Image classification on CIFAR-10

using basic Feed-Forward Neural Networks

# Digit recognition example (lecture)

The example provided in the course material is a very simple neural network [NN], trained to recognize handwritten digits. The MNIST database is a subset of the much larger Special Databases 1 and 3 provided by NIST (The National Institute for Standards and Technology).

This example will be explained in detail in the following sections, starting with the tools that are being used. The code, copied from the lecture slides, is presented, and then explained.

## Requirements

This section discusses the tools and resources necessary to create the example NN. The contents of this section also apply to later sections discussing the creation of a simple NN using the same tools.

### Keras

The example uses the Keras API, a tool with the focus to let researchers quickly build deep learning models for fast experimentation.

Keras uses layers and models in an object-oriented manner, where the model is the object which contains the layers. Keras provides simple and more complex models to fit the needs of the researcher. Apart from the layers, which have their own set of parameters, it is also possible to adjust optimizer, loss function and hyper parameters.

Keras is a high-level API of TensorFlow 2 and can be run on many kinds of systems, from GPU clusters to browser or mobile devices.[[1]](#footnote-1)

### TensorFlow (2)

TensorFlow is a machine learning platform, developed and maintained by Google employees, and open source. It provides the infrastructure to calculate with multi-dimensional arrays, which are also called tensors. It automatically finds the gradient of any differentiable array. In contrast to other array calculation APIs, TensorFlow can take advantage of GPUs and distribute calculations over multiple machines.[[2]](#footnote-2)[[3]](#footnote-3)

### MNIST

As described in the beginning, MNIST is a database containing handwritten digits. It is a good benchmark for the performance of a machine learning algorithm (in OCR) and is small enough to be also manageable by weaker machines.

The images of the digits are 28 by 28 pixels, each having a grey scale value of 8 bit. This constitutes a 3-dimensial vector, two dimensions for the position of the pixel and one for the greyscale vector.

There are 70,000 of these images in the dataset, 60,000 of them are the training set and 10,000 are the test set. The writing style of the digits varies widely from clean office writing to messy writing of high-school members.[[4]](#footnote-4)

## Description

This section is meant to describe the programming aspect of the example exercise. The Keras developer documentation[[5]](#footnote-5) is used to help better understanding the code at hand.

1. import tensorflow as tf
2. from tensorflow.keras import models
3. from tensorflow.keras import layers
4. from tensorflow.keras.datasets import mnist

The lines 1 to 4 import the TensorFlow package, two APIs from Keras, namely models and layers, and the MNIST dataset. The models   
API contains NN model classes, representing the base structure of the neural network. The layers API contains the layer class, building blocks of Keras neural networks which will be contained in the model.

1. (train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

The function ‘load\_data’ contained in the built-in dataset returns a tuple of NumPy arrays. The image arrays contain integer values of the pixels with values from 0 to 255. The labels arrays are one dimensional and carry the actual digit labels that accords to the collection of pixels in the images array.

1. network = models.Sequential()

Here, the most simple and basic NN model class is instantiated and stored in a variable called ‘network’. The sequential model can only feed forward, meaning there are no loops in its design, and each layer that is contained can only accept one array (tensor) as input and one array as output.

1. network.add(layers.Dense(512, activation='relu', input\_shape=(28 \* 28,)))
2. network.add(layers.Dense(10, activation='softmax'))

In the following two lines, two new layers are added to the empty sequential model. Both are of the Dense class. A dense layer is the most basic kind, all its neurons will be connected to all neurons in the next layer, therefore the name fully connected layer. The use of these layers is useful for smaller networks, where it approximates the function well, but becomes computationally heavy for bigger networks.

In line seven, the first parameter of the layer is the unit number, in other words, the number of neurons contained in the layer. In this case it is set to 512. The second parameter is the activation function, set to the ReLU (rectified linear unit) function. This function simply returns the maximum value between one value of the input tensor and 0. The last parameter can only be set on the very first layer in the model. The input shape determines the size of the tensor, it is set to 28 times 28 (782) as these are the height and width position values of a single image that needs to be processed. Further down in the code (line 10) we can see how these 2-dimensional image vectors are being flattened to 1-dimensional tensors.

The second layer in line 8 is also the output layer, thus we have no hidden layers in this model. It is also a dense layer, but with a unit number of 10. This time the unit number is set to the number of desired categories, namely the 10 distinct digits, 0 to 9. The activation function in this case is called SoftMax. SoftMax is used to transform the values of a vector into a probability distribution. This is useful for the last layer as the result is indeed a probabilistic decision.

1. network.compile(optimizer='rmsprop', loss='categorical\_crossentropy',metrics=['accuracy'])

In line 9 the model is configured for training, this is done after all layers have been defined, after this step the model is ready for training. The parameters set here are the optimizer, the loss function and a list of measurements that should be taken while training, in this case the accuracy of the model. The chosen optimizer is called RMSprop, an algorithm that keeps track of a discounted average of squared gradients and divides the gradient by the root of that average. The loss function here is a categorical cross entropy, computing the loss between labels and predictions. This function is applied in case where the result can belong to 2 or more categories. The categories in this case are of course the digits from 0 to 9.

1. train\_images = train\_images.reshape((60000, 28 \* 28))
2. train\_images = train\_images.astype('float32') / 255
3. test\_images = test\_images.reshape((10000, 28 \* 28))
4. test\_images = test\_images.astype('float32') / 255

In the lines 10 to 13 the training and testing image arrays are converted from a number of 2-dimensional arrays to the same number of 1-dimensional arrays. The pixel value is no longer on a grid representing the x and y axes, but each value is numbered from 0 to 784. The conversion from integer to float is necessary for the model to be able to work with the values. Lastly, the division by 255 normalizes the pixel value, which can range from 0 to 255, to a value ranging between 0 to 1. Large values are known to cause issues during training, such as over-saturated neurons.

1. from tensorflow.keras.utils import to\_categorical
2. train\_labels = to\_categorical(train\_labels)
3. test\_labels = to\_categorical(test\_labels)

In the lines 14 to 16, a function is imported that converts NumPy vectors with integers into a binary class matrix. This function is applied to both training and testing labels. Thus, the labels do not represent anymore a simple array of digit values, but a matrix filled with 0’s and 1’s where a 1 means that the category applies for the according image. This conversion is necessary when using the categorical cross entropy loss function.

1. network.fit(train\_images, train\_labels, epochs=5, batch\_size=128)

In this step we start training the model for the training images. The parameters that have been given are the number of epochs, which is 5 in this example, saying the model should reiterate the 60000 images 5 times, and the batch size determining how many samples should be taken per gradient update. The default batch size is 32 but has been overwritten to 128.

1. test\_loss, test\_acc = network.evaluate(test\_images, test\_labels)

In line 18, the model is now tested with the test images. The variables that are assigned in this process are test loss, which is the loss value, and the testing accuracy, telling how accurately the model has guessed the digit. The loss is a sum of the errors the model made, the accuracy is the percent value of correct guesses to incorrect guesses.

> print(test\_loss)

52.34154510498047

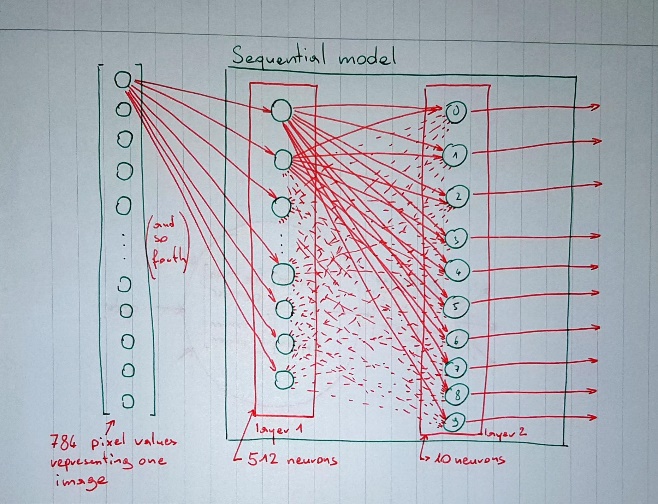
> print(test\_acc)

0.8180000185966492

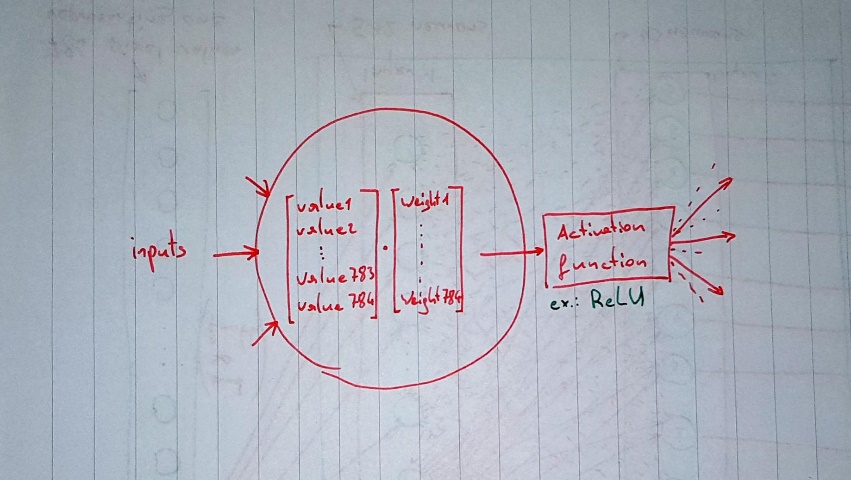
Here we see the values that have been attained. The accuracy is at ~82%, which could surely be improved.

## Explanation

Now, this explanation of the code does not fully explain what has happened. A training set and testing set have been defined, a model has been chosen and layers have been added, then the model has been trained and tested and the accuracy value has been recorded. But what happens within the neural network and how the given parameters influence the process are yet to be explained.



This previous drawing represents the structure of the neural network created in the example.



In a close-up to one single neuron, we see how the information is processed within the model.

In a dense layer, the dot product is taken of the input values and the weights that are stored in each neuron. It is the weights that are adapted during the training and are modified by the optimization procedure. How those weights are adjusted depends on the optimizer. In this case, the RMSprop optimization compares the sign of the gradients to determine if, in the gradient descend algorithm, local minima are jumped over because of a too large step size or if the correct direction is taken and the step size could be increased. Furthermore, by dividing the learning rate by the root of the squared mean gradient, the magnitudes of the gradients can be balanced out.

Afterwards, the activation function is applied on the results from the dot product. In the first case, the ReLU simply returns the value if it is >0 and else returns 0. The SoftMax function on the other hand turns an array of real values into an array of values that sum to 1. This means that the sum of the 10 output values is a hundred percent, where one of the outputs will probably have a much higher value, being thus the guess the network makes. It can be seen as a normalization procedure.

A last discussion point might be the unit number that is defined per layer. But other than the output layer, whose unit number must meet the number of the categories we want to train the model for, that unit number can be chosen arbitrarily and must be adjusted through trial and error. In this sequential model, it seems to make sense that we go from a higher number to a lower one, considering the big number of pixels that is reduced to one in 10 possibilities.

# The Cifar-10 dataset

## Description

## Loading and inspection

## (possible) Preprocessing

## Statistics and Graphs

# Implementation of basic NN

## Model design

* 2 categories
* 1 in, 1 out
* extendable

## Evaluation of first results

# Adaptations

## 5 Categories

## 10 Categories

## Evaluation of performances

# Improvements

## Adding and changing Layers

## Hyperparameter adjustments

## Explanation of most note-worthy improvements

# Final Performance evaluation

## 2 Categories

## 5 Categories

## 10 Categories

.

# References

# Annex

1. About Keras, <https://keras.io/about/> [↑](#footnote-ref-1)
2. <https://www.tensorflow.org/about> [↑](#footnote-ref-2)
3. <https://keras.io/getting_started/intro_to_keras_for_researchers/> [↑](#footnote-ref-3)
4. <http://yann.lecun.com/exdb/mnist/> [↑](#footnote-ref-4)
5. <https://keras.io/api/> [↑](#footnote-ref-5)