INTRODUCTION TO SCIENTIFIC COMPUTING WITH PYTHON

# Final Project – Deep Feedforward Neural Network

Jiri Stepanovsky
J093010001

June 14, 2021

## 1. Abstract

In this project, we present the theory behind deep neural networks and the principle of gradient descent learning algorithm. We illustrate it by implementing a functional, fully connected neural network with linear, sigmoid, ReLu, and softmax layers trainable by a gradient descent algorithm using mean squared and cross-entropy loss functions. We test our network on regression and multinominal classification tasks. Our results are comparable to the same network implemented in Keras deep learning framework, both in performance and in accuracy.

## 2. Introduction

Feedforward neural networks are simply a function:

$$f : \mathbf{x} \to \mathbf{y} \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is an input vector (also called a feature vector), and $\mathbf{y} \in \mathbb{R}^m$ is an output vector. To obtain an output $\mathbf{y}$ from a feature vector $\mathbf{x}$, a *forward pass* is performed, which simply means computing the function $\mathbf{y} = f(\mathbf{x})$. A typical feedforward network consists of multiple independent layers chained together, so we can write:

$$\mathbf{y} = f(\mathbf{x}) = (f_L \circ \ldots \circ f_2 \circ f_1)(\mathbf{x}) \tag{2}$$

where $f_i$ represents the function of i-th layer in the network with $L$ layers.

### Training neural network

In order to train the network to perform a specific operation, first, it is necessary to evaluate the network's performance. For this purpose a *loss function*:

$$\mathscr{L} : \mathbf{y}, \mathbf{z} \to \mathbb{R} \tag{3}$$

is used, where $\mathbf{y} \in \mathbb{R}^m$ is the output vector from a *forward pass* in Eq. (2), and $\mathbf{z} \in \mathbb{R}^m$ is a target vector. The *loss function* gives an estimate of how much the network output differs from the target, and in its simplest form, it can be computed as the distance between the two vectors:

$$\mathscr{L} = |\mathbf{y} - \mathbf{z}| = |f(\mathbf{x}) - \mathbf{z}| \tag{4}$$

To compute the *loss function*, both $\mathbf{x}$ and $\mathbf{z}$ vectors are required, which means a labeled dataset containing features and corresponding (desired) targets.

We say that to train the network, we mean to minimize the *loss function* over the whole dataset. This is an optimization problem, which can be solved using various methods. In this project, however, a gradient descent algorithm is used.

**Gradient descent method**

Gradient descent is an iterative method that finds the minimum of a function $f$ by following the negative gradient of that function:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \gamma \frac{\partial f}{\partial \mathbf{x}} \tag{5}$$

where $\gamma$ defines the relative step size (known as the learning rate) and $\mathbf{x}$ defines a set of variables that can be adjusted in order to minimize $f$.

During neural network training, we are trying to minimize the *loss function* $\mathscr{L}$ over a training dataset by adjusting the parameters of each layer in the network. We can define the gradient descent update for a neural network similarly to Eq. (5):

$$\mathbf{w}_{i_{t+1}} = \mathbf{w}_{i_t} - \gamma \frac{\partial \mathscr{L}}{\partial \mathbf{w}_i} \tag{6}$$

where $\mathbf{w}_i$ are parameters of i-th layer within the network.

**Backpropagation algorithm**

Computing $\frac{\partial \mathscr{L}}{\partial \mathbf{w}_i}$ in Eq. (6) for deep layers is not straightforward, so lets express it for the last layer of a network with $L$ layers:

$$\frac{\partial \mathscr{L}}{\partial \mathbf{w}_L} = \frac{\partial \mathscr{L}(\mathbf{y}, \mathbf{z})}{\partial \mathbf{w}_L} = \frac{\partial \mathscr{L}(f_L(\mathbf{x}_L), \mathbf{z})}{\partial \mathbf{w}_L} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}} \frac{\partial f_L}{\partial \mathbf{w}_L} \tag{7}$$

The *loss function* depends on the output of the last layer performing a function $f_L$ with input $\mathbf{x}_L$ and parameters $\mathbf{w}_L$. Therefore, a chain rule can be applied to get the expression in Eq. (7). Similarly, the second and third last layers can be expressed as:

$$\begin{aligned} \frac{\partial \mathscr{L}}{\partial \mathbf{w}_{L-1}} &= \frac{\partial \mathscr{L}(f_L(\mathbf{x}_L), \mathbf{z})}{\partial \mathbf{w}_{L-1}} = \frac{\partial \mathscr{L}(f_L(f_{L-1}(\mathbf{x}_{L-1})), \mathbf{z})}{\partial \mathbf{w}_{L-1}} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}} \frac{\partial f_L}{\partial \mathbf{x}_L} \frac{\partial f_{L-1}}{\partial \mathbf{w}_{L-1}} \\ \frac{\partial \mathscr{L}}{\partial \mathbf{w}_{L-2}} &= \frac{\partial \mathscr{L}(f_L(f_{L-1}(f_{L-2}(\mathbf{x}_{L-2}))), \mathbf{z})}{\partial \mathbf{w}_{L-1}} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}} \frac{\partial f_L}{\partial \mathbf{x}_L} \frac{\partial f_{L-1}}{\partial \mathbf{x}_{L-1}} \frac{\partial f_{L-2}}{\partial \mathbf{w}_{L-2}} \end{aligned} \tag{8}$$

There is a clear repeating pattern in Eq. (8), so by expressing:

$$\delta_i = \delta_{i+1} \frac{\partial f_i}{\partial \mathbf{x}_i} \tag{9}$$

for each layer, it is possible to simplify Eq. (7) and (8) to:

$$\frac{\partial \mathscr{L}}{\partial \mathbf{w}_i} = \delta_{i+1} \frac{\partial f_i}{\partial \mathbf{w}_i} \tag{10}$$

Equations (6), (9), and (10) are the core of a gradient descent training algorithm using backpropagation. We start by computing $\delta_{L+1} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}}$ and then gradually compute $\delta_i$ according to Eq. (9) starting from the last layer $\delta_L$. By using $\delta_{i+1}$, it is then trivial to compute Eq. (10) for each layer and subsequently apply the gradient descent update in Eq. (6) for all parameters within the network.

## 3. Methods

In the implementation below, we assume both the input vector $\mathbf{x}$ and target vector $\mathbf{z}$ are row vectors.

### Linear layer

We start with the implementation of a linear layer as seen in Figure 1.

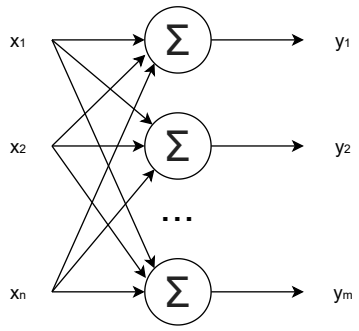


Figure 1: A representation of a linear layer with $m$ outputs

The linear layer consists of $m$ units, called perceptrons, where each unit performs an affine transformation of the input vector $\mathbf{x}$ with $n$ elements:

$$y_i = \sum_{j=1}^{n} x_j w_{ji} + b_i = \mathbf{x}\mathbf{w}_i + b_i \,, \qquad i = 1, 2, \ldots, m \tag{11}$$

which can be written in a matrix form as:

$$f_{linear} = \mathbf{x}\mathbf{W} + \mathbf{b} \tag{12}$$

where weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ represent the layer's parameters.
Equation (12) is the *forward pass* function. To complete the layer, it is also necessary to express the $\delta$ function from Eq. (9):

$$\delta_{linear} = \delta_{i+1} \frac{\partial f_{linear}^{\top}}{\partial \mathbf{x}} = \delta_{i+1} \frac{\partial (\mathbf{x}\mathbf{W} + \mathbf{b})^{\top}}{\partial \mathbf{x}} = \delta_{i+1} \mathbf{W}^{\top} \tag{13}$$

the transposition of $f_{linear}$ is necessary due to $\mathbf{x}$ being a row vector. And finally expressing the gradient descent Eq. (6) with respect to layer's parameters $\mathbf{W}$ and $\mathbf{b}$ using Eq. (10):

$$\begin{aligned} \mathbf{W}_{t+1} &= \mathbf{W}_t - \gamma \frac{\partial \mathscr{L}}{\partial \mathbf{W}} = \mathbf{W}_t - \gamma \delta_{i+1} \frac{\partial f_{linear}^{\top}}{\partial \mathbf{W}} = \mathbf{W}_t - \gamma \delta_{i+1} \frac{\partial (\mathbf{x}\mathbf{W} + \mathbf{b})^{\top}}{\partial \mathbf{W}} = \mathbf{W}_t - \gamma \, \mathbf{x}^{\top} \delta_{i+1} \\ \mathbf{b}_{t+1} &= \mathbf{b}_t - \gamma \frac{\partial \mathscr{L}}{\partial \mathbf{b}} = \mathbf{b}_t - \gamma \delta_{i+1} \frac{\partial f_{linear}^{\top}}{\partial \mathbf{b}} = \mathbf{b}_t - \gamma \delta_{i+1} \frac{\partial (\mathbf{x}\mathbf{W} + \mathbf{b})^{\top}}{\partial \mathbf{b}} = \mathbf{b}_t - \gamma \delta_{i+1} \end{aligned} \tag{14}$$

### Sigmoid layer

Next, lets define a sigmoid activation function. It is popular in regression tasks, since it offers a smooth 'S' shaped output, which is differentiable on $\mathbb{R}$.
The *forward pass* function may look like the *logistic* function:

$$f_{(sigmoid)_j} = \frac{1}{1 + e^{-x_j}} \tag{15}$$

where $x_j$ is the j-th element of input vector $\mathbf{x}$.

The delta function for the sigmoid layer expressed from Eq. (9) is therefore:

$$\delta_{(sigmoid)_j} = \delta_{(i+1)_j} \frac{\partial f_{(sigmoid)_j}}{\partial x_j} = \delta_{(i+1)_j} \frac{e^{-x_j}}{(1+e^{-x_j})^2} \tag{16}$$

which can be written in a vector form as:

$$\delta_{sigmoid} = \delta_{(i+1)} f_{sigmoid}(1 - f_{sigmoid}) \tag{17}$$

Because sigmoid layer does not have any parameters, there is no gradient descent update.

**ReLu layer**

ReLu activation function has a sharp transition, which allows to split the input feature space into multiple domains. The *forward pass* function is:

$$f_{(ReLu)_j} = \begin{cases} 0 & x_j \leq 0 \\ x_j & x_j > 0 \end{cases} \tag{18}$$

where $x_j$ is again the j-th element of input vector $\mathbf{x}$.

The $\delta$ function for ReLu layer expressed from Eq. (9) is:

$$\delta_{(ReLu)_j} = \delta_{(i+1)_j} \frac{\partial f_{(ReLu)_j}}{\partial x_i} = \begin{cases} 0 & x_j \leq 0 \\ \delta_{(i+1)_j} & x_j > 0 \end{cases} \tag{19}$$

and similar to sigmoid layer, there are no parameters for a gradient descent in Eq. (6).

**Softmax layer**

Softmax is a function commonly used to normalize the output of a neural network to a probability distribution. It is useful mainly in multinominal classification tasks. The *forward pass* function looks like:

$$f_{softmax} = \frac{e^{\mathbf{x}}}{\sum_{i=1}^{n} e^{x_i}} \tag{20}$$

The $\delta$ function is then:

$$\delta_{softmax} = \delta_{i+1} \frac{\partial f_{softmax}}{\partial \mathbf{x}} = \delta_{i+1} \cdot [diag(\mathbf{x}) - (f_{softmax}^{\top} \cdot f_{softmax})] \tag{21}$$

Again, the layer does not have any parameters, so there is no gradient descent update.

**Mean squared loss function**

As already mentioned in Eq. (4), a common loss function is the distance between output vector $\mathbf{y}$ and target vector $\mathbf{z}$. Mean squared loss function computes the mean distance as:

$$\mathscr{L} = \frac{1}{N} |\mathbf{y} - \mathbf{z}|^2 \tag{22}$$

where N is the length of output and target vectors. It has the same properties as Eq. (4), but has a more practical derivation for the $\delta$ function:

$$\delta_{L+1} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}} = \frac{1}{N} \frac{\partial |\mathbf{y} - \mathbf{z}|^2}{\partial \mathbf{y}} = \frac{2}{N}(\mathbf{y} - \mathbf{z}) \tag{23}$$

**Cross-entropy loss function**

Cross-entropy loss function is commonly used in combination with softmax function in multinominal classification tasks. It is defined as:

$$\mathscr{L} = -\sum_{i=1}^{N} z_i \log(y_i) \tag{24}$$

where $z_i$ and $y_i$ are the i-th elements of vectors $\mathbf{z}$ and $\mathbf{y}$, and N is their length. The $\delta$ is:

$$\delta_{(L+1)_i} = \frac{\partial \mathscr{L}_i}{\partial y_i} = \frac{\partial(\ -z_i \log(y_i)\ )}{\partial y_i} = -\frac{z_i}{y_i} \tag{25}$$

**Softmax + Cross-entropy loss**

In softmax layer, we perform $e^x$ and in cross-entropy loss we do $\log(x)$. If we combine these layers together, although, the *forward pass* function stays the same, it significantly simplifies the computation of $\delta$ function.

The $\delta$ function of softmax layer in Eq. (21) can be expressed in index notation as:

$$\delta_{(softmax)_j} = \delta_{(i+1)_j} y_j - \delta_{i+1} \mathbf{y}^\top y_j = \delta_{(i+1)_j} y_j - \sum_{k=1}^{N} \delta_{(i+1)_k} y_k y_j \tag{26}$$

where N is the length of vector $\mathbf{y}$. Because this softmax layer is combined with cross-entropy loss, we can substitute $\delta_{i+1}$ in Eq. (26) for the corresponding $\delta$ function described in Eq. (25). This simplifies the Eq. (26) to:

$$\delta_{(softmax+cross)_j} = -\frac{z_j}{y_j} y_j - \sum_{k=1}^{N} -\frac{z_k}{y_k} y_k y_j = -z_j + y_j \sum_{k=1}^{N} z_k \tag{27}$$

Both softmax and cross-entropy loss are mainly used in multinominal classification tasks. In these tasks, the target vector $\mathbf{z}$ is commonly one-hot encoded. It means, that the input features should be classified as only a single class, which is represented by single 1 in otherwise zero target vector (target vector could for example look like $\mathbf{z} = [0\ 1\ 0\ 0\ 0]$). This also means, that for any one-hot encoded target vector $\mathbf{z}$ the following holds:

$$\sum_{i=1}^{N} z_i = 1 \tag{28}$$

So when using a one-hot encoded target vectors, we can simplify Eq. (27) to:

$$\delta_{(softmax+cross)_j} = -z_j + y_j \sum_{k=1}^{N} z_k = -z_j + y_j \cdot 1 = y_j - z_j \tag{29}$$

Compared to the previous $\delta$ functions for softmax and cross-entropy loss layers in Eq. (21) and (25), the $\delta$ function for the combination of these two layers is significantly simpler. Equotation (29) can be also expressed in vector notation simply as:

$$\delta_{softmax+cross} = \mathbf{y} - \mathbf{z} \tag{30}$$

### Training a network

When training a network composed of the above layers, first, the output vector $\mathbf{y}$ of the whole network is computed according to Eq. (2) from an input vector $\mathbf{x}$ in the training dataset using the *forward pass* functions defined in Eq. (12), (15), (18), and (20). Then, a gradient of the *loss function* is computed according to Eq. (23) or (25) depending on the loss function used (or according to Eq. (30), if the combination of softmax and cross-entropy loss is used). By propagating this gradient from the last layer to the first, we can compute $\delta$ functions for all layers in the network using Eq. (13), (17), (19), and (21). Finally, we compute the update of the network's parameters using the gradient descent algorithm. That actually means computing the update using Eq. (14) for linear layers only, since there are no other layers with parametes in this implementation.

The network's parameters are not updated after each sample $\mathbf{x}$. For a better stability, the process above is repeated for a certain number of samples in the dataset, and the parameters are then updated according to the average trom all these samples. The number of samples required to perform an update is called a batch.

Each update of the parameters after processing a batch is called an epoch, and the training ends after a selected number of epochs was finished.

### Regression experiment

The implemented network was tested on a curve fitting experiment. The input $x$ was a random real value within an interval $[-2\pi, 2\pi)$, and the target was a function $\sin(x)$.
The network's architecture is in Table 1, the sigmoid layer was used at the end to smooth out the sharp edges from ReLu layers. The loss function is mean squared from Eq. (22).

| **Layer:** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **L+1** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Type:** | input | linear | ReLu | linear | Relu | linear | Relu | linear | sigmoid | linear | loss |
| **Size:** | 1 | 10 | 10 | 100 | 100 | 100 | 100 | 10 | 10 | 1 | 1 |

Table 1: The architecture of the network used on a curve fitting experiment

The network was trained with a learning rate 0.01 for 100000 epochs with 100 random samples per batch. The testing dataset consists of 500 real values equidistantly spaced over the interval $[-2\pi, 2\pi]$ with the target being again $\sin(x)$.

### Multinominal classification experiment

The second experiment was a digit classification task using MNIST public dataset of handwritten digits [1]. Both the training and testing datasets were normalized. The first 200 samples from the training dataset can be seen in Figure 2.

The network's architecture is in Table 2, it contains several ReLu layers followed by one softmax layer. The loss function was a cross-entropy loss from Eq. (24).

| **Layer:** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **L+1** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Type:** | input | linear | ReLu | linear | Relu | linear | Relu | linear | softmax | loss |
| **Size:** | 28x28 | 14x14 | 14x14 | 14x7 | 14x7 | 7x7 | 7x7 | 10 | 10 | 10 |

Table 2: The architecture of the network used on a digit classification experiment

The network was trained with a learning rate 0.1 for 1000 epochs with 2000 samples per batch. The dataset was not shuffled, so the samples were taken in sequence.

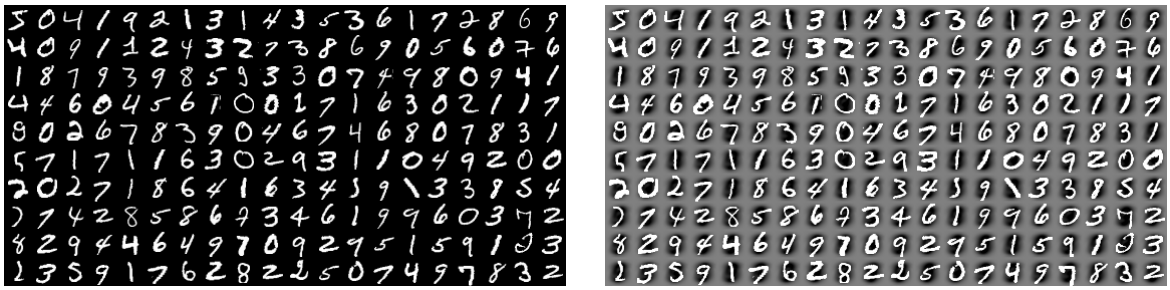

Figure 2: The first 200 samples from MNIST training dataset. Original (left), normalized (right).

## 4. Results

Both experiments were carried out on our implementation and on a model using Keras deep learning framework [2]. In both cases, the network's architecture, training algorithm, training dataset, loss function, batch size, and the number of epochs were the same.

**Regression experiment**

Results from the first experiment are summarized in Table 3.

| | Our implementation | Keras implementation |
|---|---|---|
| Training time: | 77 secs | 118 secs |
| Loss on testing dataset: | $3.2 \cdot 10^{-5}$ | $1.6 \cdot 10^{-5}$ |

Table 3: Results from the regression experiment

Our network required less time to finish training but performed a bit worse than the network implemented in Keras framework. The output for both networks over the interval $[-3\pi, 3\pi]$ can be seen in Figure 3.

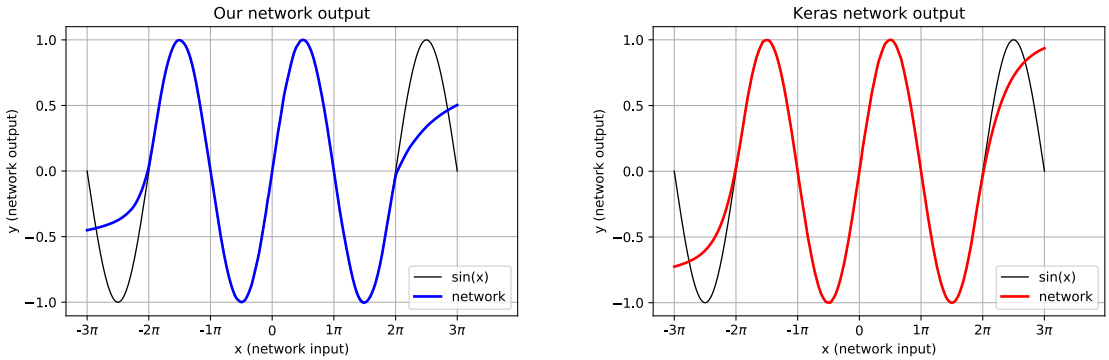


Figure 3: The network's outputs over the interval $[-3\pi, 3\pi]$ in the regression experiment

Both models perfectly matched the target function on a training interval $[-2\pi, 2\pi)$, but, as expected, they significantly diverge from the target outside of the training interval. This is a common issue, and the only solution would be to use a different, possibly recurrent, neural network architecture, that is able to generalize a periodic function.

The loss functions during training are in Figure 4. Our implementation seems to have a bit more jitter during the training.

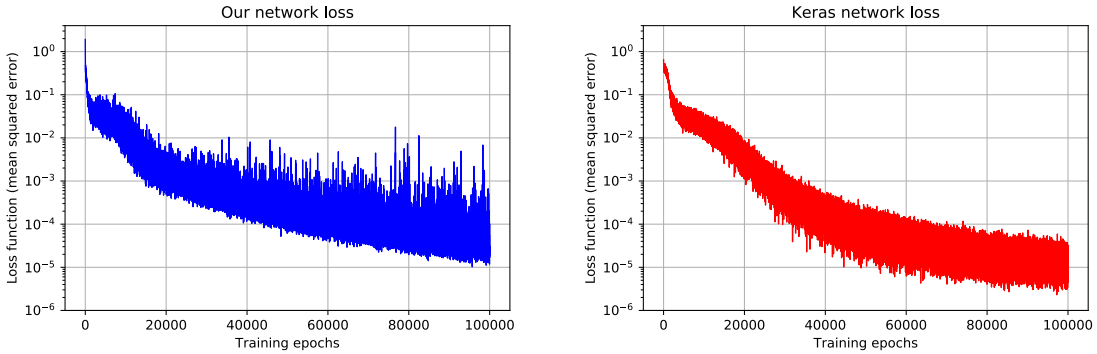


Figure 4: The network's loss functions during training

### Multinominal classification experiment

The classification experiment was performed on three models. One was using our implementation with separate softmax layer and cross-entropy loss, one was using our implementation with combined softmax layer and cross-entropy loss as defined in Eq. (30), and the last one was using the Keras framework. Results from the experiment are summarized in Table 4.

| | Our: separate layers | Our: combined layers | Keras implementation |
|---|---|---|---|
| Training time: | 84 secs | 65 secs | 20 secs |
| Accuracy on training data: | 97.8% | 97.7% | 98.6% |
| Accuracy on testing data: | 96.3% | 96.3% | 96.5% |

Table 4: Results from the multinominal classification experiment

The simplification of the $\delta$ function computation in Eq. (30) noticeably reduced the training time, however, the Keras framework still performed several times faster.
The accuracy during training can be seen in Figure 5. The progress is clearly periodic, which is caused by the repeating format of the training dataset, as it is not shuffled.

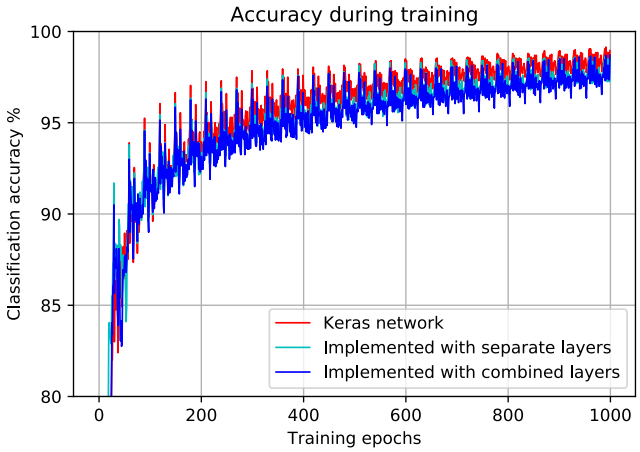


Figure 5: The accuracy progress of the three networks during training

## 5. Discussion

All results presented above are only illustrative, and there is no intention to precisely compare the performance of our implementation of deep feedforward neural network to the Keras deep learning framework. A much more thorough evaluation would be necessary to arrive at a conclusion with any statistical significance.

We may, however, conclude that our implementation can be successfully trained on both regression and classification tasks. The required training time is not too far off of the Keras framework, and the accuracy and loss are very close to it, though not as good.

## 6. Conclusion

We have presented the mathematical background for deep feedforward neural networks. We have described some common layers and loss functions and introduced the concept of a backpropagation algorithm with a gradient descent optimization. Finally, we have implemented the mathematical model into a functional neural network and successfully tested it on regression and classification experiments.

## References

[1] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner. (1998). Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86(11):2278-2324. http://yann.lecun.com/exdb/mnist/

[2] Keras deep learning framework. (June 2021). https://keras.io