Report
Project 1
Methods of Deep Learning

Michał Bortkiewicz
Jeremiasz Wołosiuk

March 2020

Contents

1 Goal of the project

First project of the course Methods of Deep Learning consisted of implementation of multilayer perceptron (MLP) artifical neural network (ANN) with usage of basic python libraries such as numpy and pandas. The main goal of the project was to deepen understanding of how simple ANN works by developing modular solution. As a result we implemented class NeuralNetwork with subclass Dense that incorporate majority of basic features for MLP networks which can be found in popular frameworks including tunable number of hidden layers, neurons, epochs etc. In addition we designed visualization that explains model training (bacpropagation algorithm) and weights obtaining.

2 Multilayer perceptron artificial neural network

Multilayer perceptron artificial neural network refer to networks composed of multiple layers of perceptrons (with threshold activation), sometimes referred to as "vanilla" NN.

2.1 Structure

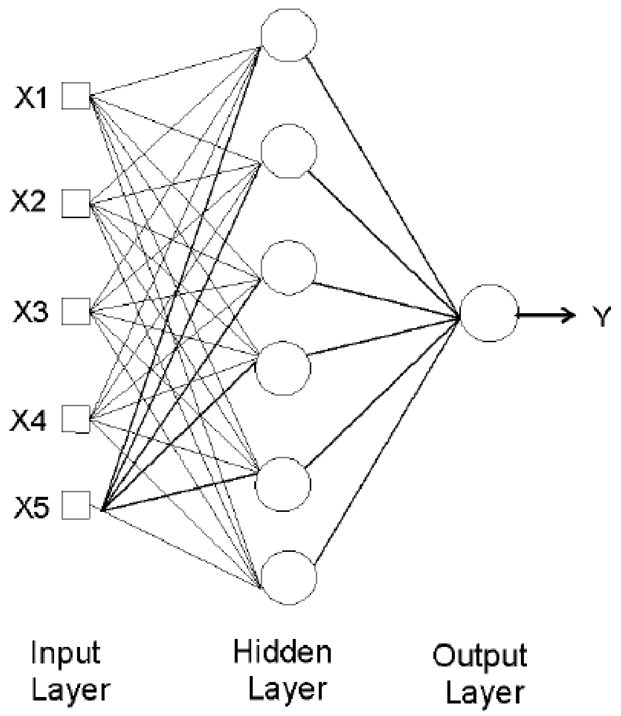

Figure 1: MLP neural network with one hidden layer.

An MLP consists of at least three layers of nodes (Figure 5): an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. For training MLP utilizes a supervised learning technique called backpropagation.

3 Implementation

We concluded that solution should be written with compliance with the OOP rules; therefore two specific objects were determined:

- NeuralNetwork object that consists of many layers objects and other features,
- Dense object which is a dense MLP neural network layer.

Having proposed structure of the project we foresee that adding another different type of layer will not pose a problem.

Forward pass and backpropagation works using Python dict as cache for transfering data between layers.

Implementation was partially based on `https://github.com/SkalskiP/ILearnDeepLearning.py/blob/master/01_mysteries_of_neural_networks/03_numpy_neural_net/Numpy%20deep%20neural%20network.ipynb`.

4 Trial and Error

4.1 First classification results

During first testing and debugging sessions having implemented majority of classification features we conducted some trials with provided and external data sets. Our main goal during these sessions was to check if back-propagation algorithm works as intended.

Provided data set - data.three$_{g}auss.train.500.csv$

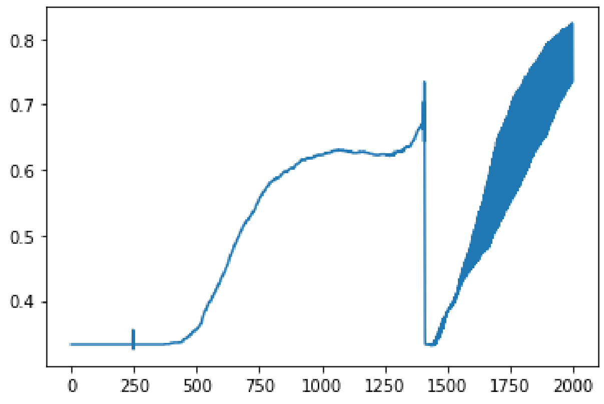

Figure 2: Accuracy per epoch of training.

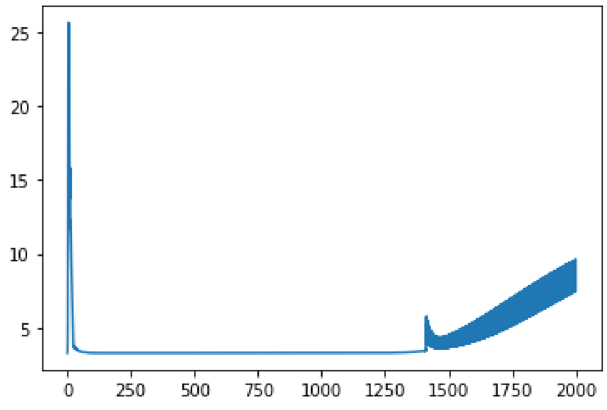

Figure 3: Cost per epoch of training.

During these trails our solution did not have implemented momentum which should probably smooth both accuracy and cost.

External data set - SAheart
Not decreasing cost value over epochs may indicate that chosen hyperparameters are not optimal for specific problem. In upcoming trails we will try find optimal hyperparameters using grid search.

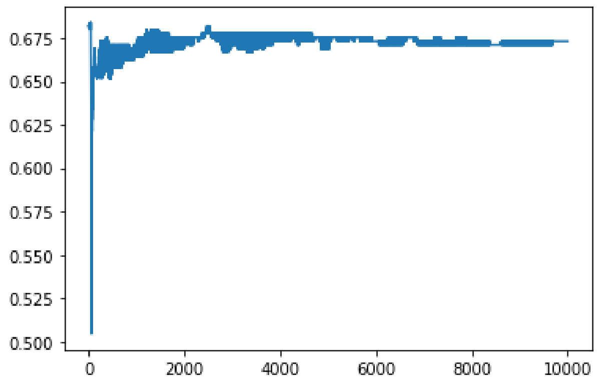

Figure 4: Accuracy per epoch of training.

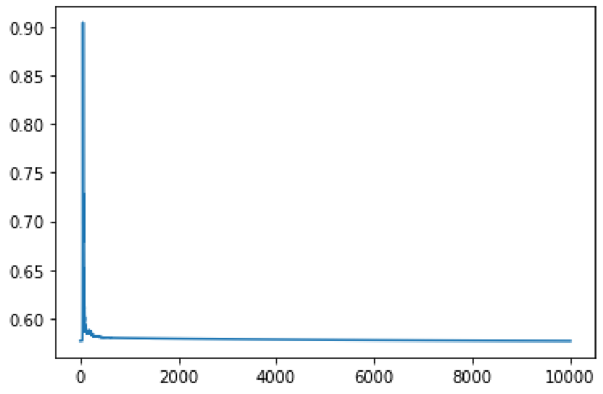

Figure 5: Cost per epoch of training.

5 Description of experiments

5.1 Classification

Experiments were designed as follows. All experiments were conducted to measure cost and accuracy for each epoch during training. After training of several models with same architecture but differently initialized weights we took mean predictions of them and ploted in green colour those points that were predicted properly and in red those wrong predicted ones. The fourth subplot represents accuracy for particular trial. Figure 6 presents exemplary subplot that includes all of the mentioned analysis.

In experiment phase seven different architectures were tested for both of provided datasets. Models hyperparameters were chosen quite randomly to check how different values and different combinations of values change models predictive power. Accordingly to guidelines models differ in:

- number of hidden layers,
- number of neurons per layer,
- activation functions,
- error functions,

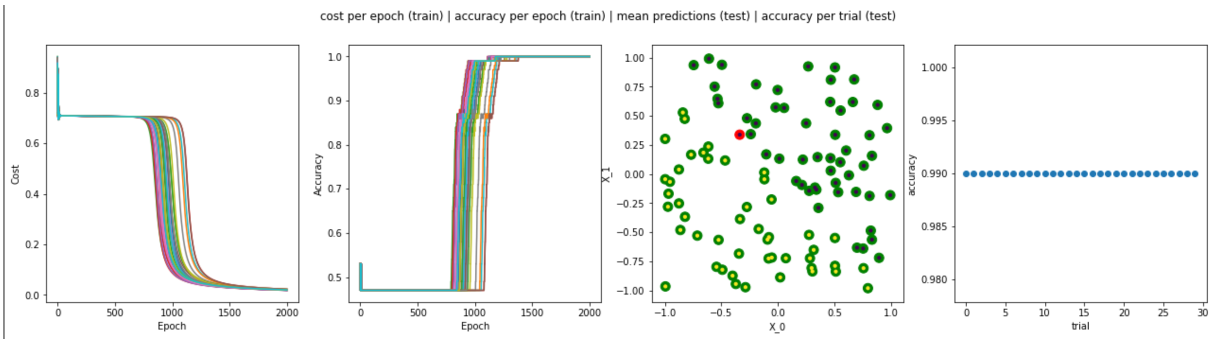

Figure 6: Exemplary subplot with classification results.

- learning rate (alpha) and momentum parameter (beta).

Because of long training time trails were conducted on small datasets.

5.2 Regression

Experiments were designed as follows. All experiments were conducted to measure influence on final R2 score for train set and test set.

Experiment 1 measured influence of:

- number of hidden layers = [0, 1, 2, 3, 4],
- number of neurons per layer = [4, 16, 32, 64],
- activation functions = ['sigmoid', 'linear', 'relu'],
- learning rate (alpha) = [0.3, 0.7, 1],
- nr of epochs = [100, 500, 1000, 2000, 5000].

Constant hiperparameters:

- error function = L2 norm,
- learning rate (beta) = 0.9.

Grid search was used to check every parameter exhaustively. Each set of parameters was tested 10 times, for statistical assurance.

Experiment 2 measured influence of error function. Due to highly time consuming problem, only one architecture was tested, selected as best from previous experiment. Tested error functions: [L1, L2].

Due to high learning time, all experiments were conducted on small data.cube dataset.

5.3 Kaggle MNIST

Experiment idea was to test our implementation against common machine learning problem - MNIST dataset, which solves (or creates) classification problem for hand written digits.
Link: https://www.kaggle.com/c/digit-recognizer/

6 Results

Results were obtained in jupyter notebooks (*classification.ipynb* and *regression_test.ipynb*) which are included in our project repository. They are mostly self descriptive, therefore, we include in this report only few of received results.

6.1 Classification

6.1.1 Dataset - *data.simple*

In dataset *data.simple* there are two input variables and two distinct classes which could be easily split by straight line. Input variables are in range from -1 to 1 to make models convergence faster.

Architecture 3
Model hyperparameters:

- number of hidden layers = 1,
- number of neurons per layer = [2, 1] (first layer is input layer with no trainable weights),
- activation functions = ['relu', 'sigmoid'],
- error functions = 'binary cross entropy',
- learning rate (alpha) and momentum parameter (beta) = 0.5, 0.9.

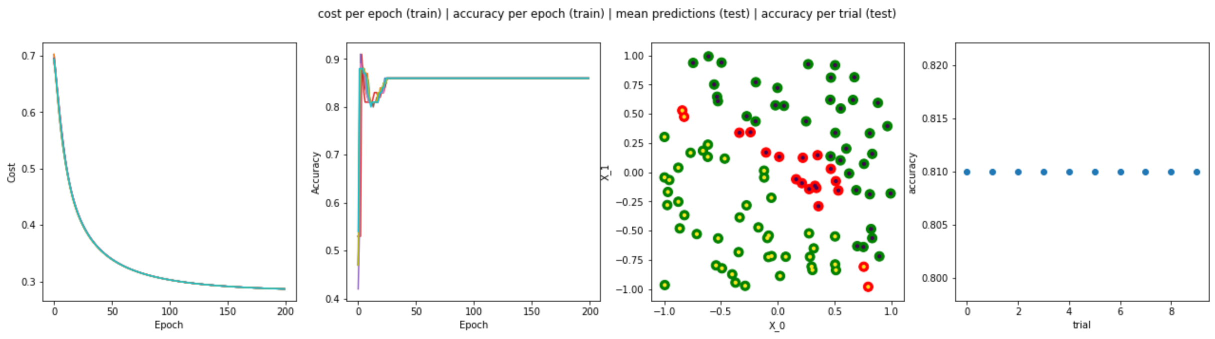

Figure 7: Architecture 3 results for *data.simple*

Architecture 4
Model hyperparameters:

- number of hidden layers = 3,
- number of neurons per layer = [2, 10, 100, 2] (first layer is input layer with no trainable weights),
- activation functions = ['sigmoid', 'sigmoid', 'sigmoid', 'softmax'],
- error functions = 'cross entropy',
- learning rate (alpha) and momentum parameter (beta) = 0.5, 0.9.

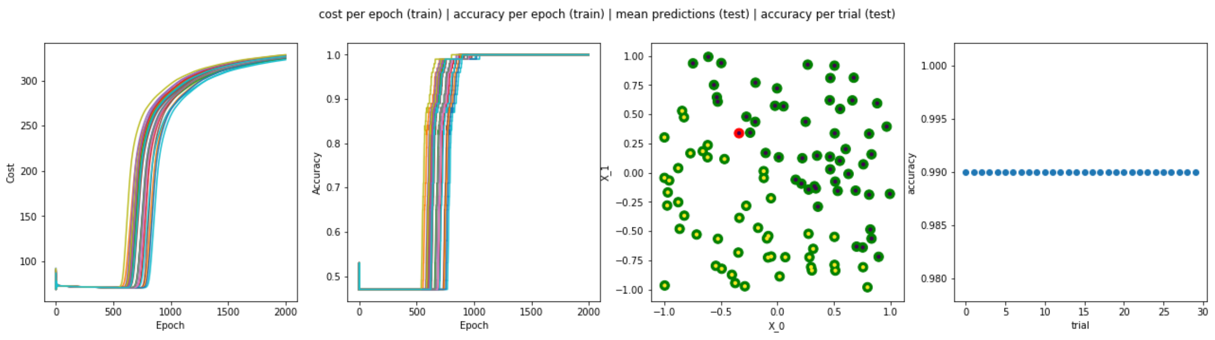

Figure 8: Architecture 4 results for *data.simple*

Architecture 4

Conclusions
As one would expect, model with higher number of hidden layers saturates on higher accuracy but much later than basic model. During experimental phase we observed much more rapid convergence of models with 'relu' activation functions than those with 'sigmoid' activations. Increasing cross entropy cost results probably from implementation error because accuracy is behaving as expected.

6.1.2 Dataset - *data.threegauss*

In dataset *data.threegauss* there are two input variables and three distinct classes with gauss distribution. Input variables are in range from -1 to 1 to make models convergence faster.

Architecture 2
Model hyperparameters:

- number of hidden layers = 2,
- number of neurons per layer = [2, 10, 3] (first layer is input layer with no trainable weights),
- activation functions = ['relu', 'relu', 'softmax'],
- error functions = 'cross entropy',
- learning rate (alpha) and momentum parameter (beta) = 0.1, 0.9.

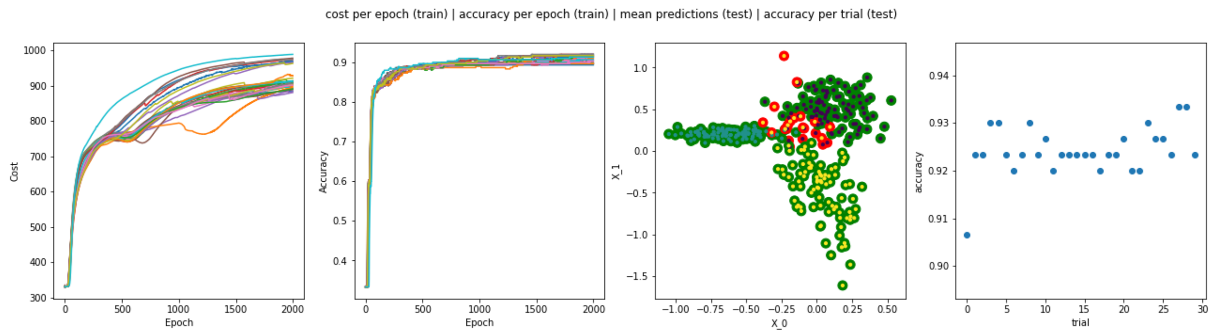

Figure 9: Architecture 2 results for *data.threegauss*

Architecture 4
Model hyperparameters:

- number of hidden layers = 4,
- number of neurons per layer = [2, 10, 10, 10, 3] (first layer is input layer with no trainable weights),
- activation functions = ['sigmoid', 'relu', 'relu', 'relu', 'softmax'],
- error functions = 'cross entropy',
- learning rate (alpha) and momentum parameter (beta) = 0.1, 0.9.

Architecture 4

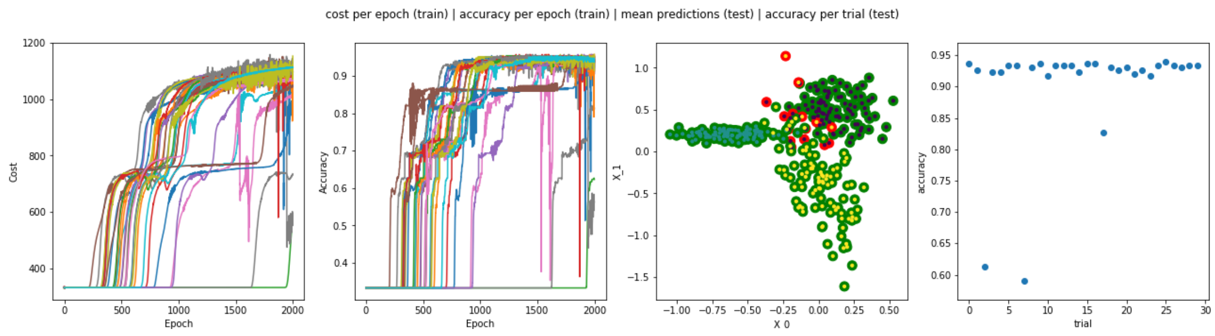

Figure 10: Architecture 4 results for *data.threegauss*

Conclusions
As in previous section we observe better overall results for deeper model, however convergence again takes longer. Moreover, there are at least two distinguishable phases in training for more complex model where model 'learns' another crucial dependencies that result in higher accuracy. Although model with more hidden layers achieved better results, it was more subjected to weight initialization as there are models that did not start to converge even after 1000 epochs.

6.2 Regression

Adequate plots can be seen in "regression_test.ipynb" notebook.

6.2.1 Experiment 1

Best architecture overall was Architecture 1, with R2 test score 0.1915 +/- 0.0007 and training score: 0.909 +/- 0.002.

Architecture 1
Model hyperparameters:

- number of hidden layers = 2,
- number of neurons per layer = [1, 32, 32, 1] (first layer is input layer with no trainable weights),
- activation functions = [relu, relu, relu, linear],

- error function = 'L2',
- nr of epochs = 2000,
- learning rate (alpha) = 1.

During experiments we noticed:

- linear activation function does not gain significantly from increasing number of hidden neurons/layers or number of learning epochs - probably due to implementation error or achieving maximal fitness of linear model,
- high number of hidden neurons (64) and linear activation function causes float overflow - probably due to problem with numerical stability of implementation or implementation error.

6.2.2 Experiment 2

Probably due to implementation problem usage of L1 loss function caused ANN to return always 0.

6.3 Kaggle MNIST

We tried to approach the problem with following architecture:

Architecture Kaggle
Model hyperparameters:

- number of hidden layers = 3,
- number of neurons per layer = [784, 50, 30, 20, 10], (first layer is input layer with no trainable weights),
- activation functions = ['sigmoid', 'relu', 'relu', 'relu', 'softmax'],
- error function = 'cross_entropy',
- nr of epochs = 1200. (one night of learning on our hardware)

Unfortunately, probably due to implementation error or disappearing gradient problem, our architecture always predicted class 1 and achieved very low score. We didn't submit our results on kaggle.com experiment page.

Our experiment can be seen in "kaggle.ipynb" notebook.

7 Learning visualization

Visualization can be seen in "visualization_test.ipynb" notebook.

8 Summary

Our implementation definitely could learn and solve simple classification/regression problems, however large ANN architectures and complex problems were not solvable by the implementation, probably due to problems such as exploding/disappearing gradients, numerical stability of implemented solutions and simply implementations errors.

Project was for sure very educational and if having more time, as the project deserved, found problems could be solved and better results could be achieved. We are planning to do so, as value of the project in portfolio can be very high for future employments.