Deep Learning with Tensorflow 2 and Keras

Introduction

In this workshop we are going to examine the different parts of a neural network and the deep learning techniques needed to build such a network in order to solve real life problems. Shown on Figure 1. is a neural network applying deep learning techniques and used for image recognition.

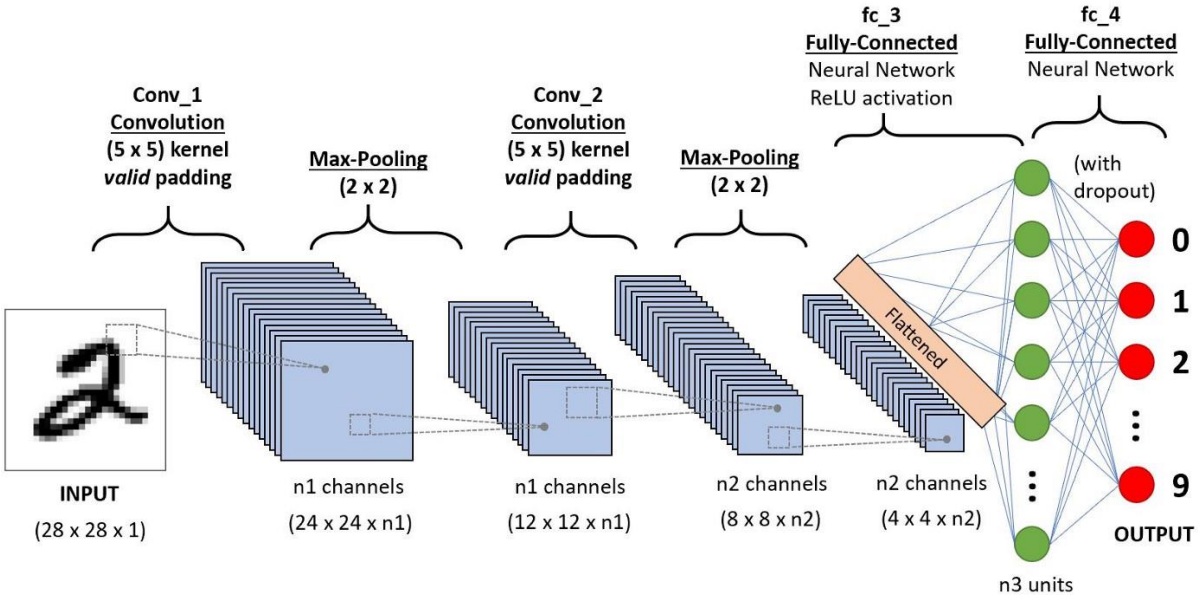

Figure 1. Image recognition using Deep Learning (the image should output 2)

The tools we would need for this workshop are the following:

Keras - https://keras.io/ - an extremely powerful deep learning framework written in Python, build on top of Tensorflow and providing easy to use APIs

Tensorflow - https://www.tensorflow.org/ - an end-to-end open-source platform for machine learning that lets you to easily build and deploy machine learning models

scitik-learn - https://scikit-learn.org/stable/ - an open-source machine learning library providing various tools for model fitting, data preprocessing, data examples and more.

Jupyter - https://jupyter.org/ - an open-source project that provides a web based interactive computing program

“Recognize the numbers” example

```
from sklearn import datasets

digits = datasets.load_digits()

print(digits.data)

digits.data.shape

digits.data[0]
```

digits.data[0] is a one dimensional vector, digits.data.shape is a tuple with only one dimension and 64 numbers. It is an 8 x 8 matrix.

And now let us set up some data for prediction.

```
digits.target

digits.target[-10:]

digits.target.shape
```

Now we want to learn to predict the class of the image. The dataset is divided into different groups (classes) with labels from 0 to 9.

We are going to create a classifier – a piece of code that essentially learns. So we first train the classifier, then we show it some example images and test it.

```
from sklearn import svm

clf = svm.SVC(gamma=0.001, C=100.)

clf.fit(digits.data[:-1], digits.target[:-1])
```

The classifier is a supervised (or semi-supervised) predictor with a finite set of discrete possible output values.

gamma is the scale of the image at which we will perform the analysis and C is the regularization parameter (it basically tells us how much misclassification we want to avoid – in our case 100%, therefore we are aiming at 100% accuracy)

```
clf.predict(digits.data[-1:])

%matplotlib inline

import numpy as np

from matplotlib import pyplot as plt

plt.figure(figsize=(2, 2))

plt.imshow(digits.images[-1], interpolation='nearest', cmap=plt.cm.binary)
```

Using Matplotlib – a comprehensive library for creating static, animated and interactive visualizations in Python, we can plot (visualize) the resulting image.

Numpy is a numeric computations library in Python and pyplot is a figure plotting library.

Environmental setup

Let us first install Tensorflow 2 – go to https://www.tensorflow.org/install

We also need to install Visual C++ Redistributable - https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist?view=msvc-170

And we also need to install Anaconda - https://www.anaconda.com/products/distribution

Now we have to set up a virtual environment and we are going to use the virtual environment package from Python to create it.

Let us open the Anaconda Prompt. Pip stands for Package Installer for Python.

```
pip install virtualenv

virtualenv keras

cd keras

Scripts\activate (bin/activate for Mac users)

pip install jupyter matplotlib sklearn h5py tensorflow

jupyter notebook
```

h5py is a Python-oriented interface to the HDF5 binary data format. It lets you store and manipulate huge amounts of numerical data.

Jupyter notebook opens your default browser and the jupyter notebook.

Lets go to our keras folder ("Users\{username}\keras" under Windows) and paste the files from Google Drive.

In the Jupyter notebook we select New -> Python 3 and create a notebook with whatever name we want. Let's write "import tensorflow" and try to execute the command with Shift+Enter. If it works – we can continue with the rest of the course.

Say “Hi” to Keras

Now that we have setup our environment we will take a look at some implementation of deep learning with Keras.

Let us open the first notebook “01-keras-mnist-begin.ipynb” and write the missing lines from this code.

```
import numpy

numpy.random.seed(1337)   # for experiment reproducibility

import tensorflow

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import SGD

num_classes = 10

batch_size = 128

epochs = 5
```

We will feed 128 data points to the network at a time which makes the computation much more efficient than having to feed all the data points at once.

When your machine learning algorithm has seen all your data points once, we call that an “epoch”. The more “epochs” you have the more likely is that the machine will learn.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

We have 60000 data points which are represented as 28 x 28 pixels images.

We will now reshape and shuffle the data into train and test sets.

First we want to turn the image into a vector.

```
x_train = x_train.reshape(60000,784)
x_test = x_test.reshape(10000,784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255

print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'train samples')

print(x_train.shape)
print(y_train.shape)

y_train[0]
```

The data is between 0 and 255 and we generally want our input data to lie between 0 and 1.

Now let us convert the class vectors into binary class matrixes.

```
y_train = tensorflow.keras.utils.to_categorical(y_train, num_classes)
y_test = tensorflow.keras.utils.to_categorical(y_test, num_classes)
y_train[0]
```

Basically we want to have an output of 1 for the specific class that is being recognized. All the other outputs would be 0, therefore at the end of the neural network we would have a binary class matrix of outputs.

We are now going to create a very simple sequential model.

```
model = Sequential()
model.add(Dense(512, activation='sigmoid', input_shape=(784,)))
model.add(Dense(512, activation='sigmoid'))
model.add(Dense(num_classes, activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy', optimizer=SGD(),metrics=['accuracy'])
```

We have a few Dense layers, the first 2 with 512 nodes and “sigmoid” activation functions and input of a single vector with 784 different values. The input of the second layer depends on the output of the first, so we don’t specify it explicitly.

For the last layer we use 10 nodes and “softmax” activation function.

We compile the model by using a “loss” function (a function compares the target and predicted output values) and we use the “Categorical Cross-entropy” as our “loss” function.

We set up an optimizer to better train the model, and we want to report the “accuracy” metric.

```
history = model.fit(x_train, y_train,
        batch_size = batch_size,
        epochs = epochs,
        verbose = 1,
        validation_data=(x_test, y_test))
```

We store the result in a history object, so that we can take a look at it later on. It has started learning and we can see the result after each epoch.

So let us evaluate the model after all the epochs.

```
score = model.evaluate(x_test, y_test)
score[0]
score[1]
```

Now let us use TensorBoard to visualize the learning. Let us go to “00-tensorflow-test.ipynb”

```
%load_ext tensorboard
import datetime
```

We import the tensorboard extension. We are going to timestamp the files so we import “datetime”.

```
%tensorboard --logdir logs/fit --host localhost --port 8088
```

At the end of the file we can review the logs in Tensor Board. If we experience an error with the default port of 6006, we can specify our own port on which to start Tensor Board, say port 8088.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  # tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
predictions = model(x_train[:1]).numpy()
predictions
```

Problem Description

Now we are ready to work with a real world case study. For this we have picked up an example of a file containing some protein data (a .fasta file). “

We go to “data-scrapes” folder and select “all-human-0001.fasta”.

In the file we have information about proteins and their IDs, say “P27361”. Each protein is composed of an amino acid and all the data is composed different amino acids. Each protein is responsible for certain function within the cell.

In the file “all-human-0001-annotations.txt” we can see what the function of each specific protein is. For the protein with ID of “P27361” the function is “ATP binding”. And the ATP binding function has been given an ID of “GO:0005524”, and we can see in the file that this function is being performed by several other proteins.

What we want to predict with deep learning is:

“Given an amino acid sequence what function would the composed protein perform?”

Note: This is useful for instance in disease identification or development of medicine. Typically this task takes a human expert and performing experiments that could take days. And with this algorithm we can lower the time to just a couple of seconds.

When we look at the data – the first line is the annotation and it tells us which protein has what ID, what database it comes from (say “sp”), what is its human identifiable name, what species it is originating from and so on.

If we go to the “data” folder and open one of the “protein-seqs” files what we want to do is to put the protein ID and then the sequence of amino acids on one line, so we can use it for the algorithm. And in another file “protein-functions” we want to put all the proteins that exibit a particular function. Say “ATP binding”.

Preparing the data

So we go to “02-scrape-to-vec-begin”.

```
from __future__ import print_function
import re
import os
import glob
scrape_dir = os.path.join('..','data-scrapes')
print(scrape_dir)
import datetime, time
ts = time.time()
st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d-%H%M%S')


print("Converting sequences ... ")
out_file = os.path.join('..','data','protein-seqs-' + st + '.txt')


print("Writing to: %s" % out_file)
```

Now we need to define where we are, based on the environment we use “../data-scrapes” on Linux and “..\data-scrapes” on Windows, so we write “os.path.join(‘..’,’data-scrapes’)”.

We also import datetime and time packages and we are going to create a timestamp and convert it to a particular format.

We also want to create an output file “data” and create a new file with protein sequences and append the timestamp with it, and append the “.txt” extension on it.

```
num_proteins_done = 0   # TODO: Remove (here to reduce complexity)


# All files are read like this:
fasta_files = glob.glob(scrape_dir + "/*.fasta")
print(fasta_files)
```

So first of all we want the algorithm to work with a maximum number of records, say 10, so we can get a better understanding of the data. So we load all the .fasta files using “glob”, as it searches all the files that are available.

```
def dump_to_file(protein_id, sequence):
    with open(out_file, "a") as f:
        f.write(protein_id + "," + sequence + "\n")
for fname in fasta_files:
    print("Converting: %s: " % fname)

    proteins = {}   # will hold all proteins in this form ->  id: seq

    with open (fname, 'r') as f:
        protein_seq = ''
        protein_id = ''

        for line in f:
            match = re.search(r'^>([a-z]{2})\|([A-Z0-9]*)\|', line)
            if match:
                if protein_id != '':
                    dump_to_file(protein_id, protein_seq)
                num_proteins_done += 1
                if num_proteins_done > 10: break   # TODO: Remove
                protein_id = match.group(2)
                protein_seq = ''
            else:
                protein_seq += line.strip()

        if protein_id != '':  # we also need the last one dumped
            dump_to_file(protein_id, protein_seq)
```

We have a helper function that will take a protein ID and its sequence and process them using a regular expression. If this matches, either we are starting with the first protein record or we are starting with another one (and in that case we want to write the previous one to a file). We also dump the last line into the file.

```
print("Converting functions ...")
out_file_fns = os.path.join('..', 'data', 'protein-functions-' + st + '.txt')
print(out_file_fns)


target_functions = ['0005524']   # just ATP binding for now
```

And so we save the output file in our “data” folder.

Now we want to convert the functions and we are going to process the annotation files.

We are again going to process each line. When we have found a match, what we are going to do is, we are going to save the protein ID and the function.

```
annot_files = glob.glob(scrape_dir + "/*annotations.txt")
print(annot_files)
has_function = []  # a dictionary of protein_id: boolean  (which says if the protein_id has our target function)


for fname in annot_files:
   with open (fname, 'r') as f:
      for line in f:
         match = re.search(r'([A-Z0-9]*)\sGO:(.*);\sF:.*;', line)
         if match:
            protein_id = match.group(1)
            function = match.group(2)
            if function not in target_functions:
                  continue
            has_function.append(protein_id)


   import json
   with open(out_file_fns, 'w') as fp:
      json.dump(has_function, fp)


   print(has_function[:10])
```

We check if the function is in “target_functions” (which in our case is “ATP binding”) and if not, we skip it, otherwise we append the protein ID that has that function.

We dump the protein sequence in a .json file.

And finally we can take a peek at the last 10 data points.

Loading the data and getting the shapes right

Now that we have the data in the format we want let’s feed it to a deep learning algorithm.

So we are going to start with “03-train-begin” notebook.

```
import numpy as np
import tensorflow as tg
import json
import os

from tensorflow.keras.preprocessing import sequence

sequences_file = os.path.join('..', 'data', 'protein-seqs-2018-01-16-131956.txt')
functions_file = os.path.join('..', 'data', 'protein-functions-2018-01-16-131956.txt')

with open(functions_file) as fn_file:
    has_function = json.load(fn_file)

has_function  # just to see what we have loaded
```

Again we import numpy, json and so on, but we also import “sequence” from tensorflow.keras.preprocessing.

We retrieve the paths to the protein sequences and protein functions and retrieve the files.

```
max_sequence_size = 500
X = []          # sequences in the same order corresponding to elements of p
y = []          # output class: 1 if protein has the function, 0 if not
pos_examples = 0
neg_examples = 0
```

Now there are certain protein sequences with a very large size (over 500) that we want to skip (they are only a few).

We also want to create our own X values and the data points of our Y values would be the corresponding labels of the protein function. The Y values would be either 1 if the protein has the function, or 0 if it doesn't have the function.

And for analysis sake, we are going to track how many positive and negative examples we have.

```
with open(sequences_file) as f:
    for line in f:
        ln = line.split(',')
        protein_id = ln[0].strip()
        seq = ln[1].strip()

        # we're doing this to reduce input size
        if len(seq) >= max_sequence_size:
            continue

        print(line)
        X.append(seq)

        if protein_id in has_function:
            y.append(1)
            pos_examples += 1
        else:
            y.append(0)
            neg_examples += 1

 print("Positive Examples: %d" % pos_examples)
print("Negative Examples: %d" % neg_examples)
```

We open the file and we read a line, get the protein ID and then the sequence and if the length of the sequence is greater than our max_sequence_size we are going to skip it.

We append the sequence to a list of data points (it's one data point) and if the protein ID exists in the has_function list, we append 1 and increment the positive examples, otherwise we append 0 and increment the negative examples.

We can review the proteins that we have processed and the positive and negative examples. We have 2 positive and 5 negative examples, because in "02-scrape-vec-begin" we have limit the code to read only the first 10 proteins.

For the machine to work however we need to convert those sequences to certain numeric values, so we define a new function "sequence_to_indices".

```
def sequence_to_indices(sequence):
    try:
        acid_letters = ['_', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M',
                'N', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y']


        indices = [acid_letters.index(c) for c in list(sequence)]
        return indices
    except Exception:
        print(sequence)
        raise Exception


sequence_to_indices('AD')  # just testing
```

The '_' sign is used in order to fill in the empty space for certain protein sequences in order to make them equal, the '_' sign would be converted to '0' (it's index in the "acid_letters" array). For the rest of the letters, we are going to replace them with the corresponding index in the "acid_letters" array.

Now we are going to loop through all the X values, convert the sequences to indexes and append them to x_all.

```
X_all = []
for i in range(len(X)):
    x = sequence_to_indices(X[i])
    X_all.append(x)
X_all = np.array(X_all)
y_all = np.array(y)
```

We are going to convert the Xs and Ys to numpy arrays. And we can now see the first output, the first data point and the length of all the sequence.

```
print(y[0])
print(X_all[0])
print(len(X_all[0]))
```

But the length is less than 500, so we want to pad it and append additional zeros.

```
X_all = sequence.pad_sequences(X_all, maxlen=max_sequence_size)
X_all[0]
```

So we have our data points in X and Y over here. Now we want to split it into train and test data. So let us look at the shapes first.

```
print(X_all.shape)  # extremely important that you view this!
print(y_all.shape)  # make sure you are comfortable with shapes!
```

We have 7 data points and the vector size is 500, as we have 500 values. And the number of labels that we have is also 7.

Now we want to shuffle the data points. We want to make sure that we are giving it examples on all the chapters and then we test them all.

```
n = X_all.shape[0]
randomize = np.arange(n)
np.random.shuffle(randomize)
randomize
X_all = X_all[randomize]
y_all = y_all[randomize]
test_split = round(n * 2 / 3)
X_train = X_all[:test_split]   # start to (just before) test_split
y_train = y_all[:test_split]
X_test  = X_all[test_split:]   # test_split to end
y_test  = y_all[test_split:]
```

np.arrange(n) is going to give us an array within the provided range and then we are going to shuffle it and we can see the shuffled sequence.

Note: In Deep Learning we want to randomize our data and then split it into train and test sets. We want to have 2 by 3 split, and given that n is 7, test split will be 5. We take the data until the “test_split” as the train data and the rest would be used for testing.

Now we print the shapes and x.train.shape is (5,500) and x_test is (2,500).

```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

Next we are going to take a look at the shapes in detail. So we have 5 data points and each data point has 500 elements. Each data point has 500 letters, those are the letters from the amino acid sequences, that we have converted into arrays of 23 elements, where the letter was changes with its index.

Creating the model

We are now going to create a Sequential model.

```
from tensorflow.keras.layers import Embedding, Input, Dropout, Flatten, Dense, Activation
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.optimizers import SGD

num_amino_acids = 23
embedding_dims = 10
nb_epoch = 2
batch_size = 2

model = Sequential()
model.add(Embedding(num_amino_acids,embedding_dims,
input_length=max_sequence_size))

model.add(Flatten())

model.add(Dense(25, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
```

Now we are going to set some variables – the number of amino acids is going to be 23 and the embedding dimensions are going to be 10. This means that instead of using vectors of 23 dimensions for our data we will embed it into smaller vectors of 10 dimensions. We can even use vectors of 2, 3 dimensions, or say 1 dimension.

We are going to set the number of epochs to 2 (as this is a quite large dataset) and the batch size to 2.

Note: The major problem that most people have when they are trying to construct models for their own dataset is that they don't truly understand how to take care of the data and they get a lot of shape mismatch errors. So the way to handle that is to avoid trying to create the whole model at once.

```
model.compile(loss='binary_crossentropy',
        optimizer=SGD(),
metrics=['accuracy'])
model.summary()
```

So let us compile the model and create the embedding layer first. We want to tell it the number of amino acids that we will use and the embedding dimensions that we will have, and the input_length would be equal to the max sequence size.

We have a shape of embedding (Embedding) => (None, 500, 10) as we have 500 data points and 10 embedded dimensions.

What we want however is to have 2 dimensions (None, 5000) and so we want to collapse (flatten) the input, and so we add another layer.

```
hist = model.fit(X_train, y_train,
            batch_size = batch_size,
            epochs = nb_epoch,
            validation_data = (X_test, y_test),
            verbose=1)
```

Once we have compiled the model, let us try to fit it and see the result.

Using the Functional API

For the Functional API everything is going to be a function. So first we define the input layer as an object and we only give it the shape which is going to be the max_sequence_size.

Afterwards what we need is the embedding layer and we are going to give it the number of amino acids and the embedding dimensions, and because now the Embedding layer is treated as a function, we pass the input as a function.

```
input = Input(shape=(max_sequence_size,))
embedding = Embedding(num_amino_acids, embedding_dims)(input)
```

Now we are going to create a new layer – the Flatten layer and we will pass the embedding variable to it.

```
x = Flatten()(embedding)
x = Dense(25, activation='sigmoid')(x)
x = Dense(1)(x)
output = Activation('sigmoid')(x)
model.summary()
```

We add the Dense layers. Now we have the layers to define the Model with the input and output values.

```
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
hist = model.fit(X_train, y_train,
                 batch_size = batch_size,
                 epochs = nb_epoch,
                 validation_data = (X_test, y_test),
                 verbose=1)
hist.history
```

Convolutional Neural Networks (CNN)

In a fully connected neural network – every node in one layer is connected to every node in the next layer and this is what makes it dense.

We want to apply a Convolutional Neural Network to an example dataset "CIFAR-10" – a dataset of 60000 images 32 x 32 and these are colored images, which means there are in 3 channels (red, green and blue). It has 10 classes with 6000 images per class.

You can explore the "CIFAR-10" dataset at the following link:

https://www.cs.toronto.edu/~kriz/cifar.html

So what we want to do is go ahead and apply CNN to try and classify any of these images into one of these classes.

We go to “04-cnn-basic-begin”. We can see that we import the print_function and numpy. We have several imports from Tensorflow as well.

```
from __future__ import print_function
import numpy as np
np.random.seed(1337)
import tensorflow
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Flatten
from tensorflow.keras.layers import Conv2D
batch_size = 32
num_classes = 10
epochs = 5
```

Note: For TensorFlow 2 we want to change keras.optimizers.rmsprop to tensorflow.keras.optimizers.RMSprop.

We also have a new layer Conv2D which is for 2 dimensional convolution. We load the data and convert it to float and then we output the shapes – the shape is (50000, 32, 32, 3). We have 50000 train samples, each 32 x 32 pixels and each sample has 3 channels (red, green and blue). And we have 10000 test samples.

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

We convert the y-train and y-test to their categorical representations.

```
y_train = tensorflow.keras.utils.to_categorical(y_train, num_classes)
y_test = tensorflow.keras.utils.to_categorical(y_test, num_classes)
```

We have a Sequential model that we can code.

```
model = Sequential()


model.add(Conv2D(16,(3,3),padding='same',input_shape=x_train.shape[1:]))
model.add(Activation('relu'))


model.add(Conv2D(8,(3,3)))
model.add(Activation('relu'))


model.add(Flatten())


model.add(Dense(num_classes))
model.add(Activation('softmax'))


model.summary()
```

We are creating a Convolutional Neural Network, so we are creating the filters – there will be 16 of them and each would be 3 x 3 with zero padding.

The input shape would be the x-train shape 32 x 32 x 3, we can just write it with 1: (starting from the first index until the end).

We are also going to add a new “Activation” layer and the activation value that we will use is called “relu”, and then we are going to add another layer and this layer is again a Conv2D layer with 8 filters with the size of 3 x 3, then we will add another Activation layer.

And then we add another “Dense“ layer, but it needs values in one dimension, so we flatten the output from the other layers by adding a “Flatten” layer and then a “Dense” layer with a “softmax” Activation function.

```
opt = tensorflow.keras.optimizers.RMSprop(lr=0.0001, decay=1e-6)
model.compile(loss='categorical_crossentropy',
            optimizer=opt,
            metrics=['accuracy'])
model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(x_test, y_test))
```

Now we can initialize an optimizer.

Note: With Tensorflow 2 we change the code from keras.optimizers.rmsprop to tensorflow.keras.optimizers.RMSprop.

We compile the model using “categorical_crossentropy”, we give it an optimizer and for metrics we track “accuracy”.

Pooling

For pooling we can add a 2x 2 filter with:

```
from tensorflow.keras.layers import MaxPooling2D

model.add(MaxPooling2D(pool_size=(2, 2)))
```

And in the summary we can see that the size was 30 x 3, but now it is 15 x 15.

Dropout

We hide from the model part of the image in order to recognize the type of image they are trying to recognize.

```
from tensorflow.keras.layers import Dropout

model.add(Dropout(0.2))
```

We give it the percentage of pixels that we want to dropout. Say 0.2, which means that we will drop 20% of the image’s pixels.

Functional API for CNN

So the last thing that we want to do is to convert our current convolutional neural network with pooling and dropout to the Functional API.

```
import numpy as np

np.random.seed(1337)

import tensorflow

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, Activation, Flatten, Input, MaxPooling2D, Dropout

from tensorflow.keras.layers import Conv2D
```

We again split and shuffle the data and convert the vectors to binary class matrixes.

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()


x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255


print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')


y_train = tensorflow.keras.utils.to_categorical(y_train, num_classes)
y_test = tensorflow.keras.utils.to_categorical(y_test, num_classes)
```

So we create an input layer and the shape is going to be (32, 32, 3). Next we are going to create our first layer – the convolutional layer and the padding is going to be “some”. In the next layer we set up the activation.

```
inputs = Input(shape=(32, 32, 3))


x = Conv2D(16, (3, 3), padding='same')(inputs)
x = Activation('relu')(x)
x = Conv2D(8, (3, 3))(x)
x = Activation('relu')(x)


x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.2)(x)


x = Flatten()(x)


x = Dense(num_classes)(x)


output = Activation('softmax')(x)
```

We have a max pooling layer with a pool size of 2 x 2.

We have constructed the model and now we only have to execute it.

```
model = Model([inputs], output)
model.summary()


# initiate RMSprop optimizer
opt = tensorflow.keras.optimizers.RMSprop(lr=0.0001, decay=1e-6)


# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])




model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test))
```