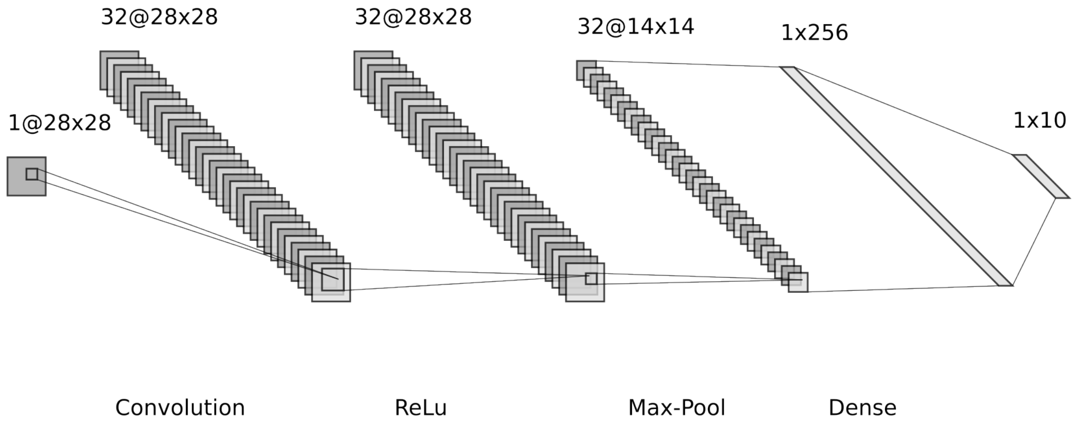
UNIVERSIDAD DE GRANADA

Project assignment

Developing an image recognition neural network from scratch

December 8th, 2024
Granada



Name:	Raoul Luqué
University:	Universidad de Granada
Study program:	Computer Science M.Sc.
Teachers:	Prof. Berzal,
	David Criado

Contents

1 Introduction

This work is supposed to accompany a project for a computational intelligence course that I took part in while in my Erasmus semester abroad at the Universidad de Granada. We will try to develop a neural network for recognizing digits with an error rate of below 00.30%, that is, a maximum of 30 errors on 10,000 images.

Our goal will be to implement a neural network from scratch in Python that achieves the error rate of 0.30% while not using any other libraries/code other than NumPy for the linear algebra. We will start by revisiting the basics of neural networks, including forward and backward propagation. Based on this, we will discuss the first model that was designed to achieve the task of classifying digits. After that, we will introduce several techniques to refine our model and introduce new models as we learn enough to be able to understand the theory behind them. At last, we will reach the final model based on our implementation, the twelfth model. After this, there is another model using TensorFlow, since our implementation does not run on GPUs, and therefore it would not have been feasible to train the model we were aiming for.

This work assumes basic knowledge of analysis (multidimensional differentiation) and some linear algebra (matrix multiplication and scalar products) and will assume all scalar products to be the standard scalar product in $\mathbb{R}^n$ ($\mathbb{R}$ being the real-numbers of course). Furthermore, we will use zero indexing when we see fit, since it is the default in Python. At last, we will use the terms artificial neural network, neural network and network interchangeably.

2 Neural network basics

To start, we want to briefly revisit the basics of (artificial) neural networks. For this, let us remind ourselves of the idea behind neural networks. Coming from biology and, more specifically, the study of brains, one can observe that the structure of a brain consists of tens of millions of interconnected neurons that function in a very complex network.

Breaking it down however, one neuron has several dendrites with which it receives electrical signals from other neurons and, simply speaking, if a certain threshold of summed signal strength is exceeded, the neuron fires a signal via its axon to the dendrite of another cell. This leads us already to the structure of our neurons in artificial neural networks. For further reading however, please refer to [Wik24].

We want to start by looking at the visualization of a neuron in an artificial neural network, shown in figure 9.1.

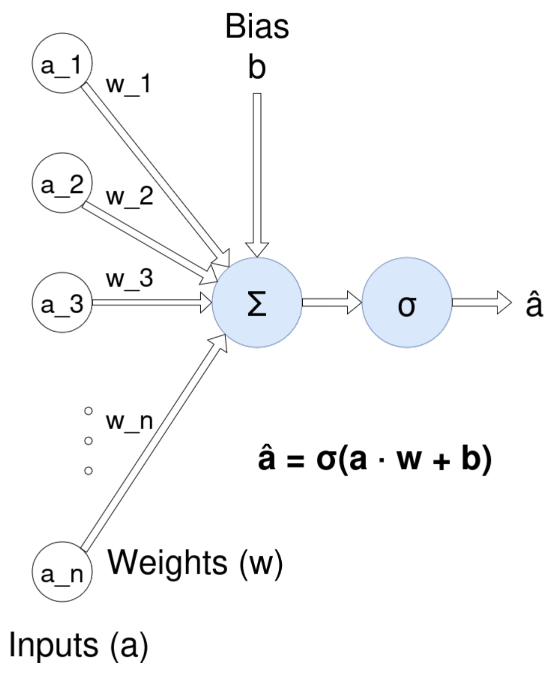


Figure 2.1: Visualization of neuron in artificial neural network[1]

We will try to understand the entirety of the visualization in the course of this section. First, we focus on the red circles on the left and the big circle with an Σ in the middle. The red circles on the left can be interpreted as other neurons, or rather the outputs of other neurons, whereas the big blue circle with the Σ can be interpreted as the neuron we are currently looking at. Technically, one could say that both blue circles are what makes up the neuron, but we will discuss that in a minute. This structure is of course quite similar to the one that can be observed in brains like the one of humans. One neuron has multiple inputs and one output (the latter will be different in our models but more on that later). Also, similar to how biological neurons interconnect, connections can be stronger or weaker, this is represented by weights, $w_1, \ldots, w_n$ that connect $a_1, \ldots, a_n$ with Σ, respectively. These weights are one of the most important parameters, that is, variables we will be adjusting when training our network.

Now, similar to how a biological neuron fires when a certain threshold is surpassed, our neuron passes a signal based on the signals received. This is displayed by the formula $\hat{a} = \sigma(w \cdot a + b)$. Note that w and a are both just vectors of the weights and entry signals, and therefore $w \cdot a$ is just their scalar product. The b refers to a bias parameter which we will be adjusting throughout training, just like the weights. It is just added to the scalar product and therefore does not depend directly on the inputs from the other neurons. Note that we will also be referring to the term so far, $w \cdot a + b$, as z. The last part, making up $\hat{a} = \sigma(z)$ is σ, the so-called activation function. Remember that in biological neurons, the total input signal had to exceed a certain threshold, well in

[1]Source of image: Self made, using app.diagrams.net

our neural networks, the activation function takes care of that. A neuron simply passes the signal that the activation function outputs. Note that when we refer to signals for our artificial neural networks, we are actually talking about floating point numbers or vectors of such. Coming back to what the neuron actually consists of, one might argue that the neuron is both its activation function and the linear first part, Σ. For now, we will skip over the details of what an activation function and just say it is a fixed function from $\mathbb{R}$ to $\mathbb{R}$. We will however discuss this more thoroughly in chapter **??**.

The last step, before being able to understand the entirety of a neural network, is looking at how neurons are laid out to form a neural network. Such a basic neural network is depicted in figure 2.2.

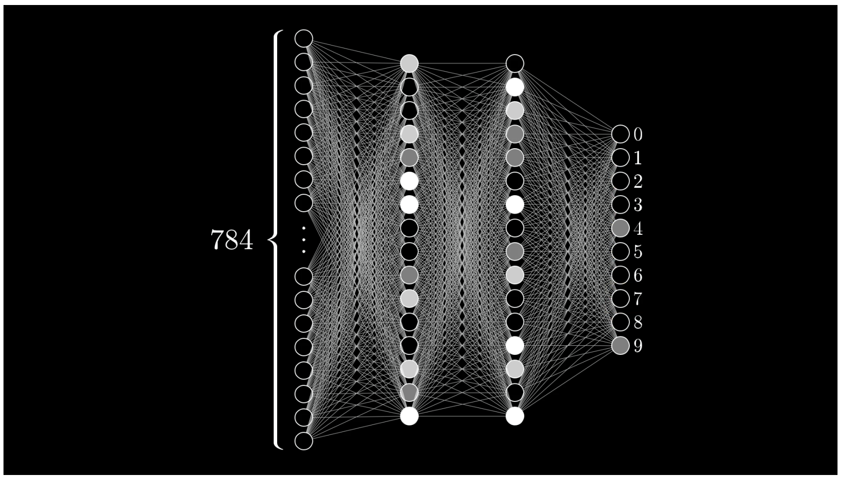


Figure 2.2: Basic neural network[2]

In the figure above, each dot (no matter the color) represents a neuron. Therefore, the neurons can be divided into four vertical lines, also called layers. The first layer being the input layer, which consists of 784 neurons in this case. The last layer is the output layer, with 10 neurons in this case. We can see that there are two layers in between the input and output layers, these are also referred to as hidden layers. In order to understand this, let us actually talk about what this neural network was designed to do. It is, coincidentally, a network for recognizing digits given images of size $28 \times 28 = 784$. Each of the 784 neurons in the input layer corresponds to a pixel in the $28x28$ input image and gets fed its brightness value (we are considering images without color, i.e. only a brightness value for each pixel). These signals are then passed through the network.

Since there are so many neurons, it is not clearly visible in the figure, but each neuron (or rather its output) is connected with all neurons (or rather their respective input) from the next layer. More importantly, neurons from the same layer are not connected and there is no connection from the output of a neuron to the input of a neuron in a previous layer. These type of layers are also called fully connected layers (FC) due to all neurons from one layer (or rather their respective outputs) being connected with all

[2]Source of image: https://www.3blue1brown.com/lessons/neural-networks

neurons from the next layer (or rather their respective their inputs). As a side note: this of course breaks the analogy to biological neurons, since biological neurons only have one axon and therefore output and our neurons will be able to output their output signal to arbitrarily many neurons. Also note that the output signal of neurons in the input layer is just their respective input signal, and in our case no activation function, weight or bias will be applied.

The input signal which was received by the input layer is then propagated to the next layers' neurons as shown in figure 9.1. This process is therefore also referred to as forward propagation, since the signal from one layer as a whole can be propagated to the next layer. For now, we will skip the details of what the values of the weights and biases are and what activation functions are specifically and discuss this in the next section 2.2. For exemplary visual purposes, in the above figure 2.2, neurons that have a higher output (that are firing 'stronger') are shown as more white than those that are outputting a lower signal. The network will then forward propagate the signal through all layers until reaching the output layer (the right-most layer). This layer has 10 neurons (in our case), each of which will have an own output signal after the forward propagation. These can then be used to determine which class the network thinks the input image corresponds to. Or, in our case, which digit the input image is. Usually, the neuron which has the highest output is then used to determine the prediction of the network (the neurons are labelled according to the classes).

The notation and information used in the above section is taken from 3Blue1Browns' course on neural networks [San17, But what is a Neural Network?].

To recap, we feed our neural network images of 28×28 and it will output 10 values from the last layer of the network. This will stay the same for all networks/models that were developed in this work, that is 28×28 in and 10 out, whereas the layers in between will change depending on the specific model. That is why, those layers are also referred to as hidden layers, since an outside observer can only see the input and output (layers).

Having done this brief revisit of the basics of neural networks, we will dive into the actual learning process of neural networks in the following section. Before doing so however, we have to talk briefly about the encoding and dataset used in this work.

2.1 Dataset and setup

The dataset used for this work is the MNIST dataset, which can be found here, a dataset of 60,000 training images, 10,000 test images and their respective labels. The images all show digits, that is, numbers from 0 to 9, and are of shape 28×28. Each pixel has a brightness value of 0 to 255 which we will however be normalizing to 0 to 1 for easier use in our network. Note that, as we will discuss in the following section, it is important that our dataset is labeled, that is, for each image, the label (digit) is given. These are given as integers from 0 to 9, we will however convert them to an encoding called one-hot encoding.

Given an image of size 28×28, the corresponding label shall be a vector of size 10 of only zeros except for the index of the digit the label corresponds to. That is, the label of a zero will look like:

$$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)^T$$

This will simplify computations in the following sections. First and foremost, this will be the same dimension as the output our networks will give us (not only the one shown in figure 2.2, but all with an output layer with 10 neurons). That is, we will take the output values of the neurons in the last layer as our output, which will therefore be a vector of size 10.

For testing the different neural network (models), we feed all 10,000 test images through the network and evaluate the percentage of images that were not classified correctly. This will be referred to as the error rate.

2.2 Gradient descent

We finished the neural network section by discussing how an input signal (in our case an 28×28 image) in a neural network is propagated forward through the layers to obtain the output of the network (in our case a digit). What we left out in the previous section however is the values of the bias and weights when propagating the signal. This leads us right to the learning process of a neural network, which is based on gradient descent.

From the previous explanations, it is easy to see that the bias and weights of the neurons, or rather their connections, determine the output of the network (given a fixed input and activation function(s)). Therefore, what we want to achieve by learning is adjusting these parameters, that is, the weights and biases, to be the values that produce the best results, that is, increase the accuracy of our network at predicting digits.

Now, gradient descent solves this with the following basic idea. We will find these adjustments by looking at the gradient of the error that our network commits when trying to predict a digit it gets fed as input. As is known from basic analysis classes, the gradient points towards the steepest ascend of the function, i.e. the direction towards which the input would have to be adjusted to obtain a higher function value. By now using the sign inverted gradient, we obtain the direction towards which the input parameters would have to adjusted to reduce the function output. Note that the function output in this case is the error function output, therefore we would be reducing the error - which is good. An analogy of gradient descent in 2D would be to be in a mountainous region and trying to reach a low point. One can just look at the gradients, i.e. inclination, in the different directions and then take a step towards the direction with the steepest decline. This is what gradient descent does. See [GBC16, p. 82] for a more rigorous definition.

To reiterate, the gradient descent method will try to minimize the error function by changing the weights and biases which function as inputs to the error function, since they determine the output of the network. This is done by shifting the weights and biases towards the direction that locally decreases the error.

Of course, for gradient descent to determine the direction towards which the weights and biases should be changed, it has to have a starting point. This is the initialization of the weights and biases. We will discuss this topic more thoroughly in chapter 8, but for now assume that the weights and biases are initialized with a uniform random distribution in the interval $[-0.5, 0.5]$.

2.2.1 Error function

One might now ask what the error even means in the above explanations. That is exactly what we want to discuss in the following. A naive error function, which we will also be using to start, is the mean-squared error function (MSE).

Definition 2.1 (Mean-squared error function [GBC16, p. 108])**.** Given an output of the network (prediction) $\hat{y}$ and the actual label of the image y in one-hot encoding, the mean-squared error is:

$$\sigma_{MSE}(\hat{y}, y) = \frac{1}{\mathcal{C}} \sum_{i=0}^{\mathcal{C}-1} (\hat{y}_i - y_i)^2$$

where $\mathcal{C}$ is the number of classes.

Note that in the above definition $\mathcal{C} = 10$, since we are classifying 10 different digits. Also, the MSE is of course closely related to the $2-$norm, or Euclidean norm, since $\sigma_{MSE}(\hat{y}, y) = \frac{1}{\mathcal{C}} \|\hat{y} - y\|_2^2$. The latter definition is interesting of course, since we can then interpret the prediction of our network as a point in $\mathcal{C}-$dimensional space, where the MSE just calculates a scaling of the square of the Euclidean distance to the actual label.

As we will see in chapter 4, MSE is not actually the best suited loss function for categorization problems (as ours is one) and instead used more in regression tasks.

2.2.2 Gradient descent calculus

Having answered the question what error functions are, we can actually calculate the gradient (at least for specific error functions) and specify exactly what gradient descent does. To do so, let us first establish an example we will be basing the following computations on. We want to consider a network consisting of only 2 (fully connected) layers. The input and the output layers with 784 and 10 neurons, respectively.

We will use w_{ij} to denote the weight connecting the neuron i from the input layer to the neuron j from the output layer, use b_j to denote the bias corresponding to the neuron j of the output layer and a_j to denote the output of neuron j in the output layer, as done above.

Let us start by looking at the gradient of the MSE function. Using the notation from figure 9.1 and with basic derivation rules, we obtain:

$$\frac{\partial \sigma_{MSE}}{\partial \hat{y}_i} = \frac{\partial \sigma_{MSE}}{\partial a_i} = \frac{2}{\mathcal{C}}(a_i - y_i) \tag{2.1}$$

Note that in figure 9.1 we used a as the output of a neuron. And since the output layer is also just a layer, we denote with a_i the output of one of the $\mathcal{C} = 10$ neurons in the output layer. This will allow for easier notation in the following computations.

The gradient from equation 2.1 can be interpreted as giving us the rate of change of the error function σ_{MSE} with respect to the input of the error function a (the output of the network). We, however, are interested in the rate of change (the gradient/derivative) of the error function with respect to the weights and biases, respectively. That is, we are interested in $\frac{\partial \sigma_{MSE}}{\partial w_{ij}}$ and $\frac{\partial \sigma_{MSE}}{\partial b_j}$.

Luckily, the chain rule comes in very handy for computing these partial derivatives. With it, we obtain for the weights:

$$\frac{\partial \sigma_{MSE}}{\partial w_{ij}} = \frac{\partial \sigma_{MSE}}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ij}} \tag{2.2}$$

Here, we can apply the chain rule again to $\frac{\partial a}{\partial w_{ij}}$ and obtain:

$$\frac{\partial \sigma_{MSE}}{\partial w_{ij}} = \frac{\partial \sigma_{MSE}}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}} \tag{2.3}$$

Note that the notation from figure 9.1 was used to refer to z_j as the linear combination of the inputs, weights and bias of the neuron j of the output layer. Since we have not specified an activation function yet, we can not compute $\frac{\partial a_j}{\partial z_j}$ for our neural network, we can however compute the other partial derivates and obtain:

$$\frac{\partial \sigma_{MSE}}{\partial w_{ij}} = \frac{2}{\mathcal{C}}(a_i - y_i) \cdot \frac{\partial a_j}{\partial z_j} \cdot w_{ij} \tag{2.4}$$

Similarly, we obtain:

$$\frac{\partial \sigma_{MSE}}{\partial b_j} = \frac{\partial \sigma_{MSE}}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_j} = \frac{2}{\mathcal{C}}(a_i - y_i) \cdot \frac{\partial a_j}{\partial z_j} \cdot 1 \tag{2.5}$$

Using these partial derivatives, we can now update our weights and biases. One detail we have omitted so far, is a parameter called the learning rate α. It is multiplied by the derivative before subtracting it from the current parameter values. Therefore, the parameters are updated as follows:

$$w_{ij} \leftarrow w_{ij} - \alpha \cdot \frac{\partial \sigma_{MSE}}{\partial w_{ij}} \qquad b_j \leftarrow b_j - \alpha \cdot \frac{\partial \sigma_{MSE}}{\partial b_j} \tag{2.6}$$

We will take a deeper look into the learning rate parameter in chapter 10. For now, we assume that the learning rate stays the same over the course of the learning process.

To recap, we have derived the gradient descent method for a neural network with 2 layers. Applying the method to networks with more than two layers, that is, at least one hidden layer, exactly as we have done above except that the computation of the partial derivatives changes slightly. Suppose we have a network with 3 layers. The first layer containing 784 neurons, the second with 100 and the third with 10. Computing the partial derivative of the error with respect to the output of a neuron on layer 3 stays the same, that is

$$\frac{\partial \sigma_{MSE}}{\partial a_k^{(3)}}. \tag{2.7}$$

Note that we are using the exponent 3 in this case to denote the layer the output corresponds to, and the index k to denote the neuron's index in the respective layer. If one now wants to compute,

$$\frac{\partial \sigma_{MSE}}{\partial a_j^{(2)}}, \tag{2.8}$$

that is, the partial derivative of the error function with respect to the output of the $j-$th neuron in the second layer, $a_j^{(2)}$ influences the error via all neurons in the third layer. More precisely:

$$\frac{\partial \sigma_{MSE}}{\partial a_j^{(2)}} = \sum_{k=0}^{9} \frac{\partial z_k^{(3)}}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(3)}}{\partial z_k^{(3)}} \cdot \frac{\partial \sigma_{MSE}}{\partial a_j^{(3)}} \tag{2.9}$$

Of course all of the above applies similarly to varying error functions, a furthermore increased number of layers (and activation functions, once we discussed those).

One last thing that has to be addressed is, that we technically have not been doing classic gradient descent. Note that we have only looked at one example when computing the error and therefore the gradient with respect to our weights and biases. However, if we want to know how to optimize our models' weights and biases to more accurately categorize all 60,000 training examples, we would to compute the gradients with respect to all training examples, or rather average their gradients. This is what gradient descent would actually be doing. But as we will be seeing in the next section, this is not feasible for training neural networks, therefore different variations of gradient descent exist. The above variation is usually referred to as stochastic gradient descent (SGD) where we take one example and take the gradient of the error with respect to the parameters (for one example) to update our parameters, knowing that the gradient is only stochastically estimating the gradient with respect to the entire training set. One then repeats this process for all samples in the training set.

To read more on gradient descent and its calculus with visually appealing images, see [San17, Backpropagation Calculus].

2.3 Epochs

We want to use this section to discuss a common neural network term. As we have just learned, we will be using a variation of gradient descent to train our neural network, which we will be referring to as stochastic gradient descent. According to this variant, we will look at all test samples one by one and for each sample update all the weights and biases in the network using the computed gradients of the error function (and multiplying it by the learning rate). After having seen all samples from the training set, we will start over and iterate the training set again. We will refer to one loop iterating over the entire set once as an epoch. The number of epochs will be a hyperparameter that can be set before running the model.

Furthermore, before each epoch, we will be shuffling all training samples (and labels) in order to reduce the bias (not the bias in the layers/neurons) the order of the samples may have on training the network.

Having discussed the theoretical details, we can look at an implementation of the forward and backward propagation. Of course, we will vectorize the operations we discussed above to greatly decrease the computation time, a reoccurring theme as we will see.

2.4 Forward propagation code

We want to use the following two sections to show and discuss the implementation of the forward and backpropagation functions for fully connected layers. The code can be found here

For forward propagation, we can express the computations as a vector matrix multiplication if the weight matrix contains weights w_{ij} in entry (i, j). Then we simply have to add the bias to the resulting vector.

```python
def forward_propagation(self, input_data: NDArray) -> NDArray:
    self.input_data: NDArray = input_data
    return np.dot(self.input_data, self.weights) + self.bias
```

Listing 1: Forward propagation for fully connected layer

Note that we apply the activation function in a separate layer, which would follow the fully connected layer right after. Its implementation will be discussed in section 2.6. Also, we are caching the input data, since it will be used in the backpropagation step. For this, we have implemented the fully connected layer as a class and are actually looking at a method of it.

2.5 Backward propagation code

The backward propagation step is similarly simple, hence first show the code and then discuss its details.

```python
def backward_propagation(self, output_error: NDArray, learning_rate: NDArray) -> NDArray:
    input_error: NDArray = np.dot(output_error, self.weights.T)
    weights_error: NDArray = np.dot(self.input_data.T, output_error)

    # Update weights and bias
    self.weights -= learning_rate * weights_error
    self.bias -= learning_rate * output_error
    return input_error
```

Listing 2: Backward propagation for fully connected layer

Note that the output error in this case is the error $\frac{\partial E}{\partial z^{(L)}}$ if E is the error function and $z^{(L)}$ is the output of the current layer (input of the next layer, which is an activation layer). Therefore, as we have seen in section 2.2.2, we obtain the input error $\frac{\partial E}{\partial a^{(L-1)}}$ by multiplying the output error by $\frac{\partial z^{(L)}}{\partial a^{(L-1)}}$, that is, the weights. We can then propagate this input error further to the previous layer. The weights' error on the error hand is computed by multiplying the output error by $\frac{\partial z^{(L)}}{\partial w^{(L)}}$ which is just the input data (see, figure 9.1 if the relation of a, z and w is unclear).

2.6 Activation functions

As we have discussed before, activation functions act as a way to modularize the output of a neuron before propagating its signal to the inputs of the neurons its output is connected to. They are usually put immediately or slightly after every fully connected (or other layer with weights and biases). However, this slightly understates the importance of activation functions. Before discussed this, we have to look at some examples of activation functions. A relatively simple and very effective activation function, which we will be focussing on in this work, is the rectified linear unit (ReLu) function. Simple put, given a vector, it computes $\max(0, x)$ entrywise.

Definition 2.2. The rectified linear unit function is defined as:

$$\sigma_{ReLu} : \mathbb{R}^n \to \mathbb{R}^n, \sigma_{ReLu}(x_i) = \max(0, x_i)$$

This function can be applied to vectors and even matrices of any size, and it stands out with its very easy implementation. Similarly easy, we can compute the derivative:

$$\frac{\partial \sigma_{ReLu}}{\partial x_i} = \begin{cases} 0, & \text{if } x_i <= 0 \\ 1, & \text{else} \end{cases} \tag{2.10}$$

Other activation functions may include logistic or hyperbolic tangent functions. For more information on why ReLu is suitable for larger convolutional networks (where we will end up), see for example [GBB11].

We also want to briefly show the tanh (hyperbolic tangent) activation function, since this one was used for the first neural networks in this work. For this, let us instead look at a plot of the function.

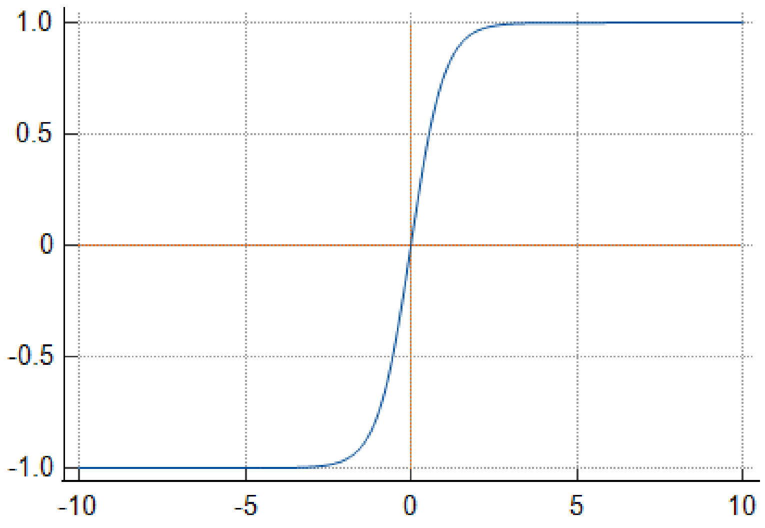


Figure 2.3: Plot of tanh[3]

As one can see (at least guess from the above section of tanh, we are seeing), the tanh functions maps all real values to a value in the interval $[-1, 1]$. This is interesting for our purposes, because therefore signals cannot exceed a threshold and will be 'normalized' by the activation function to be in a certain interval.

Let us now get back to why we even need activation functions. Before this section, our network seemed to have worked and learned and fine, and we did explicitly need activation functions. However, activation functions, and especially their non-linear nature, are what enables our network to learn non-linear classification tasks. If we did not use any activation function (or a linear one), our network would break down to a very large and complex linear function. And although that might be useful, we would be constraint by the number of neurons and therefore variables in our linear model and using non-linear activation functions, converts the 'function' that our network computes (and tries to minimize its' error of) into a non-linear one which can classify way more complex datasets.

2.7 First model - 10.05% error rate

We have now acquired enough knowledge to look at our first model. Models in this case just refer to neural networks. All the models and the current codebase can be found in the GitHub repository used for this project.

Let us start by looking at the structure of the model first:

[3]Source of image: https://www.medcalc.org/manual/tanh-function.php

```
# input_shape=(1, 28*28) ; output_shape=(1, 100)
model.add_layer(FCLayer(28 * 28, 100))
model.add_layer(ActivationLayer(ActivationFunction.tanh))

# input_shape=(1, 100)   ; output_shape=(1, 50)
model.add_layer(FCLayer(100, 50))
model.add_layer(ActivationLayer(ActivationFunction.tanh))

# input_shape=(1, 50)    ; output_shape=(1, 10)
model.add_layer(FCLayer(50, 10))
model.add_layer(ActivationLayer(ActivationFunction.tanh))
```

Listing 3: First model

As we can see, we have three fully connected layers, with an activation layer immediately after every fully connected layer. This is at least what it looks like. But due to the nature of the implementation, one would actually consider this to be four layers. The input layer with 784 neurons, a second layer with 100 neurons, a third layer with 50 neurons and the output layer with 10 neurons. As discussed in the first section of this chapter, all our models with display the structure of an input layer with size 28×28 and an output layer with size 10.

The error function used for this model is the mean square error function, furthermore we used the stochastic gradient descent as discussed in section 2.2 and trained the model for 100 epochs with a fixed learning rate of 0.1. For the following models, we will simply list these hyperparameters without any further explication.

As discussed in 2.1, we evaluated the model using the 10,000 test images of the MNIST dataset, where the error rate is the percentage of images that were misclassified. Considering this is a first model without any optimizations, a 10% error rate seems fine. Nonetheless, will rapidly decrease this error rate throughout the course of this work.

3 Mini-batch gradient descent

So far, we used the gradient of the error function with respect to one sample from the training set (at a time) to back propagate and therefore train the network. This is useful, since many steps of stochastic gradient descent can be achieved in one epoch (to be specific, 60,000 in our case) and it still yields somewhat good results. To increase the accuracy of the gradient however, we can just use multiple training samples at the same time, i.e. mini-batches, to estimate the gradient of the error function with respect to the entire training set (the entire batch). Doing so, increases the accuracy of our (stochastic) gradient descent steps, while still maintaining the upside of having multiple updates every epoch.

While the upside is (stochastically) higher 'accuracy' of the gradients, the downside is of course the increase in computation. This however can be slightly mitigated, if the forward and backward propagation steps are simply computed at the same time for the entire batch, that is, using a vectorized implementation.

3.1 Second model - 06.75% error rate

Two changes were made from the first to the second model. For one, the number of layers was reduced to 3, and of course mini-batch gradient descent was used instead of stochastic gradient descent. Therefore, the structure of the second model looks like this:

```
# input_shape=(1, 28*28) ;   output_shape=(1, 128)
model.add_layer(FCLayer(28 * 28, 128))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 128))

# input_shape=(1, 128)   ;   output_shape=(1, 10)
model.add_layer(FCLayer(128, 10))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 10))
```

Listing 4: Second model

As can be seen, doing so, we were able to noticeably decrease our error rate by about 4%. This is probably largely due to using mini-batch gradient descent which, as discussed above, increased the 'accuracy' of our gradients.

4 Softmax and cross entropy loss

So far, we have used the hyperbolic tangent function as an activation function and the mean-squared error function to measure our error/loss per sample/mini-batch. There are, however, alternative functions that could be used for those tasks which are more prominent for solving classification problems. The first of which is softmax. An activation function, which maps the input to an output, that resembles a probability distribution, that is, having outputs in the interval of $(0, 1)$ and the values summing up to 1. This is very useful in classification problems, since a probability distribution as an output is easier to interpret than any real values.

There is however even more benefit to using softmax. Remember that we encoded our labels in so called one-hot encoding. That is, the label '0' is encoded as a vector of size

10, where each entry is a 0 except for the first entry. Coincidentally, this also resembles a probability distribution, and we can therefore use a loss function which 'measures the distance' between our output, that is, our estimated probability distribution, and the one-hot encoded label, that is, the actual probability distribution. The loss function in this case is called cross-entropy loss function. The details on why this function computes a distance between probability distributions go too far for this work, but a great article which goes slightly more in-depth can be found here.

For completeness, we will show the necessary formulas to compute softmax and cross-entropy loss, respectively. Since cross-entropy loss is in our case actually categorical cross-entropy loss, we will abbreviate it using CCE.

Definition 4.1 (Softmax)**.** The softmax function $\sigma_{softmax}$ is defined as:

$$\sigma_{softmax} : \mathbb{R}^n \to \mathbb{R}^n, x_i \mapsto \frac{e^{x_i}}{\sum_{i=0}^{n-1} e^{x_i}}$$

As can be seen from the definition of softmax, the sum of the entries of the resulting vectors will always be one, while the entries are in the interval $(0, 1]$.

Definition 4.2 (Cross-entropy loss)**.** Given an output of the network (prediction) $\hat{y}$ and the actual label of the image y in one-hot encoding, the mean-squared error is:

$$\sigma_{CCE}(\hat{y}, y) = -\sum_{i=0}^{\mathcal{C}-1} y_i \cdot \log \hat{y}$$

where $\mathcal{C}$ is the number of classes.

4.1 Third model - 03.10% error rate

By simple adding a softmax layer at the end of the network and changing choosing the cross-entropy loss function, we were able to drastically reduce the error rate once again to around 50% of the error rate of the second model. Since everything else stayed the same, we will only show the structure of the model.

```
# input_shape=(1, 28*28) ;   output_shape=(1, 128)
model.add_layer(FCLayer(28 * 28, 128))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 128))

# input_shape=(1, 128)   ;   output_shape=(1, 10)
model.add_layer(FCLayer(128, 10))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10))
```

Listing 5: Third model

5 Optimizers

In the first chapter 2, we discussed how neural networks train using stochastic gradient descent. There is, however, much to be desired from classic stochastic gradient descent. For one, if multiple 'steps' are taken into the same direction, why should the step length (learning rate) not be increased to reach the goal of minimizing the loss earlier. So-called optimizers try to solve this and other problems by adapting the learning rate throughout training. A very popular optimizer, which we will be using in this work, is Adam and was first published in 2014, see [KB17]. It is based previously existing optimizers and combines the idea of using momentum, that is, accumulate previous gradients to smooth out updates and accelerate convergence and RMSProp which adjusts the learning rate for each parameter based on the recent magnitudes of the gradients. Another advantage of Adam is its easy implementation, low memory requirements and light computational intensity. Basically, one vector keeps track of the previous gradients and another one of the squared gradients. These are then used to adjust the current parameters instead of adjusting them directly using the learning rate. Pseudocode for the algorithm and further explications are provided in the original paper, see [KB17].

Our implementation can be found at src/add_ons/optimizers.py in the repository containing the source code.

5.1 Fourth model - 02.64% error rate

Although the error rate decrease might seem as drastic, using the Adam optimizer (instead of no optimizer) we were able to drastically decrease training time, achieving a slightly lower error rate already after 30 epochs instead of 100 epochs for the previous third model, see 4.1. That of course implies that we changed nothing else than adding the Adam optimizer (and the learning rate to 0.01 instead of 0.1). Therefore, the structure of the model still looks as follows (the optimizer can be chosen on a per-layer basis). From now one, we will also show the hyperparameters used for the model.

```
# input_shape=(1, 28*28) ;   output_shape=(1, 128)
model.add_layer(FCLayer(28 * 28, 128, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 128))

# input_shape=(1, 128)   ;   output_shape=(1, 10)
model.add_layer(FCLayer(128, 10, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=30,
        learning_rate=0.01,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
)
```

Listing 6: Fourth model

6 Overfitting

A very prevalent issue in machine learning is overfitting. This can happen, if a neural network adapts so much/so well to the training data that it is not able to generalize to other data and achieves poor results on test data. There are different reasons why this can happen, but one way to reduce this effect is by 'not allowing' the network to overfit, that is, adapt too well to the training data. One way to do so is, to keep presenting the network with data that it has not seen before to prevent it from adjusting too much to certain data. This is of course easy if there is plenty of data. Another option is to artificially generate data or adjust the existing the data to create seemingly new data.

6.1 Dropout layers

Dropout layers randomly set inputs to 0 which can be interpreted as disabling the neurons that were responsible for this signal. This might in turn lead to the network not relying as heavily on specific neurons and being more robust as a whole. Also, since this alters the inputs of the next layers, it may lead to less overfitting.

More precisely, dropout layers have a parameter for the chance of dropping out an input signal. Then, given such a signal, they set a certain percentage $dropout_rate * 100\%$ of values of the input vector/matrix to 0 and rescale the rest by $\frac{1}{1-dropout_rate}$. When using the trained model to predict labels of some data, they simply pass the signal through and in back propagation, they set the values to 0 which correspond to the input values that were set to 0 in the forward propagation step.

The idea of using dropout layers as a method of reducing overfitting was first proposed in 2012, see [Hin+12], and has gained much traction since then.

Since the implementation is not very complicated, we only refer to the implementation in the projects' repository here instead of showing any code in this work.

6.2 Fifth model - 02.19% error rate

We are approaching an error rate where any slight improvement is a win, which is why the addition of dropout layers speaks for itself. Again, all other parameters except for the learning rate stayed the same. We again lowered the learning rate further from 0.01 to 0.001. The structure of the model looks therefore as follows.

```
# input_shape=(1, 28*28) ;   output_shape=(1, 128)
model.add_layer(FCLayer(28 * 28, 128, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 128))
model.add_layer(DropoutLayer(0.2, 128))

# input_shape=(1, 128)   ;   output_shape=(1, 10)
model.add_layer(FCLayer(128, 10, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=50,
        learning_rate=0.001,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
)
```

Listing 7: Fifth model

Note that the 0.2 parameter in the dropout layer is the dropout chance parameter (20%).

6.3 Sixth model - 02.02% error rate

Just by adding another fully connected layer and slightly altering the hyperparameters, we were able to drop to an error rate of almost exactly 2%. The listing shows the model setup.

```
# input_shape=(1, 28*28) ;   output_shape=(1, 128)
model.add_layer(FCLayer(28 * 28, 128, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 128))
model.add_layer(DropoutLayer(0.2, 128))

# input_shape=(1, 128)   ;   output_shape=(1, 50)
model.add_layer(FCLayer(128, 50, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 50))
model.add_layer(DropoutLayer(0.2, 50))

# input_shape=(1, 50)   ;   output_shape=(1, 10)
model.add_layer(FCLayer(128, 10, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=200,
        learning_rate=0.0005,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
)
```

Listing 8: Sixth model

6.4 Data Augmentation

Another method of preventing overfitting is artificially generating or altering the input data to the network. Of course, this has to be done in a way where the artificial training data still represents valid training data and their labels can be determined easily (for back propagation).

In our case of digit recognition or more generally image recognition, an easy approach is to linearly transform the training data to obtain slightly altered images, which of course still have the same labels. The paper [Cir+10] already applied this technique successfully to the MNIST dataset, achieving an error rate of 0.35%. We used similar parameters as they presented in their work, since increasing the distortions seemed to yield better results, especially with convolution networks as we will be using later. The only parameter that was varied for different models was the chance of even applying alteration to an input image. The exact alterations were, vertical and horizontal scaling from -20 to 20% of the height and width of the images respectively. Rotations from -25 to 25° and a zoom of -20 to 20%. Since the implementation is not of particular machine learning nature, we will only link to the source in the project repository, which can be found in $src/add_ons/data_augmentation.py$ in the repository of the project. Note that for some of the transformations, OpenCV or rather CV2 was used, see [Ope15].

6.5 Seventh model - 01.93% error rate

Applying the data augmentation as another method to prevent overfitting, we were able to just so slightly decrease the error rate to 01.93%. The models' structure is given in the following listing.

```
# input_shape=(1, 28*28) ;   output_shape=(1, 128)
model.add_layer(FCLayer(28 * 28, 128, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 128))
model.add_layer(DropoutLayer(0.2, 128))

# input_shape=(1, 128)   ;   output_shape=(1, 50)
model.add_layer(FCLayer(128, 50, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 50))
model.add_layer(DropoutLayer(0.2, 50))

# input_shape=(1, 50)   ;   output_shape=(1, 10)
model.add_layer(FCLayer(128, 10, optimizer=Optimizer.Adam))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=100,
        learning_rate=0.0005,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
        data_augmentation=DataAugmentation(chance_of_altering_data=0.25),
)
```

Listing 9: Seventh model

7 Early stopping

As described in [GBC16, p. 246], early stopping is at its core a technique to reduce overfitting or rather overtraining. It basically tracks the status of different performance parameters of the current parameters and possibly stops training if the performance parameters seem to only be getting worse after some time. More specifically, it often tracks the error on the validation set, validation sets being a technique we are not applying in this work. Basically, if the error on a second test set (the validation set) seems to increase, although the error on the training set is still decreasing, we suppose that this indicates overfitting and worse generalization of the model. In that case, we want to be able to load the parameters from when the model had a lower error on the validation set. This is what early stopping enables us. There is, however, still ongoing discussion on whether early stopping should always be applied with this setup. For example, the paper [HHS18, p. 9], suggests that good generalization can also result from an extensive amount of gradient updates in which there is no apparent validation error change and the training error continues to drop.

We, on the other hand, applied a more simple version of early stopping. Since we are not using validation sets, we are simply keeping track of the training set error (or simply, error) and store the parameters once the error stops decreasing. A hyperparameter called patience can be set, which sets the number of epochs with no decrease in the error, after which training is stopped. Another parameter called min delta rel can be set, which sets the minimum relative change which has to occur in the error for it to count as a decrease/change.

Since the implementation is as simple as it sounds, we shall not show any code for this feature and instead refer to the project repository at github.

8 Weight initialization

The Stanford course on Deep Learning for Computer Vision, see [Fei24, Neural Networks Part 2], introduces weight initialization by explaining what not to do: Initialize all weights with 0. In this case, the output for all neurons would be symmetrical and similarly the gradients would be the same for all neurons in back propagation. Therefore, we would have no source of asymmetry for our neurons and most of them would be redundant.

Another, more promising approach is, initializing the weights with random values. Since we normalized our data, we expect our weights to distribute evenly regarding negative and positive weights. This led to the weight initialization technique used so far: drawing from a normal distribution on the interval $[-0.5, 0.5]$. Turns out, this approach has other problems. With an increasing number of neurons, the variance increases and is unbounded. According to [Fei24], calibrating the variance in proportion to the number of neurons significantly improves the performance of networks. This resulted in a first approach of scaling the normal distribution (in the interval $[-1.0, 1.0]$) by, $\sqrt{\frac{1}{n}}$ since this leads to a variance of $\frac{1}{n}$. He refined this initialization strategy further for ReLu networks and suggested his 'He weight initialization' which uses $\frac{2}{n}$ to scale the distribution, see [He+15].

The above discussions only revolved around weight initialization and not bias initialization. He also discussed bias initialization, see [He+15], and came to the conclusion of initializing the biases with zero since the initialization of the weights will already break the symmetry in the initial biases during training.

We used the techniques presented by He et al. and its implementation can be found at src/add_ons/weight_initialization.py in the projects' repository.

8.1 Eight model - 01.58% error rate

Applying both early stopping and He weight initialization, we were able to significantly drop the error rate while also reducing the number of necessary epochs, since the training automatically stops when the training does not improve the performance of the model any further. Instead of the intended 175 epochs, the training stopped early on loading the weights from epoch 91. The models' structure is given in the following listing.

```
# input_shape=(1, 28*28) ;   output_shape=(1, 128)
model.add_layer(FCLayer(28 * 28, 128, optimizer=Optimizer.Adam,
                weight_initialization=WeightInitialization.he_bias_zero))
model.add_layer(ActivationLayer(ActivationFunction.tanh, 128))
model.add_layer(DropoutLayer(0.2, 128))

# input_shape=(1, 128)   ;   output_shape=(1, 50)
model.add_layer(FCLayer(128, 50, optimizer=Optimizer.Adam,
                weight_initialization=WeightInitialization.he_bias_zero))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 50))
model.add_layer(DropoutLayer(0.2, 50))

# input_shape=(1, 50)   ;   output_shape=(1, 10)
model.add_layer(FCLayer(128, 10, optimizer=Optimizer.Adam,
                weight_initialization=WeightInitialization.he_bias_zero))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=175,
        learning_rate=0.0005,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
        data_augmentation=DataAugmentation(chance_of_altering_data=0.25),
        early_stopping=EarlyStopping(patience=15, min_delta_rel=0.005),
)
```

Listing 10: Eight model

Note that for the following models we will omit the optimizer and weight initialization code for clarity, since Adam and He will be used for all following models, respectively.

9 Convolutional layers

The last big chapter of this work will be the following, in which we discuss convolutional layers and networks. The technique gained attraction in the 80s and has since been widely adopted for image recognition networks due to its similarities to the human brain in trying to learn and recognize certain patterns/features in images. The following sections notation and content will be based on the Stanford course on Deep Learning for Computer Vision, see [Fei24, Convolutional Networks].

Convolutional layers are quite similar to fully connected layers in the sense that they also have neurons which are connected to the previous and following layers and with these neurons come learnable weight and bias parameters which we try to optimize. Basically nothing else except the architecture of the specific layers changes, we still model a differentiable function with the network, have an error function, use activation layers etc..

To introduce convolution layers, let us first look at an example. Suppose, we have a dataset of images of size 200×200 and 3 color channels, that is $200 \cdot 200 \cdot 3 = 120,000$ neurons in the first layer of the network. The number of connections between the first and second layer is now $120,000 \cdot$ number of neurons in second layer. If we had even $1,000$ neurons in the second layer, we would already be at $120,000,000$ weights. This is of course quite huge. Convolutional layers can mitigate this huge increase in number of neurons when handling images by encoding certain properties into the architecture.

Instead of handling data samples that have height and width like before (28×28), we will now feed our network 3D (and layers) 3D volumes, that is data samples with height, width and depth. Coming back to the example before, the input of the network would be interpreted as having a height and width of 200 and a depth of 3. We will also refer to the depth of the network as the number of channels (color channels).

9.1 Convolutional layers

Let us first start by giving an intuitive description of what a convolution layer does. Each convolution layer has a number of filters and filter size. Suppose say we have a filter size of 5 and 16 of these filters. These filters will each slide across the input and apply the filter with current weights to every 5 by 5 patch of the input. From each of these 5 by 5 patches, the filter returns one value. All of these values then make up one of the input layers (channels) of the output of the convolution. Now to the intuition: the network weights for the filters that activate whenever a certain visual shape or feature is present. These shapes will typically be smaller for input layers and larger for layers that are later in the network.

Okay, the following is a list of the vocabulary/variables used so far:

- Number of channels (C): The depth of the inputs
- Height of input (H): The height of the inputs

- Width of the input (W): The width of the inputs
- Number of filters (NF): Number of filters
- Height of the filters (HF): Height of the filters (usually equals WF)
- Width of the filters (WF): Width of the filters (usually equals HF)

And lets introduce some new variables:

- Batch size (N): The size of the batches for mini-batch gradient descent
- Stride length (S): The step size for the filters when sliding across the input
- Padding (P): The number of added layers of zeros around the input

With regard to the new variables, we have already seen the batch size. The stride length is as described the steps we take on the image when applying the filters as described in the beginning. That is, given an input image of size 7×7 and filter dimensions 5×5 with a stride length of $S = 1$ (and padding $P = 0$), there are 9 possible 5×5 patches (all 5×5). If instead a stride length of $S = 2$ is used, there are only 4 possible patches (the corners). The padding can be used to keep the dimension of the input as output. With the last example of input 7×7 and stride length $S = 1$, there are 9 possible patches, which is why the output will be of width and height 3. If we instead added two layer of zeros around the input (that is, used padding $P = 2$), we would have 49 possible 5×5 patches, that is the output would be of size 7×7 which is the same as the input. This is also referred to as 'same' padding in major machine learning libraries.

We want to now briefly specify what exactly the filters do. We have already discussed that they shift over the input. For this, we will stick to the parameters $HF = WF = 5$, $S = 1$ and $P = 2 =' same'$. First, note that each filter has 5×5 many weights (times C actually, but we will see that in a second). When sliding across the input, as discussed before, these weights are then multiplied by the corresponding values in the 5×5 patch and these products are summed. The bias of the filter is then added to this value, and one obtains the value for entry in the output matrix for this filter. Now, if the input has more than one channel, this is done with C 5×5 weight matrices at the same time across the different input channels and their results are all summed up (with the bias). This process is visualized very well on the Stanford page, see [Fei24, Convolutional Networks]. We will only show a screenshot of the former, shown in the following figure.

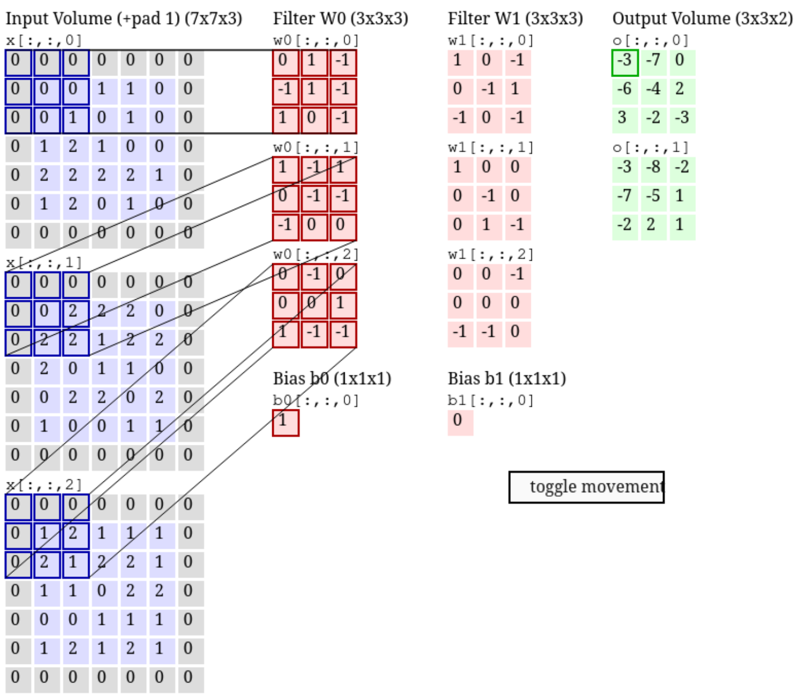


Figure 9.1: Convolution layer being applied with parameters $C = 3$, $H = W = 7$, $NF = 2$, $HF = WF = 3$, $N = 1$, $S = 2$ and $P = 1$[1]

To recap, our input will be of size $N \times C \times H \times W$, using the parameters as stated above, our output will have the dimensions $N \times NF \times H \times W$ where NF is the number of filters. This is because the height and width stay the same because of $S = 1$ and $P = 2 =' same'$. Furthermore, for each filter, we obtain an output matrix of size $H \times W$, which yields $NF \times H \times W$ output matrices (for each input sample). Since we now have N many input samples in the mini-batch, the output of the layer will be of size $N \times NF \times H \times W$.

Now, one might ask, why bother? One answer was already presented above. Large images just do not go well with fully connected layers, since too many neurons and therefore weights are required. For our setup, we need $NFxCxHFxWF$ many weights per layer. Note that this does not scale with the image size, and only with the channels (either color channels or number of filters for the previous convolutional layer). Another answer is of course because it works. As we will see, once the convolutional network was implemented correctly, we were able to drop the error rate quite significantly. Before we can look at the first model using convolutional layers, however, we need to discuss the other parts of a convolutional network.

Before doing so however, we want to quickly note that the backpropagation can be similarly derived to the one for fully connected and shall be left as an exercise for the

[1]Source of image: https://cs231n.github.io/convolutional-networks/

reader. Of course, our implementation can be found in the projects' repository here. Note, that a part of the implementation uses the im2col technique suggested by the Stanford course on Deep Learning for Computer Vision, see [Fei24] and is similar to their implementation provided for the exercises in their class.

9.2 Max pooling layers

Pooling layers and more specifically max pooling layers are used to reduce the dimensionality of the inputs. This reduces the computational cost and is an option to control overfitting by controlling the number of parameters in the model. Basically, a max pooling layer slides across the input like a filter from the section before and instead of computing some sum of products, we look at patches of size 2×2 (in our - and most - cases) and take the maximum entry of the 4 in the patch. Doing so, we can halve the height and width of the input (if stride $S = 2$).

Since there is not much to it and the vocabulary used is the same as the previous section, we shall only add that this is done for all channels in parallel in the sense that the pooling layer keeps the number of channels (and batch size) the same and only 'changes' the height and width of the input (with respect to the output). That is, with the parameters as discussed above, given an input of size $N \times C \times H \times W$, the output will be of size $N \times C \times \frac{H}{2} \times \frac{W}{2}$. Note that since our inputs have $H = W = 28$, we will add a maximum of two max pooling layers in the entire network. Furthermore, during backpropagation, we will, of course, only propagate the error back for those inputs that had the maximum value of the respective patches, since the other inputs and therefore neurons did not contribute to the output and therefore result.

9.3 Flatten layers

Flatten layers are layers used to convert the outputs of convolutional layers/blocks to valid inputs for fully connected layers. They basically convert the input of size $N \times C \times H \times W$ to an output of size, $N \times C \cdot H \cdot W$ that is, convert the matrix of each sample to a vector of the corresponding size. This is quite straightforward and the forward and backward propagation implementations should be clear. We need this because it is common to transform the output of a convolutional layer at the end to one that can be fed into a fully connected layer at the end to obtain an output of the network.

9.4 Network architecture

The Stanford course on Deep Learning for Computer Vision, see [Fei24, Convolutional Networks], also contains a section on convolution network architecture. We are basing the architecture of our convolutional networks on this. According to them, it is common practice to have a structure of the form:

$$INPUT- > [[CONV \rightarrow RELU]{\cdot}N \rightarrow POOL?]{\cdot}M \rightarrow [FC \rightarrow RELU]{\cdot}K \rightarrow FC \quad (9.1)$$

Note that the multiplication in this case indicates that blocks of layers can occur multiple times, whilst the ? indicates one or no layer of that type. Conv of course abbreviates convolution, whilst pool stands for pooling, and FC for fully connected layers. In our case, we will choose $N = 1$, $M = 2$, a (max) pooling layer after every convolution, ReLu block, and $K = 0$.

9.5 Ninth model - 00.80% error rate

This leads us to our first model using convolutional layers (that is, our first convolutional network).

Note that the optimizer and weight initialization (Adam and He) were omitted for code visibility. Furthermore, the convolution and max pooling layers have default parameters for the stride and padding, according to the explanations given above ($S = 1$ and $P = 2$ for the convolution layers and $S = 2$ and $P = 0$ for max pooling layers).

One can immediately see, by the error rate, that we were able to dramatically improve the performance of our models by using a convolutional network. Notably, the network already stopped training after 29 epochs due to early stopping.

```
# Block 1: input_shape=(BATCH_SIZE, 1, 28, 28) output_shape=(BATCH_SIZE, 8, 28, 28)
model.add_layer(Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=1, NF_number_of_filters=8,
                H_height_input=28, W_width_input=28))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2,
                C_number_channels=8, H_height_input=28, W_width_input=28))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 2: input_shape=(BATCH_SIZE, 8, 28, 28) output_shape=(BATCH_SIZE, 16, 14, 14)
model.add_layer(Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=8, NF_number_of_filters=16,
                H_height_input=14, W_width_input=14))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2,
                C_number_channels=16, H_height_input=14, W_width_input=14))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 3: input_shape=(BATCH_SIZE, 16, 7, 7) output_shape=(BATCH_SIZE, 16 * 7 * 7)
model.add_layer(FlattenLayer(D_batch_size=BATCH_SIZE, C_number_channels=16,
                H_height_input=7, W_width_input=7))

# Block 4: input_shape=(BATCH_SIZE, 128 * 7 * 7) output_shape=(BATCH_SIZE, 10)
model.add_layer(FCLayer(16 * 7 * 7, 10, convolutional_network=True))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10, convolutional_network=True))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=150,
        learning_rate=0.001,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
        data_augmentation=DataAugmentation(chance_of_altering_data=0.25),
        early_stopping=EarlyStopping(patience=20, min_delta_rel=0.005),
)
```

Listing 11: Ninth model

10 Learning rate schedulers

This will be the last technique added to our model, which was technically implemented before others due to its simplicity, however not used before. Learning rates are historically a heavily discussed hyperparameter due to its possibility to have a large impact on the training process and final performance of a model. So far, we have somewhat neglected the aspect of varying the learning rate during training by just using a constant learning rate, however this also resembles the experimentation process, since a fixed learning rate seemed to have worked fine so far, whilst the Adam optimizer has the side benefit of implicitly managing the learning rate.

Learning rate is such a crucial parameter, because it can stop or help a model while training. If the learning rate is too high, the gradient descent (variant) might be unable to further optimize the weights and biases because the steps that are taken are too big and therefore skip the local minima that could be achieved. If the learning rate is too low on the other hand, training takes very long and possibly learning might get stuck at a local, suboptimal minima. We show no graphs of the error rate over time in this work, since we based our learning rate tuning on the learning logs that are generated by our library. The logs for the best model can be found in the file best_result.log in our projects' repository.

We do not want to overcomplicate this, which is why we will only show one learning rate scheduler, since we ended up only using one. Due to its simplicity, we will just show its code:

```
def tunable_learning_rate_scheduler(
    learning_rate: float, epoch: int, halve_after: int
) -> float:
    if epoch < halve_after:
        if epoch == 0:
            return learning_rate
        return learning_rate + (learning_rate / epoch)
    if epoch % halve_after == 0 and epoch != 5 and epoch <= 20:
        return learning_rate * 0.5
    if epoch % 3 == 0 and epoch > 20:
        return learning_rate * 0.5
    return learning_rate
```

Listing 12: Tunable learning rate scheduler

As stated before, this is quite a simple scheduler. For the first 'halve_after' many epochs, it simply outputs the initial learning rate times the number of epochs so far. Then, it basically halves the learning rate after every 'halve_after' epochs. This is further intensified by halving after every 3 epochs after the 20th epoch. The latter was added only for the last (twelfth) model and will be discussed later. For the exact corresponding implementations for each model, please check the projects' repository on GitHub.

10.1 Tenth model - 00.44% error rate

Applying this learning rate scheduler with $halve\_after = 5$, we obtain a new model with an astounding 0.44% error rate. Additionally, we slightly bumped up the chance of altering data for the data augmentation to 0.5 since we suspected some overfitting to be going on with the model. And since this seemed to work, we kept this change for the following models.

Note that the optimizer and weight initialization (Adam and He) were omitted for code visibility. Furthermore, the convolution and max pooling layers have default parameters for the stride and padding, according to the explanations given above ($S = 1$ and $P = 2$ for the convolution layers and $S = 2$ and $P = 0$ for max pooling layers).

With this change, our model ran sightly longer, that is 48 epochs before early stopping kicked in. The following listing shows the networks' structure.

```
# Block 1: input_shape=(BATCH_SIZE, 1, 28, 28) output_shape=(BATCH_SIZE, 8, 28, 28)
model.add_layer(Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=1, NF_number_of_filters=8,
                H_height_input=28, W_width_input=28))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2,
                C_number_channels=8, H_height_input=28, W_width_input=28))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 2: input_shape=(BATCH_SIZE, 8, 28, 28) output_shape=(BATCH_SIZE, 16, 14, 14)
model.add_layer(Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=8, NF_number_of_filters=16,
                H_height_input=14, W_width_input=14))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2,
                C_number_channels=16, H_height_input=14, W_width_input=14))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 3: input_shape=(BATCH_SIZE, 16, 7, 7) output_shape=(BATCH_SIZE, 16 * 7 * 7)
model.add_layer(FlattenLayer(D_batch_size=BATCH_SIZE, C_number_channels=16,
                H_height_input=7, W_width_input=7))

# Block 4: input_shape=(BATCH_SIZE, 128 * 7 * 7) output_shape=(BATCH_SIZE, 10)
model.add_layer(FCLayer(16 * 7 * 7, 10, convolutional_network=True))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10, convolutional_network=True))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=150,
        learning_rate=0.001,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
        data_augmentation=DataAugmentation(chance_of_altering_data=0.5),
        early_stopping=EarlyStopping(patience=25, min_delta_rel=0.005),
        learning_rate_scheduler=LearningRateScheduler.tunable,
        learning_rate_halve_after=5,
)
```

Listing 13: Tenth model

10.2 Eleventh model - 00.42% error rate

From now on, it was only a matter of scaling up the model and tuning the hyperparameters. For this model, another convolution layer (block) was added with 64 filters, which resulted in a tiny performance increase. The following listing shows the resulting models structure.

```
# Block 1: input_shape=(BATCH_SIZE, 1, 28, 28) output_shape=(BATCH_SIZE, 16, 14, 14)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=1, NF_number_of_filters=16, H_height_input=28,
                  W_width_input=28, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 2: input_shape=(BATCH_SIZE, 16, 28, 28) output_shape=(BATCH_SIZE, 32, 7, 7)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=16, NF_number_of_filters=32, H_height_input=28,
                  W_width_input=28, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2, C_number_channels=32,
                                  H_height_input=28, W_width_input=28))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 3: input_shape=(BATCH_SIZE, 32, 14, 14) output_shape=(BATCH_SIZE, 48, 14, 14)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=32, NF_number_of_filters=48, H_height_input=14,
                  W_width_input=14, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 4: input_shape=(BATCH_SIZE, 48, 14, 14) output_shape=(BATCH_SIZE, 64, 7, 7)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=48, NF_number_of_filters=64, H_height_input=14,
                  W_width_input=14, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2, C_number_channels=64,
                                  H_height_input=14, W_width_input=14))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 5: input_shape=(BATCH_SIZE, 64, 7, 7) output_shape=(BATCH_SIZE, 64 * 7 * 7)
model.add_layer(FlattenLayer(D_batch_size=BATCH_SIZE, C_number_channels=64, H_height_input=7, W_width_input=7))

# Block 6: input_shape=(BATCH_SIZE, 64 * 7 * 7) output_shape=(BATCH_SIZE, 10)
model.add_layer(FCLayer(64 * 7 * 7, 10, optimizer=optimizer, convolutional_network=True))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10, convolutional_network=True))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=150,
        learning_rate=0.001,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
        data_augmentation=DataAugmentation(chance_of_altering_data=0.5),
        early_stopping=EarlyStopping(patience=25, min_delta_rel=0.005),
        learning_rate_scheduler=LearningRateScheduler.tunable,
        learning_rate_halve_after=5,
)
```

Listing 14: Eleventh model

10.3 Twelfth model - 00.40% error rate

Our last models' idea comes from/came from looking at the training log of the eleventh model, see here. Looking at epochs 15 and onwards, it seemed like every 5 epochs, when the learning rate was halved, a bump down in error was able to be down and then the training had to 'wait' another five epochs before being able to drop further in error because the learning rate was too high, and the error was jumping around. Therefore, we added an option in the tunable learning rate scheduler to halve the learning rate every 3 epochs after the 20th epoch. Doing so increased the training time for (from 94, down to 56 epochs). We also changed the chance of altering data for the data augmentation to, 0.8 and both changes ended up giving us an error rate of 00.40%. The resulting network structure is shown in the following listing.

```
# Block 1: input_shape=(BATCH_SIZE, 1, 28, 28) output_shape=(BATCH_SIZE, 16, 14, 14)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=1, NF_number_of_filters=16, H_height_input=28,
                  W_width_input=28, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 2: input_shape=(BATCH_SIZE, 16, 28, 28) output_shape=(BATCH_SIZE, 32, 7, 7)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=16, NF_number_of_filters=32, H_height_input=28,
                  W_width_input=28, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2, C_number_channels=32,
                                  H_height_input=28, W_width_input=28))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 3: input_shape=(BATCH_SIZE, 32, 14, 14) output_shape=(BATCH_SIZE, 48, 14, 14)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=32, NF_number_of_filters=48, H_height_input=14,
                  W_width_input=14, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 4: input_shape=(BATCH_SIZE, 48, 14, 14) output_shape=(BATCH_SIZE, 64, 7, 7)
model.add_layer(
    Convolution2D(D_batch_size=BATCH_SIZE, C_number_channels=48, NF_number_of_filters=64, H_height_input=14,
                  W_width_input=14, optimizer=optimizer))
model.add_layer(ActivationLayer(ActivationFunction.ReLu, 0, convolutional_network=True))
model.add_layer(MaxPoolingLayer2D(D_batch_size=BATCH_SIZE, PS_pool_size=2, S_stride=2, C_number_channels=64,
                                  H_height_input=14, W_width_input=14))
model.add_layer(DropoutLayer(0.2, 0, convolutional_network=True))

# Block 5: input_shape=(BATCH_SIZE, 64, 7, 7) output_shape=(BATCH_SIZE, 64 * 7 * 7)
model.add_layer(FlattenLayer(D_batch_size=BATCH_SIZE, C_number_channels=64, H_height_input=7, W_width_input=7))

# Block 6: input_shape=(BATCH_SIZE, 64 * 7 * 7) output_shape=(BATCH_SIZE, 10)
model.add_layer(FCLayer(64 * 7 * 7, 10, optimizer=optimizer, convolutional_network=True))
model.add_layer(ActivationLayer(ActivationFunction.softmax, 10, convolutional_network=True))

# Set (hyper)parameters
model.set_hyperparameters(
        epochs=150,
        learning_rate=0.001,
        learning_rate_scheduler=LearningRateScheduler.const,
        batch_size=16,
        data_augmentation=DataAugmentation(chance_of_altering_data=0.5),
        early_stopping=EarlyStopping(patience=25, min_delta_rel=0.005),
        learning_rate_scheduler=LearningRateScheduler.tunable,
        learning_rate_halve_after=5,
)
```

Listing 15: Twelfth model

This is of course not the desired 00.30% error rate. This is however probably not directly due to the implementation, more on that in 12, but rather due to there not being enough filters in the convolution layers. This will lead us to the neural network using TensorFlow, see [Aba+15], which is basically the same as the one in the listing above, just with $32, 64, 96$ and 128 filters instead of $16, 32, 48$ and 64.

The details on why we were not able to construct such a model using our implementation comes down to performance. Or rather, GPU and CPU compatibility. Our implementation is based solely on NumPy and runs therefore on the CPU. TensorFlow, on the other hand, is highly optimized and runs virtually on any GPU. Therefore, the following model will be one not using our implementation of neural networks and instead TensorFlow, but only due to the limitations of not being able to train a network for 4 days straight. Note that the above network took 20 hours to train.

11 Tensorflow model - 00.25% error rate

As mentioned at the end of the previous chapter, in section 10.3, the following will be a model using the common machine learning library TensorFlow, see [Aba+15]. It will not use any features that we did not also implement, but instead run faster. TensorFlow is written for being able to run on either the CPU or GPU, the latter being way more suited for a task like this.

The basic structure of the model, using the TensorFlow library is shown in the following listing. Note, that the data augmentation and so-called 'callbacks', i.e. learning rate schedulers, are initialized/configured differently than in TensorFlow than in our library, so for the details, see tensorflow_model.ipynb.

```
model = tf.keras.models.Sequential()

# Block 1 - input_shape=(BATCH_SIZE, 1, 28, 28) output_shape=(BATCH_SIZE, 32, 28, 28)
model.add(layers.Conv2D(32, (5, 5), padding="same", input_shape=(28, 28, 1)))
model.add(layers.ReLU())
model.add(layers.Dropout(0.2))

# Block 2 - input_shape=(BATCH_SIZE, 32, 28, 28) output_shape=(BATCH_SIZE, 64, 14, 14)
model.add(layers.Conv2D(64, (5, 5), padding="same"))
model.add(layers.ReLU())
model.add(layers.MaxPooling2D(pool_size=(2, 2)))
model.add(layers.Dropout(0.2))

# Block 3 - input_shape=(BATCH_SIZE, 64, 14, 14) output_shape=(BATCH_SIZE, 96, 14, 14)
model.add(layers.Conv2D(96, (5, 5), padding="same"))
model.add(layers.ReLU())
model.add(layers.Dropout(0.2))

# Block 4 - input_shape=(BATCH_SIZE, 96, 14, 14) output_shape=(BATCH_SIZE, 128, 7, 7)
model.add(layers.Conv2D(128, (5, 5), padding="same"))
model.add(layers.ReLU())
model.add(layers.MaxPooling2D(pool_size=(2, 2)))
model.add(layers.Dropout(0.2))

# Block 5 - input_shape=(BATCH_SIZE, 128, 7, 7) output_shape=(BATCH_SIZE, 128 * 7 * 7)
model.add(layers.Flatten())

# Block 5 - input_shape=(BATCH_SIZE, 128 * 7 * 7) output_shape=(BATCH_SIZE, 10)
model.add(layers.Dense(10))
model.add(layers.BatchNormalization())
model.add(layers.Dense(10, activation='softmax'))

model.compile(optimizer=optimizers.Adam(learning_rate=0.0005), loss='categorical_crossentropy',
            metrics=['accuracy'])
```

Listing 16: Tensorflow model

As discussed before, this model differs from model twelfth basically only in the number of filters used for the four convolution layers.

This concludes our development of new models, and we shall finish by briefly discussing how we tested our code.

12 Checking if the implementation works

The Stanford course on Deep Learning for Computer Vision, see [Fei24, Learning and Evaluation], has a chapter on grad checking, that is, gradient checking and other useful tips when implementing a neural network. The problem being that, the variables and exact inputs and outputs of the different layers can be very obscure due to the random initialization of the weights and biases. Therefore, debugging by stepping through the code step-by-step is near impossible. Instead, we used an existing machine learning library, PyTorch, see [Ans+24] to check our computed gradients against. More precisely, we created the same model using PyTorch and our code, initialized the weights and biases to be the same and simultaneously forward and backward propagated the same input, checking the results to be the same after every step. The implementation of this can be found in src/tests/layers/test_convolution_network_propagation.py in the projects' repository and is run automatically after every push using GitHub Actions.

Bibliography

[Aba+15] Martín Abadi et al. *TensorFlow, Large-scale machine learning on heterogeneous systems.* Nov. 2015. DOI: 10.5281/zenodo.4724125.

[Ans+24] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24).* ACM, Apr. 2024. DOI: 10.1145/3620665.3640366.

[Cir+10] Dan Claudiu Cire·*sanEtal.*. "Deep, Big, Simple Neural Nets for Handwritten Digit Recognition". In: *Neural Computation* 22.12 (Dec. 2010), pp. 3207–3220. ISSN: 1530-888X. DOI: 10.1162/neco_a_00052.

[Fei24] Ehsan Adeli Fei-Fei Li. *CS231n: Deep Learning for Computer Vision.* last visited on 08.12.2024. 2024. URL: https://cs231n.stanford.edu/.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics.* Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 315–323.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* http://www.deeplearningbook.org. MIT Press, 2016.

[He+15] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* 2015. arXiv: 1502.01852 [cs.CV].

[HHS18] Elad Hoffer, Itay Hubara, and Daniel Soudry. *Train longer, generalize better: closing the generalization gap in large batch training of neural networks.* 2018. arXiv: 1705.08741 [stat.ML].

[Hin+12] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors.* 2012. arXiv: 1207.0580 [cs.NE].

[KB17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: 1412.6980 [cs.LG].

[Ope15] OpenCV. *Open Source Computer Vision Library.* 2015.

[San17] Grant Sanderson. *Neural Networks.* last visited on 02.12.2024. 2017. URL: https://www.3blue1brown.com/topics/neural-networks.

[Wik24] Wikipedia. *Neuron.* last visited on 02.12.2024. 2024. URL: https://en.wikipedia.org/wiki/Neuron.

List of Figures

Listings