

Neural Networks from scratch

Fischer Hauke Edgar, Krofitsch Christoph, Miksa Maciej

Core setup: Building and training neural networks

The core setup is based on **python and the numpy library** for mathematics, and two classes

- The **Layer class** provides activation functions as well as forward and backward propagation for a single layer
- The **NN class** allows to create networks (collection of layers), compute the output of the network and to train the network on data

Forward propagation method of a layer and derivatives

Takes the input $X = [x_1, \dots, x_m]$ and returns the output $Y = [y_1, \dots, y_n]$, three activation functions are provided

$$y_i = f\left(\sum_j w_{ji}x_j + b_i\right) = f(\tilde{y}_i)$$

Code

```
def forward_propagation(self, input):
    self.input = input
    self.output = np.dot(input, self.weights) + self.bias # np
    # notation in the lecture where weight wij refers to the i
    if self.activation == 'None':
        self.output = self.output
    elif self.activation == 'sigmoid':
        self.output = Layer.sigmoid(self.output)
    elif self.activation == 'tanh':
        self.output = Layer.Tanh(self.output)
    elif self.activation == 'relu':
        self.output = Layer.relu(self.output)
    else:
        print('Invalid activation function!')
    #self.output = self.output[0] # without this line I get sh
    return self.output
```

$$Y = f(XW + B) = f(\tilde{Y})$$

Backward propagation method of a layer

This method takes as input the derivative of the error w.r.t. to the output of that layer dE / dY and returns dE / dX . It also compute dE / dW and dE / dB and updates the weights and biases

$$\begin{aligned}\frac{\partial E}{\partial \tilde{Y}} &= \left[\frac{\partial E}{\partial Y_1} \frac{\partial Y_1}{\partial \tilde{Y}_1}, \dots, \frac{\partial E}{\partial Y_n} \frac{\partial Y_n}{\partial \tilde{Y}_n} \right] \\ &= \left[\frac{\partial E}{\partial Y_1}, \dots, \frac{\partial E}{\partial Y_n} \right] \circ [f'(\tilde{Y}_1), \dots, f'(\tilde{Y}_n)]\end{aligned}$$

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial \tilde{Y}} \frac{\partial \tilde{Y}}{\partial X} = \frac{\partial E}{\partial \tilde{Y}} W^T$$

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial \tilde{Y}} \frac{\partial \tilde{Y}}{\partial B} = \frac{\partial E}{\partial \tilde{Y}}$$

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k=1}^n \frac{\partial E}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial w_{ij}} = \frac{\partial E}{\partial \tilde{y}_j} x_i$$

$$\frac{\partial E}{\partial W} = X^T \frac{\partial E}{\partial \tilde{Y}}$$

Code

```
def backward_propagation(self, dEdY, learning_rate):
    output = np.dot(self.input, self.weights) + self.bias #
    # First I only back propagate the derivative w.r.t. acti

    if self.activation == 'None':
        dEdY = dEdY
    elif self.activation == 'sigmoid':
        dEdY = Layer.Dsigmoid(output)*dEdY # this is an elem
    elif self.activation == 'tanh':
        dEdY = Layer.DTanh(output)*dEdY # this is an element
    elif self.activation == 'relu':
        dEdY = Layer.Drelu(output)*dEdY # this is an element
    else:
        print('Invalid activation function!')

    # Next I propagate the derivatives w.r.t. to weights mat
    # dEdW is the derivative w.r.t. to the weights / biases

    dEdW = np.dot(np.transpose(self.input), dEdY)
    dEdB = dEdY
    dEdX = np.dot(dEdY, np.transpose(self.weights))

    # update parameters
    # self.weights -= learning_rate * dEdW
    # self.bias -= learning_rate * dEdB
    # code update (Maciej)
    self.weights -= learning_rate * dEdW.astype('float64')
    self.bias -= learning_rate * dEdB.astype('float64')

    return dEdX
```

NN class: Main method: Train

The Train method takes the full data set and iterates over it specified by the number of epochs. For each data instance the backpropagation method is applied to each layer, starting by the last layer. For the last layer the derivatives are computed directly:

$$E = \frac{1}{N} \sum_{j=1}^N (y_{\text{real},j} - y_{\text{predicted},j})^2$$

$$\frac{\partial E}{\partial Y} = \frac{2}{N} (Y_{\text{predicted}} - Y_{\text{real}})$$

Weight initialization

- Initially we tried to take random values between -0.5 and 0.5 to initialize weights and biases
- This often lead to very poor results with the Train method
- To improve our results we implemented **Xavier** (sigmoid and tanh activation function) and **He initialization** (relu activation function)
- These initialization ensure that the **variance of activations is the same across every layer** and hence prevents vanishing or exploding gradients, for both cases biases are initialized to be 0

Xavier

$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{\mathbf{1}}{n^{[l-1]}})$$

$$b^{[l]} = 0$$

He

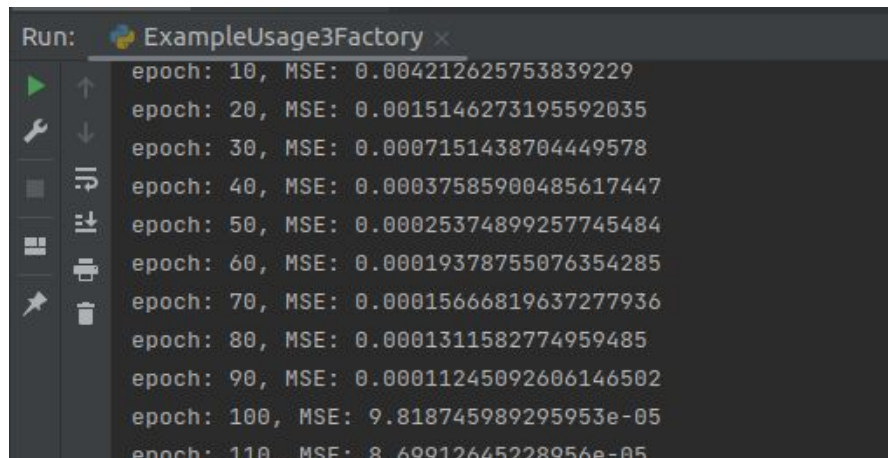
$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{\mathbf{2}}{n^{[l-1]}})$$

$$b^{[l]} = 0$$

Testing the NN with random data

Before applying it on real data sets, we implemented a tests with randomly generated example data. This files is named 'ExampleUsage3Factory.py':

- Generates two 3-dimensional sample classes with 100 samples each
- drawn from two different multivariate Gaussian distribution with 'overlap'
- delivers good results:



```
Run: ExampleUsage3Factory x
epoch: 10, MSE: 0.004212625753839229
epoch: 20, MSE: 0.0015146273195592035
epoch: 30, MSE: 0.0007151438704449578
epoch: 40, MSE: 0.00037585900485617447
epoch: 50, MSE: 0.00025374899257745484
epoch: 60, MSE: 0.00019378755076354285
epoch: 70, MSE: 0.00015666819637277936
epoch: 80, MSE: 0.0001311582774959485
epoch: 90, MSE: 0.00011245092606146502
epoch: 100, MSE: 9.818745989295953e-05
epoch: 110, MSE: 8.69912645228956e-05
```

Automatically finding good values

As per the task description we need to implement such a method.
We chose to implement **Randomized Search**.

Via the file NNFactory.py you can call a function that does the search for you via:

- a range of Epochs
- a range of Layers and their sizes
- a list of output functions
- a range of learning rates
- number of iterations, and further minor params

For each iteration, the functions draws random values for the given ranges (uniformly distributed), and tests a NN on it via **k-fold Cross Validation**

The best resulting NN is returned.

Applying the Randomized Search

Again via the file ExampleUsage3Factory.py.

It also uses the Randomized Search to return the best NN (here not printing the epochs). Here is an example run with k=4 CV and some range of parameters:

```
Run: ExampleUsage3Factory x
/home/christoph/PycharmProjects/EX-02/venv/bin/python /home/christoph/PycharmProjects/EX-02/ExampleUsage3Factory.py
Search Iteration 1: LR=0.05590607171856813, epochs=60, layers=(3in - 2relu - 1sigmoid)
Checking fold 1/4:
Fold MSE result: 0.0028514635255715835
Checking fold 2/4:
Fold MSE result: 0.001199173586642145
Checking fold 3/4:
Fold MSE result: 0.0018394776537528645
Checking fold 4/4:
Fold MSE result: 0.0011881321120452983
Search iteration completed! Result MSE = 0.001769561719502973

Search Iteration 2: LR=0.088710365518863117, epochs=100, layers=(3in - 2relu - 1sigmoid)
Checking fold 1/4:
Fold MSE result: 0.0005875965582184844
Checking fold 2/4:
Fold MSE result: 0.00040526453504053327
Checking fold 3/4:
Fold MSE result: 0.0003763889828675678
Checking fold 4/4:
Fold MSE result: 0.00016439516587003284
Search iteration completed! Result MSE = 0.00038341131049915455

Search Iteration 3: LR=0.10037786878750181, epochs=112, layers=(3in - 2relu - 1sigmoid)
Checking fold 1/4:
```

Working with nominal values

As per the task description our NN needs to handle this.

We chose to implement One-Hot-Encoding for this. The file `NNFactory.py` also provides the function **convertNominalFeatures** that takes

- a reference dataset
- a dataset to encode (needs to have same shape)

The function first **prepares the encoding** via the reference dataset: by finding out what columns contain non-numerical values and finding their distinct values).

Then, it **executes the encoding** on the second dataset, by replacing the according columns with One-Hot encoded columns.

This has been tested and demonstrated in **ExampleUsageNominal.py**.

Real datasets used in the experiment

1. Abalone Dataset

This dataset contains measurements of physical characteristics of abalone shells. The goal is to predict the age of the abalone based on these features.

2. In-Vehicle Coupon Recommendation Dataset

This dataset focuses on in-vehicle coupon recommendation. It includes various attributes related to the customer, the vehicle, and the context. The objective is to predict whether a customer will accept or reject a coupon offer.

3. Congressional Voting Dataset

This dataset focuses on congressional voting records. It contains information about how each member of Congress voted on various issues, represented as binary values (yes or no). The objective is to predict the political party (Republican or Democrat) based on voting patterns.

Datasets comparison

To ensure a comprehensive evaluation of our implementation, we intentionally selected datasets with varying characteristics, such as dimensionality and number of samples. This approach enables us to thoroughly investigate the performance of our implementation across different types of data.

	#samples	#features	#features x #samples
Abalone Age	4177	8	33416
In-Vehicle Coupon	12684	23	291732
Congressional Voting	218	16	3488

Preprocessing steps used in the experiment

- Handling missing values
- Encoding categorical data using one-hot encoder or label-encoder
- Scaling numerical data
- Handling outliers
- Reformatting time-related data (in-vehicle coupon recommendation)

To prevent data leakage, pre-processing is only measured on training data and then applied to both training and test data.

Experiment setup and techniques used

The experiment was set up to evaluate the performance of three different models on the three different datasets.

Techniques that were used were:

- Our own Neural Networks implementation
- Sequential neural network model from Keras
- K-nearest neighbors

For our custom implementation we fine-tuned it using our own implementation of a random search algorithm from previous slides. We then applied the same parameters such as activation functions, number of nodes, learning rate, number of epochs batch size and optimizer. for the Keras model to make the results as comparable as possible. As for the K-nearest neighbors algorithm, we kept all the parameters default.

Algorithms details and evaluation process

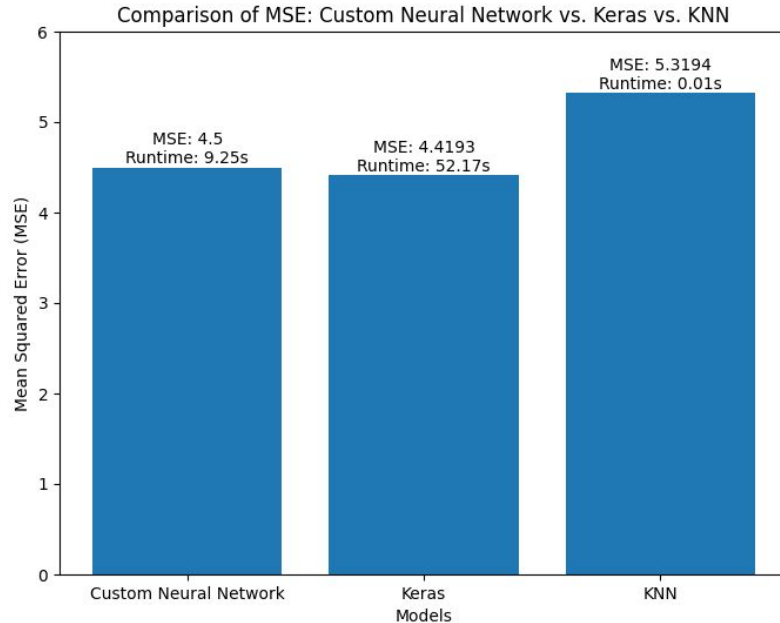
For the evaluation process, we used different metrics across the datasets since the classification task differs and accuracy itself is not sufficient.

- MSE for Abalone Age dataset
- Accuracy and Recall for In-Vehicle Coupon Recommendation dataset
- Accuracy and Precision for Congressional Voting dataset

We performed a 5-fold cross-validation to obtain final results.

In addition we decided to measure training runtimes of algorithms, since it can be a good indicator of how efficient the algorithm is in practice.

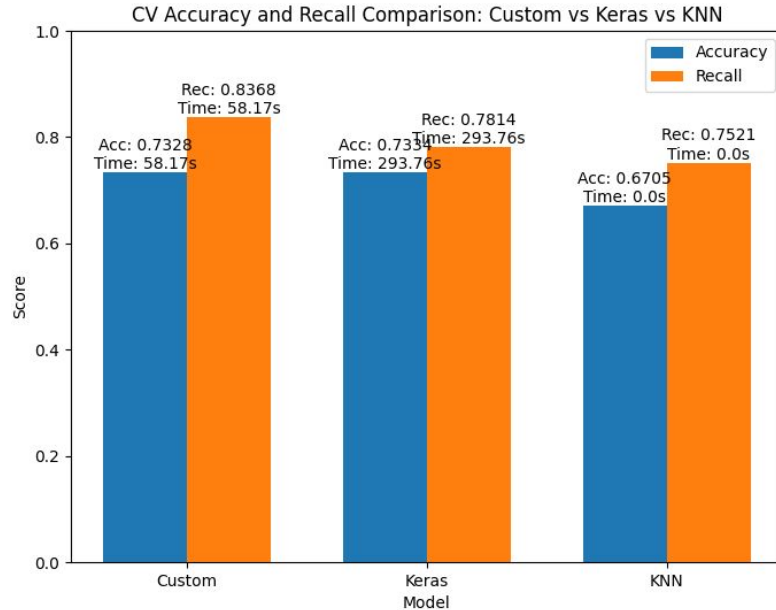
Results - Abalone Age dataset



The custom implementation performed similarly to the Keras model. Both Neural Network approaches vastly outperformed KNN. The custom implementation runs a lot faster than Keras model too. KNN obviously is the fastest here since it is a rather simple approach (expected).

As for hyperparameters, a number of neurons in the first layer affected the results notably.

Results - In-Vehicle Coupon Recommendation dataset

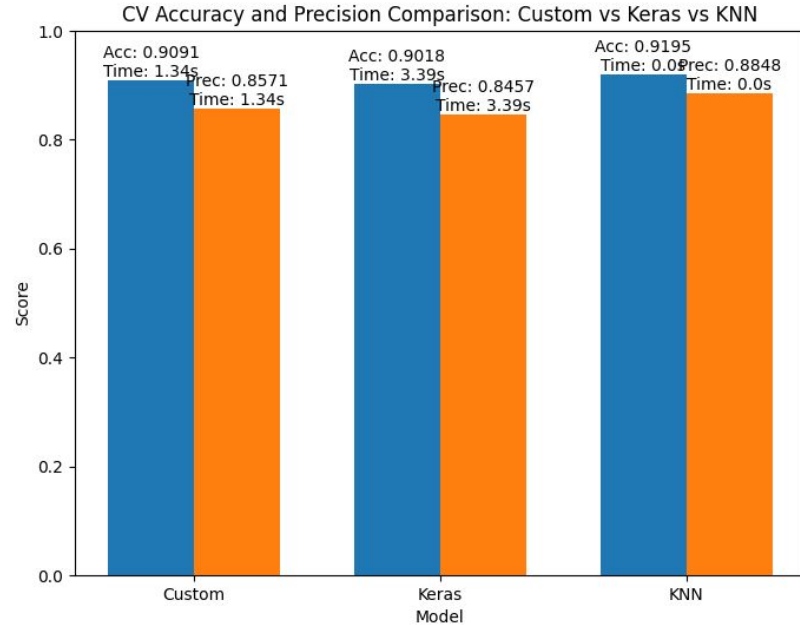


Here, the accuracy of our impl. and of Keras is similar again, however the Recall differs. K-NN has the worst performance again.

This dataset was the most computing time-intensive, however, our implementation was significantly faster again.

As for hyperparameters, results were very similar. A layer size of around 100 neurons proved to be a reasonable choice.

Results - Congressional Voting dataset



Our code achieves very similar accuracy and precision as the Keras model, while running faster again. KNN achieves a slightly better precision, and minimal better accuracy.

As for hyperparameters, the layers and epochs had quite a large variance in the search results. This shows that there are multiple configurations that work quite well for this data set.

Results - summary and conclusions

- Our code had a very similar performance compared to the Keras library. Surprisingly, our SGD was faster than the Keras implementation.
- The randomized search led to results very similar to our more involved hyperparameter optimization of EX1 (we used the same data sets)

Lessons learned:

- Pay attention to side effects: weight initialization was causing problems
- Surprisingly, it is not too hard to compete with well-established libraries