and LeakyReLU and allowed for arbitrary activation function selection when creating the classification object. For gradient descend we added an option to use a decaying learning rate, which can also be passed to the constructor of the classifier.

Dataset	Hidden layer size	Iterations	Learning rate
squares	4	3600	1
doughnut	3	3600	1

TABLE I
Smallest network capable of perfectly fitting the data in under 10 000 iterations.

The goal of this task was to develop an ANN that is capable of perfectly fitting the doughnut and squares dataset. The minimal network size and other parameters from the experiments are reported in Table I. For both datasets, a single hidden layer was sufficient, however both performed better with a high learning rate. Both also worked well with lower learning rates, however it was helpful to increase the weights at initialization. Both networks were really small, which enabled fast convergence, as seen on Figure 1.

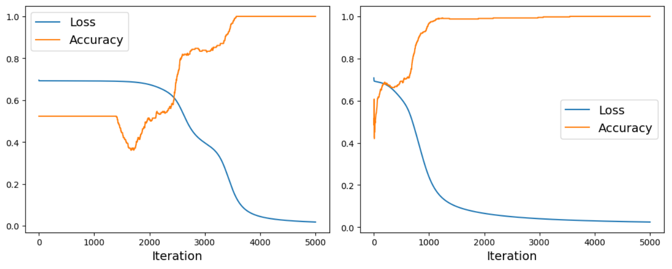

Fig. 1. Loss and accuracy on doughnut (left) and squares (right) for first 5000 iterations. Notice that the loss keeps dropping, even after a perfect accuracy is achieved.

B. Regression neural network

Next we implemented the regression ANN. The main difference in architecture is that it uses linear activation function instead of the softmax. Furthermore, the last layer always only has a single neuron and it uses the mean squared error loss instead of cross entropy. Other than that the structure is very similar to a classification network, thus we simply inherited the class and did some minor adjustments, such as specifying a different loss function, adding a single neuron to the last layer and returning raw values from network rather than softmax probabilities.

Initially we tested the model by sampling 5000 samples from a normal distribution for the target variable to see if the model fits well and adapts to the noise. Figure 2 shows the data and the fitted line, as well as the histogram of the predictions. Notably the model predicted the mean consistently, which indicates that it is working correctly.

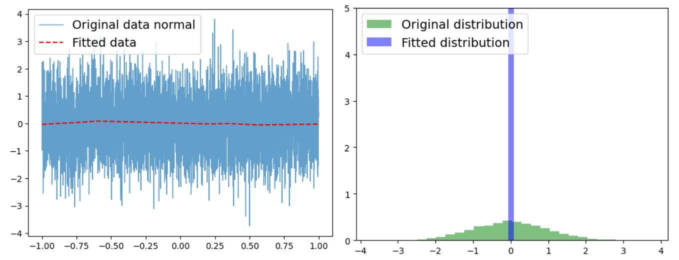

Fig. 2. Line fitted on data with a normally distributed target variable. The left image shows the data and the fitted line, and the histogram shows the prediction density with of both the data and predictions on the right.

We also tested if the model is capable of fitting to some functions, so we created exponential data and fitted the model. It turned out that it was able to fit the line well, so we also tried fitting it on sinusoidal data, where it failed at first, but as we increased the sample size and added some more noise, the model successfully fit the data relatively well, as seen on Figure 3.

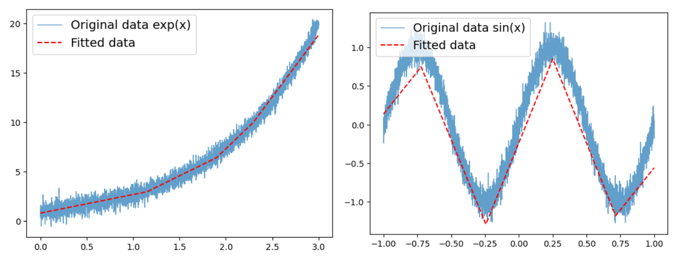

Fig. 3. Line fitted to exponential (left) and sine (right) data. The model fitted the exponential function a lot faster and better, while the fit on the sine data is not as good. Both models had a single hidden layer with 10 neurons and leaky ReLU activations.

C. Regularization

Next, we implemented support for regularization. To do so, we created an abstract class Loss, from which all the other loss types could inherit the methods regularize bias and regularize weights, although we did not actually regularize the biases. We implemented the L2 regularization and added the regularization loss to the cost function loss in the fit method. Since the regularization was a part of the loss objects, no changes were required for the ANN classes, except passing the λ parameter.

D. PyTorch comparison

We performed a comparison of our implementation to a PyTorch implementation. For classification we built a model with a single hidden layer of size 3, used sigmoid activation and cross entropy loss. We performed 10-fold cross validation on both datasets and results are reported in Table II. Both implementations performed more or less the same, the PyTorch implementation had slightly better results when using a lower learning rate.

Additionally we also compared the gradients of the models after 10 iterations by computing cosine similarities per layer. We found that the gradients of the first two layers were very unsimilar, however the weights and biases on the output layer matched very closely, with cosine similarity over 0.95.

Dataset	Iterations	Learning rate	Loss (ours)	Loss (PyTorch)
doughnut	1000	1	0.6977	0.6975
doughnut	1000	0.001	0.7352	0.6948
squares	1000	1	0.6937	0.6951
squares	1000	0.001	0.7343	0.7112

TABLE II
10-FOLD CROSS VALIDATION PERFORMANCE ON EACH DATASET BY OUR ANN AND PYTORCH IMPLEMENTATION.

E. Competition on the FTIR spectral data

For the competition we followed the example with the logistic regression, and used our implementation of the ANN. We used a model with a single hidden layer with 100 neurons with sigmoid activation and learning rate 1, with decay of 1×10^{-3} every 100 iterations. We achieved a minimal loss of 0.56204 on the leaderboard. We also tested the same model with different number of layers, learning rates etc. and also sklearn implementations, however most of the other models overfit very quickly and had very poor performance on the test set. We only used the spatial information when training.

III. Conclusion

In conclusion, we implemented classification and regression ANNs and compared it with PyTorch implementation of the same architecture, and got very similar performance. We tested the gradients by comparing them to the numerical gradient approximations and confirmed proper computation. We developed a framework for using ANNs with arbitrary hidden layer sizes and numbers, arbitrary set of activation functions and loss functions, and with regularization support. We also utilized our own model to achieve a better model than the logistic regression for the FTIR competition.