# Image classification on CIFAR-10

using basic Feed-Forward Neural Networks

## 1 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.

### 1.1 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.

#### 1.1.1 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]

#### 1.1.2 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][3]

#### 1.1.3 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]

### 1.2 Description

This section is meant to describe the programming aspect of the example exercise. The

[1] About Keras, https://keras.io/about/
[2] https://www.tensorflow.org/about
[3] https://keras.io/getting_started/intro_to_keras_for_researchers/
[4] http://yann.lecun.com/exdb/mnist/

Keras developer documentation[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.

```
[5] (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.

```
[6] 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.

```
[7] network.add(layers.Dense(512,
    activation='relu', input_shape=(28 *
    28,)))
[8] 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.

```
[9] network.compile(optimizer='rmsprop',
    loss='categorical_crossentropy',metr
    ics=['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

[5] https://keras.io/api/

categories. The categories in this case are of course the digits from 0 to 9.

```
[10] train_images =
     train_images.reshape((60000, 28 *
     28))
[11] train_images =
     train_images.astype('float32') / 255
[12] test_images =
     test_images.reshape((10000, 28 *
     28))
[13] 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.

```
[14] from tensorflow.keras.utils import
     to_categorical
[15] train_labels =
     to_categorical(train_labels)
[16] 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.

```
[17] 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.

```
[18] 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.

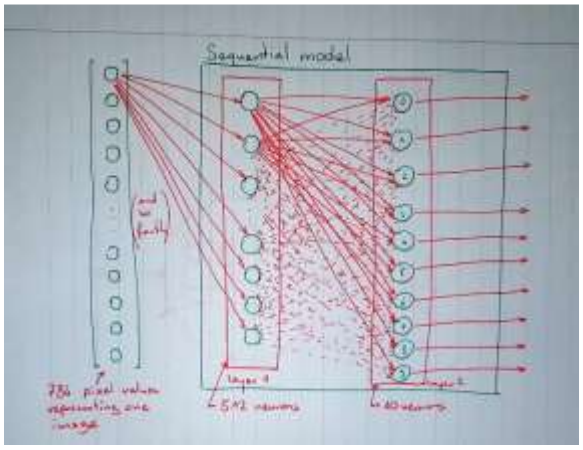
```
> print(test_loss)
52.34154510498047

> print(test_acc)
0.8180000185966492
```

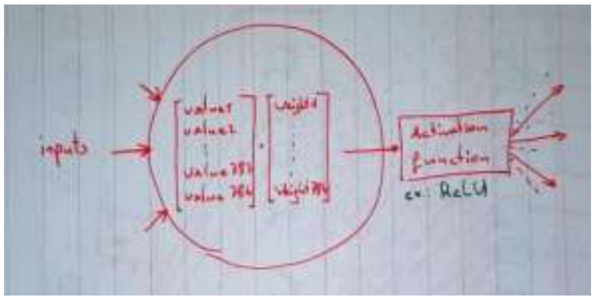
Here we see the values that have been attained. The accuracy is at ~82%, which could surely be improved.

## 1.3 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.

# 2 The Cifar-10 dataset

## 2.1 Description

As stated by the homepage of the Cifar-10 dataset, hosted by the university of Toronto and provided by Alex Krizhevsky, the dataset contains 60000 images, divided into 10 classes. Each class consists of 6000 32 by 32 pixel images. The training set has 50000 images, leaving a test set of 10000 images. There are 3 versions of the dataset available to download, a Python and a MATLAB pickle and a binary version of the dataset to use in C programs. For this project, the Python version of the dataset is chosen. In the following section the unpickling process and initial inspection of the dataset is described.

## 2.2 Loading and inspection

The Cifar-10 data downloaded from its homepage is split in 5 batches of training images and one batch of testing images, containing 10000 images each. To acquire the contents of a batch, the file needs to be unpickled first. Pickling is a common procedure to save an object in Python as a binary file. The function to unpickle the data is for convenience taken from the homepage of the dataset and looks as follows.

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo,
encoding='bytes')
    return dict
```

This function returns a dictionary with 4 keys, the first being simply a string stating the batch label.

```
>print(data_batch_1[b'batch_label'])
b'training batch 1 of 5'
```

The 'b' in front of the string means that it is encoded as bytes, which can be read from the code copied from the homepage.

The next item in the dictionary is called 'labels', a list of the category labels that belong to the images. The dictionary item 'data' contains the image data in the same order than the labels are in.

```
>print(len(data_batch_1[b'data']))
10000

>print(len(data_batch_1[b'data'][0]))
3072
```

The length of the ‘data’ item is 10000, being the number of images contained. The homepage states this item is a NumPy array, and each row of the array contains one image. Looking at the size of one such row, it is found that it contains 3072 values. The homepage explains that these are the values of the 3 color channels, 1024 values for each the red, green and blue channel, in this order. The color channel values are integers and range from 0 to 255.

The last item in the dictionary is called ‘filenames’, a list of the image file names, occurring too in the same order than the image vectors.

## 2.3 Preprocessing

The first change that needs to be made is to combine the 5 batches of training imagery to one big dataset. During this, the batch label and the image file names can be removed as they do not provide any useful information for the training of a neural network. The data and labels will also be stored in a similar fashion as in the example in the first chapter of this document.

In a further step, the color values of the images need to be converted into floating point numbers such that the NN can operate on them. The values, will also, just like in the example exercise, be normalized to a value between 0 and 1 by dividing it by 255.

Then, though it not explicitly being part of the data processing, a function will be written to retrieve only images of given categories. This will help in the later parts of this project, when only 2 or 5 of the categories should be trained on.

## 2.4 Statistics

After processing the data, the number of entries in all batches combined for each category can be summed by the value_counts() function of the Pandas API. The result looks as follows for the training labels array.

```
>train_labels.value_counts()
9    5000
8    5000
7    5000
6    5000
5    5000
4    5000
3    5000
2    5000
1    5000
0    5000

>test_labels.value_counts()

7    1000
6    1000
5    1000
4    1000
3    1000
2    1000
9    1000
1    1000
8    1000
0    1000
```

It can be said that the training set and the testing set are perfectly balanced.

When trying to check if the training and testing sets are mutually exclusive, the antique system hardware the script was run on experienced a sudden low blood pressure and refused to terminate. The script should theoretically work but is of cubic complexity. As there were no results after hours, but also no statements in the console of such findings, it was concluded that the image sets are (at least mostly) mutually exclusive. Also because the image sets were compiled by well-versed engineers employed at Google and/or the University of Toronto.

# 3 Implementation of basic NN

## 3.1 Model design

The two categories chosen for this exercise are the sets of images of airplanes and dogs. It seems very likely that a neural network would be able to better learn the difference between a machine and an animal, than between two animals of similar stature, like dogs and cats.

With only one input layer and one output layer, the model will be quite similar to the digit recognition NN. Especially if no hyperparameters should be changed. To make the same model adaptable to the maximum of 10 image categories, the output layer contains 10 neurons. Furthermore, the training and testing label arrays must be transformed in a binary matrix that recognizes 10 categories. For this example with only 2 categories, it means that 8 of 10 columns of this matrix is filled with only zeros.

```
[1] train_lbls =
    to_categorical(train_lbls,
    num_classes=10)
[2] test_lbls =
    to_categorical(test_lbls,
    num_classes=10)
```

The full code of this version of the code can be found on GitHub, tagged with ‘v0.1’, or using the commit hash 9b1ad bcb2e bbae3 4d4bd 7279d cd739 6fae4 839b5.

## 3.2 Evaluation of first results

This section is meant to quickly go over the results of the training and testing process of the very simple NN that has been created in the previous section.

```
>network.fit(train_imgs, train_lbls,
epochs=5, batch_size=128)

Epoch 1/5
79/79 [==============================]
- 5s 47ms/step - loss: 4.8176 -
accuracy: 0.5633
Epoch 2/5
79/79 [==============================]
- 4s 45ms/step - loss: 1.3566 -
accuracy: 0.6982
Epoch 3/5
79/79 [==============================]
- 4s 45ms/step - loss: 0.7358 -
accuracy: 0.7130
Epoch 4/5
79/79 [==============================]
- 4s 45ms/step - loss: 0.5275 -
accuracy: 0.7546
Epoch 5/5
79/79 [==============================]
- 4s 45ms/step - loss: 0.4810 -
accuracy: 0.7708
```

Within the training process, the NN already achieved a 70% accuracy after 2 epochs, which is better than random guessing. The final accuracy in the training phase is 77%. This percentage is far from optimal, but it certainly shows that the NN can already distinguish planes from dogs.

```
>test_loss, test_acc =
network.evaluate(test_imgs, test_lbls)

63/63 [==============================]
- 1s 7ms/step - loss: 0.4036 -
accuracy: 0.8155
```

When the NN is run on the testing images, it even achieves a higher accuracy than during training, 81 percent. A great result for not having adjusted any parameters or changed number of layers and neuron counts.

# 4 Adaptations

This chapter describes the process of adapting the previously created simple NN, such that it achieves a good accuracy on 5 and 10 distinct categories. First, the simple NN is tested as it is to see the baseline results, then parameters are changed, and layers added.

## 4.1 Training with simple NN

### 4.1.1 5 Categories

When the simple neural network is run with 5 categories, a serious drop in accuracy can be observed. The accuracy at the end of the training is only at 50%.

```
196/196
[==============================] - 9s
44ms/step - loss: 5.2150 - accuracy:
0.2745
Epoch 2/5
196/196
[==============================] - 9s
45ms/step - loss: 1.5515 - accuracy:
0.3735
Epoch 3/5
196/196
[==============================] - 9s
45ms/step - loss: 1.3210 - accuracy:
0.4313
Epoch 4/5
196/196
[==============================] - 9s
46ms/step - loss: 1.2297 - accuracy:
0.4776
Epoch 5/5
196/196
[==============================] - 9s
48ms/step - loss: 1.1852 - accuracy:
0.5036
```

In the testing process, the NN even produces a lower accuracy of 46%.

```
157/157
[==============================] - 2s
8ms/step - loss: 1.3192 - accuracy:
0.4610
```

#### 4.1.2 10 Categories

Running the same NN with the full image set containing 10 distinct categories, the outcome is again worse. The accuracy of the NN is at 42% after training.

```
Epoch 1/5
391/391
[==============================] - 37s
92ms/step - loss: 3.3594 - accuracy:
0.1965
Epoch 2/5
391/391
[==============================] - 19s
47ms/step - loss: 1.8447 - accuracy:
0.3370
Epoch 3/5
391/391
[==============================] - 18s
46ms/step - loss: 1.7368 - accuracy:
0.3830
Epoch 4/5
391/391
[==============================] - 21s
53ms/step - loss: 1.6810 - accuracy:
0.4054
Epoch 5/5
391/391
[==============================] - 22s
57ms/step - loss: 1.6262 - accuracy:
0.4236
```

The accuracy when testing the network is worse, at 41%.

```
313/313
[==============================] - 2s
6ms/step - loss: 1.6353 - accuracy:
0.4177
```

### 4.2 Initial changes

The first idea to improve the network was to decrease the batch size during training. The thought behind that decision was that if the gradient of the loss function is updated more frequently, the weights of the neurons would adapt quicker. Changing the batch size value to the standard of 32, made the network perform better while training, but it performed worse during testing. The effects of the batch size are being studied and a Medium article by Daryl Chang[6] goes into great detail. His experiments show, that at least for non-parallel computer setups and using small datasets, the smaller batch size will almost always perform better. This is enough of a motivation to keep the batch size at 32 and continue with changing the number of epochs. The number of steps displayed during testing is then 1563, as this is the rounded value of the division of the full training set by 32.

[6]https://medium.com/deep-learning-experiments/effect-of-batch-size-on-neural-net-training-c5ae8516e57

The number of epochs has been at 5, as given by the course example. Looking at the before mentioned article, it is found that with a dataset of 23000 images of 2 categories and a batch size of 32, they needed about 150 epochs until the gradient descent algorithm converged. Now, 150 epochs might be too much for fast training and testing, but it provides an idea as to what the epoch number may be. In a first step, the number of epochs is set to 25.

Doing so increased the training accuracy to 51% and a testing accuracy of 41%. This means that the network is probably over-fitted to the data. Other parameters need to be changed. This is being done at the initial 5 epochs, to reduce time spend on training.

## 4.3 Adding and changing Layers

The next change to be done is changing the model itself by adding layers to it. In a first phase, one layer is added with 100 neurons, while the input layer's number of neurons is increased to 1000 neurons. The new layer also uses the ReLU activation function. As this improved accuracy slightly, another layer of 500 neurons is added. This model achieves after 5 epochs a 44% accuracy during training and, very similarly, a testing accuracy of 42%.

To reduce the number of neurons during training that are not useful to the model and at the same time speed up the training, it is advised to add a dropout layer. This layer is placed in this case between the second and second to last layer. Through trial and error, it was found that the dropout rate was best around 0.3, a higher number impacted the network negatively. After the model had further evolved, it was found that this layer decreased the accuracy by 3%, in comparison to the same model without said layer. It was thus removed again.

## 4.4 Optimizer

To find the best optimizer for this task, all the optimizers that are available in the Keras API have been tested. Some are unfit, as they strongly decreased the efficacy of the network. But most fared better than the initially used RMSprop optimizer. The optimizers that performed best are SGD and Adamax, of which the latter reached a 49% accuracy and only needed around 60 seconds per epoch. SGD was generally faster with about 50 seconds per epoch, but the testing accuracy was 1.3% lower. Therefore, it was decided to choose Adamax and accept a slightly higher training time for a better result when using the model on the testing set.

## 4.5 Parameter adjustments

The first parameter that was changed to improve accuracy was changing the activation function of the layers. The ReLU function has been used by default but changing two of the hidden layer's activation function to ELU (exponential linear unit). The Keras documentation explains that ELUs keep the mean of the activation closer to zero, which enables faster learning due to the gradient being closer to the natural gradient.

A hyper-parameter that was changed is the learning rate of the Adamax optimizer. The learning rate was lowered from 0.001 to 0.0005, yielding an accuracy improvement of at least 1 or 2 percent over 5 epochs. This becomes even more significant if the number of epochs is raised.

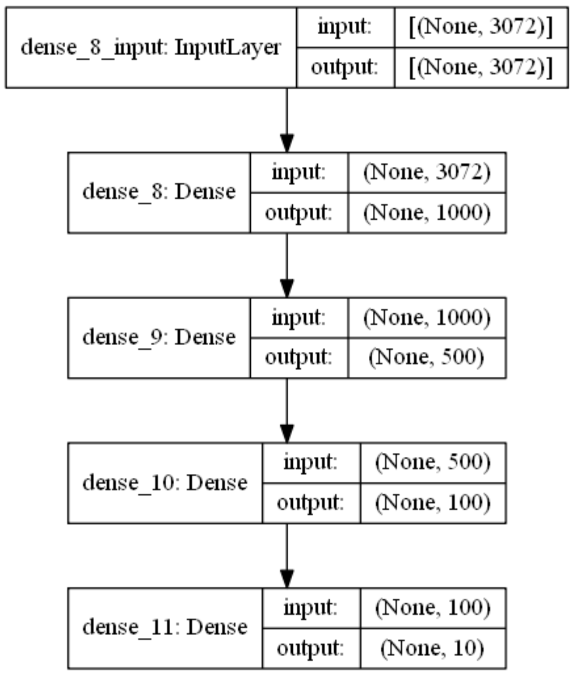
## 4.6 Most note-worthy improvements

The greatest jump in performance was attained using a different optimizer. As the course material stated, the Adam optimizer is a good start, but using Adamax yielded better results.

Another improvement, which seems more obvious, is the addition of more layers. Many resources claim that one should add layers until the performance does not increase anymore. But adding more layers also increases the time a network needs to train. This is the reason for only adding 2 more layers, as this already increased the performance greatly.

# 5 Final Performance Evaluation

In this section the best results that have been achieved for different numbers of categories are described. To have a fair comparison, even though the number of categories changes, the number of epochs and batch size will be the same for all the iterations. The model that worked best is visualized in the following figure.



All the layers are densely connected, when the same model was interlaced with dropout layers, the performance went down.

Furthermore, as already described in previous sections, the Adamax optimizer (with a learning rate of 0.005) and ELU activation function were used in all the following results, as this configuration performed best.

This specific version of the code can be found at the commit hash ‘e9a6b 42803 ddbef 53e0f 7828a 93599 0c7a8 8cb89’, it is also tagged as v1.1.

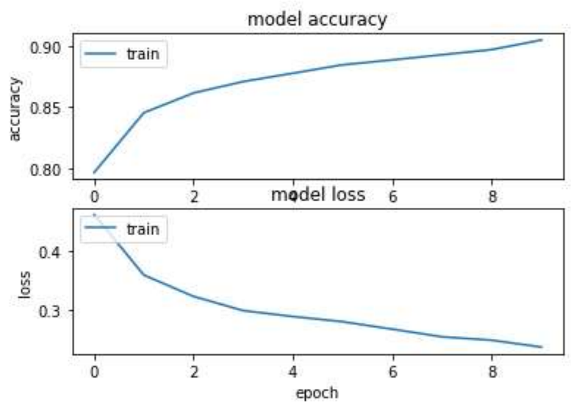
## 5.1 2 Categories

The model trained on 2 categories reported an accuracy of 90% and a loss of 0.23. These values look very promising, but when the model is given the test set, its accuracy is about 8% worse at 82%. An explanation for this is a possible overfit to the training data, where the model has learned well how to categorize the training data but performs worse on new images.

```
313/313
[==============================] - 12s
39ms/step - loss: 0.2307 - accuracy:
0.9023

The model finished training. It took
0:02:04.367327.

63/63 [==============================]
- 1s 12ms/step - loss: 0.4170 -
accuracy: 0.8235
```

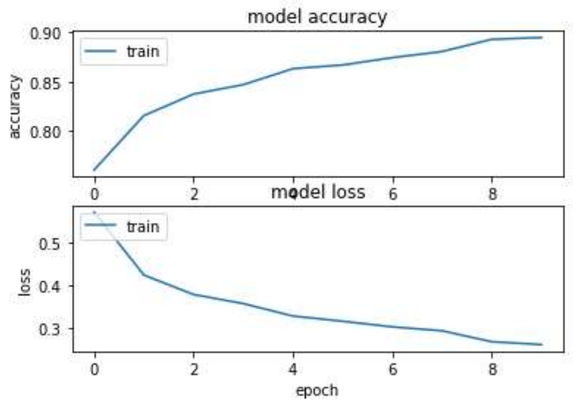


At this point, contrary to what is written in the beginning of this section, the model is run, with all the same parameters, but using dropout layers to reduce the number of active neurons. With this, the training accuracy becomes slightly worse, but the model performs a lot better on the testing images.

```
Epoch 10/10
313/313
[==============================] - 13s
42ms/step - loss: 0.2626 - accuracy:
0.8941

The model finished training. It took
0:02:07.884002.

63/63 [==============================]
- 1s 16ms/step - loss: 0.2737 -
accuracy: 0.8820
```



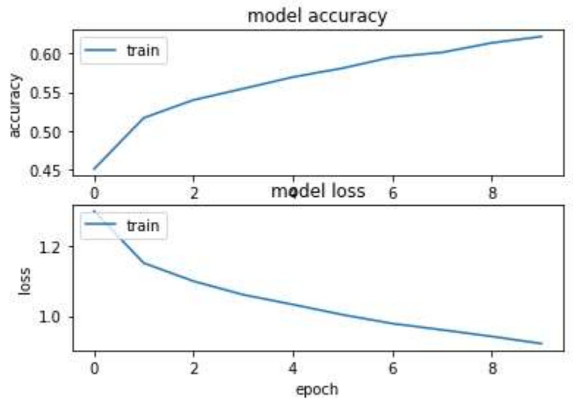
## 5.2 5 Categories

For 5 distinct categories the model attained an accuracy of 62% with a loss of 0.9. Opposed to the previous run with only 2 categories, the model has a very similar accuracy with the testing images, namely ~60%.

```
Epoch 10/10
782/782
[==============================] - 37s
48ms/step - loss: 0.9134 - accuracy:
0.6223

The model finished training. It took
0:05:44.358001.

157/157
[==============================] - 4s
13ms/step - loss: 0.9712 - accuracy:
0.5962
```



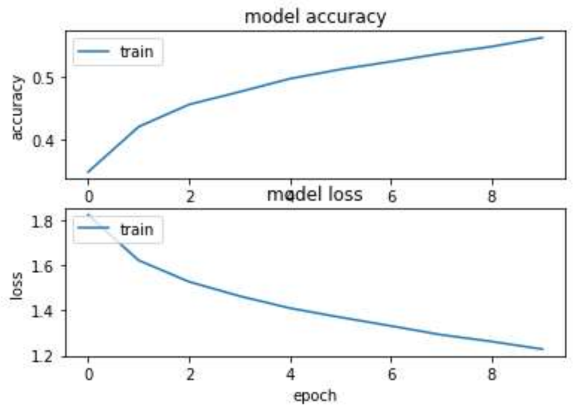
## 5.3 10 Categories

Training the model with all 10 categories yielded results similar to the run with 5 categories. Here, the accuracy after training was 56%. The accuracy for the testing set was 52% percent, which is, at the very least, better than random guessing, as it classifies every second image correctly. Here again we have a larger gap between training and testing accuracy. This could again indicate a sign for overfitting.

```
Epoch 10/10
1563/1563
[==============================] - 82s
52ms/step - loss: 1.2227 - accuracy:
0.5633

The model finished training. It took
0:12:20.633001.

313/313
[==============================] - 5s
14ms/step - loss: 1.3486 - accuracy:
0.5224
```



## 5.4 Final Remarks

Evaluating over- or underfitting is an important aspect of machine learning. But the variables it is depending on, such as bias and variance, are not known for a NN model. What can be done is looking at the error and decompose this value into estimates for bias and variance.[7] This has not been done for this project, though it would

[7] https://machinelearningmastery.com/calculate-the-bias-variance-trade-off/

provide a better notion of how the model can be improved.

Another way to evaluate the performance of the model would be to generate a confusion matrix, containing the guesses of the model on the test set. It helps the human visual understanding to grasp at which point the model fails. For example, it might be able to differentiate well between animals and machines but can't distinguish a cat from a dog.[8] This would be visible in a confusion matrix. This has not been done during this project, but it could have been a nice addition.

Finally, parameter adjustments using Scikit-learn has not been performed. This could have improved the model a lot, as it tries out all possible combinations of parameters, such that it does not have to be tested manually.

[8] https://androidkt.com/keras-confusion-matrix-in-tensorboard/