CNN_Training

November 28, 2022

0.1 Preparation

0.1.1 Libraries

It is required to import different libraries for file and image processing, as well as for the definition and construction of neural networks using the Tensorflow library. Therefor the first step is to load all these libraries into the program:

[1]:
```
import matplotlib.pyplot as plt
import glob
import os
from PIL import Image
import numpy as np
from sklearn.utils import shuffle

import tensorflow as tf
from tensorflow.keras.layers import BatchNormalization
from tensorflow.python.keras.layers import Dense, InputLayer, Conv2D,␣
  ↪MaxPool2D, Flatten
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
```

0.1.2 Model name

Next it is important to define the developed model in order to distinguish between different versions of the neural network. It is necesary to assign a unique name for the model before start running its training or testing. This name should be identical at the beginning of each script so that the same data is used consistently.

[2]:
```
ModelNameAndVersion = "counter_det"
```

0.2 Loading the training images

For this CNN, as we stated at the beginning of this section it is required to provide an input of images, that will be used to train and test the model. These images are loaded from a local directory; for this matter, it is used in this directory: 'cnnModelImages' - which can be found under the 'CNN' inside 'Code' section of the GitHub Repo.

For classification purposes, this input needs to be standardized for the program to understand them. Therefore it is also necessary for addition to the image (.jpg), that the file name is set properly and according to the following nomenclature: First Character is the identifier and the rest of the name is not relevant.

Now the output of this section will be the proper classification of the images into two arrays `x_data` (image data) and `y_data` (classification):

[7]:
```
#Setting the variables and initializing the arrays
Input_dir = 'cnnModelImages'
x_data = []
y_data = []

#Loading the image files in a loop over as jpeg images
files = glob.glob(Input_dir + '/*.jpg')
for active_file in files:
    img = Image.open(active_file)                          #Loads the image data␣
 ↪
    data = np.array(img)
    x_data.append(data)

    File_Name = os.path.basename(active_file)              # File_Name
    Classification = File_Name[0:1]                        # Fisrt digit will be␣
 ↪the classification data
    if Classification == "N":                              # Handling special␣
 ↪scenario
        category = 10
    else:
        category = int(Classification)
    category_vector = tf.keras.utils.to_categorical(category, 11) # Conversion␣
 ↪to vector
    y_data.append(category_vector)

x_data = np.array(x_data)
y_data = np.array(y_data)

#print(x_data.shape)
#print(y_data.shape)
```

0.3 Generate training data and test data

0.3.1 Merge and slice the training

Once the data set is ready to be transferred to the model it is important to implement some functions that will allow the model to be properly fit. The first thing is to mix the loaded images, this is done with the “shuffle” function. It is important in this step to make sure that both ‘x_data’ and ‘y_data’ allocation in the array is preserved since this will allow the model to distinguish the number of images.

With the data properly mixed it is time to generate the output for this section of the code, one of the key sections, for the model. The image data is going to be divided into output sets of data: -

Training data (`x_train` and `y_train`) - Test data (`x_test` and `y_test`)

Important to take into account that in this case 80% of the data will be used for training and the remaining 20% is going to be used exclusively for evaluation of the model performance.

[9]:
```
x_data, y_data = shuffle(x_data, y_data)

Testing_Percentage = 0.2       # Define the % of the total data to used for␣
 ↪testing

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,␣
 ↪test_size=Testing_Percentage)
#print(x_train.shape)
#print(x_test.shape)

#print(y_train.shape)
#print(y_test.shape)
```

0.3.2 Image augmentation

Some data transformation is required for the input images to be properly handled. For this specific action, there will not be much data, since this will be covered in the coming 'Development' Section of this same report. ImageDataGenerator function takes from the first definition section which random modifications to the input images will be implemented over the various parameters (shift, brightness, zoom, rotation) and then outputs them. A batch size of 4 is defined and used here, this is chosen since it has proven to be very effective when creating a network that will work with geometries. The generator is applied to both the training and test data.

[10]:
```
### Without Image Augmentation - This section is just as reference for the␣
 ↪original value of the parameters
# Shift_Range = 0
# Brightness_Range = 0
# Rotation_Angle = 0
# ZoomRange = 0
```

[11]:
```
### With Image Augmentation
Shift_Range = 2
Brightness_Range = 0.3
Rotation_Angle = 5
ZoomRange = 0.2

datagen = ImageDataGenerator(width_shift_range  = [-Shift_Range, Shift_Range],
                             height_shift_range = [-Shift_Range, Shift_Range],
                             brightness_range   = [1-Brightness_Range,␣
 ↪1+Brightness_Range],
                             zoom_range         = [1-ZoomRange, 1+ZoomRange],
                             rotation_range     = Rotation_Angle)
Batch_Size = 4
```

```
train_iterator      = datagen.flow(x_train, y_train, batch_size=Batch_Size)
validation_iterator = datagen.flow(x_test,  y_test,  batch_size=Batch_Size)
```

0.4 Definition and structure of the network

It is now the moment to identify the specifications of the implemented network. In this case, the layout consists of a sequence of convolutional and maxpooling layers. 1- The first layer is used to normalize the input data. 2- The second layer is a "flat" layer with 512 neurons.

0.4.1 Inputs

Images of size 32x20 pixels with 3 color channels –> input_shape = (32,20,3)

0.4.2 Output

11 neurons: Digits 0, 1, ..., 9 + Not-A-Number (N/A) as a special case.

0.4.3 Compile

Finally, the network is compiled here so that it can be used. The detailed parameters for this can be found in the relevant literature

[12]:
```
model = tf.keras.Sequential()

model.add(BatchNormalization(input_shape=(32,20,3)))
model.add(Conv2D(16, (3, 3), padding='same', activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Conv2D(32, (3, 3), padding='same', activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Conv2D(16, (3, 3), padding='same', activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(128,activation="relu"))
model.add(Dense(11, activation = "softmax"))

model.summary()

model.compile(loss= tf.keras.losses.categorical_crossentropy,
              optimizer= tf.keras.optimizers.Adadelta(learning_rate=1.0, rho=0.
 ↪95),
              metrics = ["accuracy"])
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 batch_normalization (BatchN  (None, 32, 20, 3)        12
 ormalization)
 module_wrapper (ModuleWrapp  (None, 32, 20, 16)       448
 er)

 module_wrapper_1 (ModuleWra  (None, 16, 10, 16)       0
 pper)

 module_wrapper_2 (ModuleWra  (None, 16, 10, 32)       4640
 pper)

 module_wrapper_3 (ModuleWra  (None, 8, 5, 32)         0
 pper)

 module_wrapper_4 (ModuleWra  (None, 8, 5, 16)         4624
 pper)

 module_wrapper_5 (ModuleWra  (None, 4, 2, 16)         0
 pper)

 module_wrapper_6 (ModuleWra  (None, 128)              0
 pper)

 module_wrapper_7 (ModuleWra  (None, 128)              16512
 pper)

 module_wrapper_8 (ModuleWra  (None, 11)               1419
 pper)

=================================================================
Total params: 27,655
Trainable params: 27,649
Non-trainable params: 6
_________________________________________________________________
```

0.5 Training the network

For training one of the more important concepts to take into account is the number of training cycles (`Epoch_Number`). And here also comes into play the size of the simultaneously trained images, which was previously defined as 4 (`Batch_Size`)

With these two factor the model will run over the total set of training data and will fit the model accordingly to best match the provided input.

[13]:
```
Epoch_Number  = 50
history = model.fit(train_iterator,
                    validation_data = validation_iterator,
                    epochs          = Epoch_Number)
```

```
Epoch 1/50
98/98 [==============================] - 3s 18ms/step - loss: 2.1443 - accuracy:
```

```
0.4158 - val_loss: 2.0649 - val_accuracy: 0.3469
Epoch 2/50
98/98 [==============================] - 1s 13ms/step - loss: 1.9682 - accuracy:
0.4158 - val_loss: 1.9113 - val_accuracy: 0.3571
Epoch 3/50
98/98 [==============================] - 1s 13ms/step - loss: 1.7875 - accuracy:
0.4413 - val_loss: 1.8866 - val_accuracy: 0.3673
Epoch 4/50
98/98 [==============================] - 1s 13ms/step - loss: 1.5501 - accuracy:
0.5051 - val_loss: 1.5655 - val_accuracy: 0.5000
Epoch 5/50
98/98 [==============================] - 1s 13ms/step - loss: 1.5007 - accuracy:
0.5204 - val_loss: 1.5421 - val_accuracy: 0.4796
Epoch 6/50
98/98 [==============================] - 1s 13ms/step - loss: 1.2596 - accuracy:
0.5893 - val_loss: 1.3638 - val_accuracy: 0.5102
Epoch 7/50
98/98 [==============================] - 1s 13ms/step - loss: 1.1482 - accuracy:
0.6352 - val_loss: 1.1886 - val_accuracy: 0.6327
Epoch 8/50
98/98 [==============================] - 1s 12ms/step - loss: 0.9293 - accuracy:
0.7092 - val_loss: 1.1215 - val_accuracy: 0.6429
Epoch 9/50
98/98 [==============================] - 1s 12ms/step - loss: 0.8761 - accuracy:
0.7015 - val_loss: 1.0848 - val_accuracy: 0.6633
Epoch 10/50
98/98 [==============================] - 1s 13ms/step - loss: 0.7541 - accuracy:
0.7474 - val_loss: 1.1371 - val_accuracy: 0.6633
Epoch 11/50
98/98 [==============================] - 1s 13ms/step - loss: 0.7432 - accuracy:
0.7628 - val_loss: 1.0454 - val_accuracy: 0.6633
Epoch 12/50
98/98 [==============================] - 1s 12ms/step - loss: 0.6782 - accuracy:
0.7551 - val_loss: 1.0191 - val_accuracy: 0.7143
Epoch 13/50
98/98 [==============================] - 1s 12ms/step - loss: 0.6333 - accuracy:
0.7730 - val_loss: 1.0637 - val_accuracy: 0.7143
Epoch 14/50
98/98 [==============================] - 1s 13ms/step - loss: 0.6489 - accuracy:
0.7908 - val_loss: 0.9205 - val_accuracy: 0.7041
Epoch 15/50
98/98 [==============================] - 1s 13ms/step - loss: 0.5948 - accuracy:
0.7985 - val_loss: 0.8216 - val_accuracy: 0.7245
Epoch 16/50
98/98 [==============================] - 1s 12ms/step - loss: 0.5251 - accuracy:
0.8087 - val_loss: 0.8158 - val_accuracy: 0.7551
Epoch 17/50
98/98 [==============================] - 1s 12ms/step - loss: 0.6191 - accuracy:
```

```
0.7985 - val_loss: 0.7756 - val_accuracy: 0.7347
Epoch 18/50
98/98 [==============================] - 1s 12ms/step - loss: 0.5469 - accuracy:
0.8265 - val_loss: 0.8434 - val_accuracy: 0.7143
Epoch 19/50
98/98 [==============================] - 1s 13ms/step - loss: 0.4843 - accuracy:
0.8444 - val_loss: 0.8002 - val_accuracy: 0.7755
Epoch 20/50
98/98 [==============================] - 1s 13ms/step - loss: 0.5197 - accuracy:
0.8444 - val_loss: 0.8632 - val_accuracy: 0.7041
Epoch 21/50
98/98 [==============================] - 1s 12ms/step - loss: 0.4779 - accuracy:
0.8571 - val_loss: 0.8375 - val_accuracy: 0.7449
Epoch 22/50
98/98 [==============================] - 1s 13ms/step - loss: 0.4117 - accuracy:
0.8699 - val_loss: 0.8594 - val_accuracy: 0.7653
Epoch 23/50
98/98 [==============================] - 1s 12ms/step - loss: 0.4668 - accuracy:
0.8444 - val_loss: 0.9214 - val_accuracy: 0.7143
Epoch 24/50
98/98 [==============================] - 1s 12ms/step - loss: 0.3587 - accuracy:
0.8750 - val_loss: 0.7608 - val_accuracy: 0.7857
Epoch 25/50
98/98 [==============================] - 1s 13ms/step - loss: 0.4094 - accuracy:
0.8673 - val_loss: 0.7783 - val_accuracy: 0.8061
Epoch 26/50
98/98 [==============================] - 1s 13ms/step - loss: 0.4493 - accuracy:
0.8622 - val_loss: 0.6756 - val_accuracy: 0.8163
Epoch 27/50
98/98 [==============================] - 1s 13ms/step - loss: 0.4481 - accuracy:
0.8750 - val_loss: 0.7532 - val_accuracy: 0.7857
Epoch 28/50
98/98 [==============================] - 1s 13ms/step - loss: 0.3530 - accuracy:
0.8929 - val_loss: 0.8846 - val_accuracy: 0.7143
Epoch 29/50
98/98 [==============================] - 1s 12ms/step - loss: 0.3040 - accuracy:
0.9005 - val_loss: 0.7868 - val_accuracy: 0.7653
Epoch 30/50
98/98 [==============================] - 1s 13ms/step - loss: 0.3558 - accuracy:
0.8929 - val_loss: 0.8299 - val_accuracy: 0.7755
Epoch 31/50
98/98 [==============================] - 1s 12ms/step - loss: 0.3190 - accuracy:
0.8929 - val_loss: 0.6582 - val_accuracy: 0.7959
Epoch 32/50
98/98 [==============================] - 1s 13ms/step - loss: 0.2992 - accuracy:
0.9362 - val_loss: 0.9584 - val_accuracy: 0.8163
Epoch 33/50
98/98 [==============================] - 1s 12ms/step - loss: 0.3456 - accuracy:
```

```
0.8903 - val_loss: 0.6098 - val_accuracy: 0.8265
Epoch 34/50
98/98 [==============================] - 1s 13ms/step - loss: 0.2796 - accuracy:
0.9107 - val_loss: 0.6716 - val_accuracy: 0.7755
Epoch 35/50
98/98 [==============================] - 1s 13ms/step - loss: 0.3834 - accuracy:
0.8699 - val_loss: 0.7031 - val_accuracy: 0.7857
Epoch 36/50
98/98 [==============================] - 1s 13ms/step - loss: 0.2575 - accuracy:
0.9209 - val_loss: 0.8095 - val_accuracy: 0.8163
Epoch 37/50
98/98 [==============================] - 1s 13ms/step - loss: 0.2456 - accuracy:
0.9209 - val_loss: 0.7661 - val_accuracy: 0.8061
Epoch 38/50
98/98 [==============================] - 1s 13ms/step - loss: 0.2745 - accuracy:
0.9133 - val_loss: 0.9254 - val_accuracy: 0.7755
Epoch 39/50
98/98 [==============================] - 1s 13ms/step - loss: 0.3168 - accuracy:
0.9158 - val_loss: 0.7265 - val_accuracy: 0.7959
Epoch 40/50
98/98 [==============================] - 1s 12ms/step - loss: 0.3356 - accuracy:
0.9005 - val_loss: 0.7074 - val_accuracy: 0.7959
Epoch 41/50
98/98 [==============================] - 1s 12ms/step - loss: 0.2670 - accuracy:
0.9107 - val_loss: 0.9057 - val_accuracy: 0.7653
Epoch 42/50
98/98 [==============================] - 1s 12ms/step - loss: 0.2628 - accuracy:
0.9082 - val_loss: 0.5808 - val_accuracy: 0.8265
Epoch 43/50
98/98 [==============================] - 1s 12ms/step - loss: 0.3258 - accuracy:
0.9158 - val_loss: 0.5568 - val_accuracy: 0.7959
Epoch 44/50
98/98 [==============================] - 1s 12ms/step - loss: 0.2320 - accuracy:
0.9311 - val_loss: 0.5323 - val_accuracy: 0.8265
Epoch 45/50
98/98 [==============================] - 1s 12ms/step - loss: 0.2319 - accuracy:
0.9235 - val_loss: 0.8088 - val_accuracy: 0.8163
Epoch 46/50
98/98 [==============================] - 1s 12ms/step - loss: 0.1818 - accuracy:
0.9388 - val_loss: 0.5199 - val_accuracy: 0.8571
Epoch 47/50
98/98 [==============================] - 1s 11ms/step - loss: 0.1760 - accuracy:
0.9413 - val_loss: 0.5991 - val_accuracy: 0.8367
Epoch 48/50
98/98 [==============================] - 1s 12ms/step - loss: 0.1834 - accuracy:
0.9464 - val_loss: 0.7558 - val_accuracy: 0.8061
Epoch 49/50
98/98 [==============================] - 1s 12ms/step - loss: 0.1781 - accuracy:
```

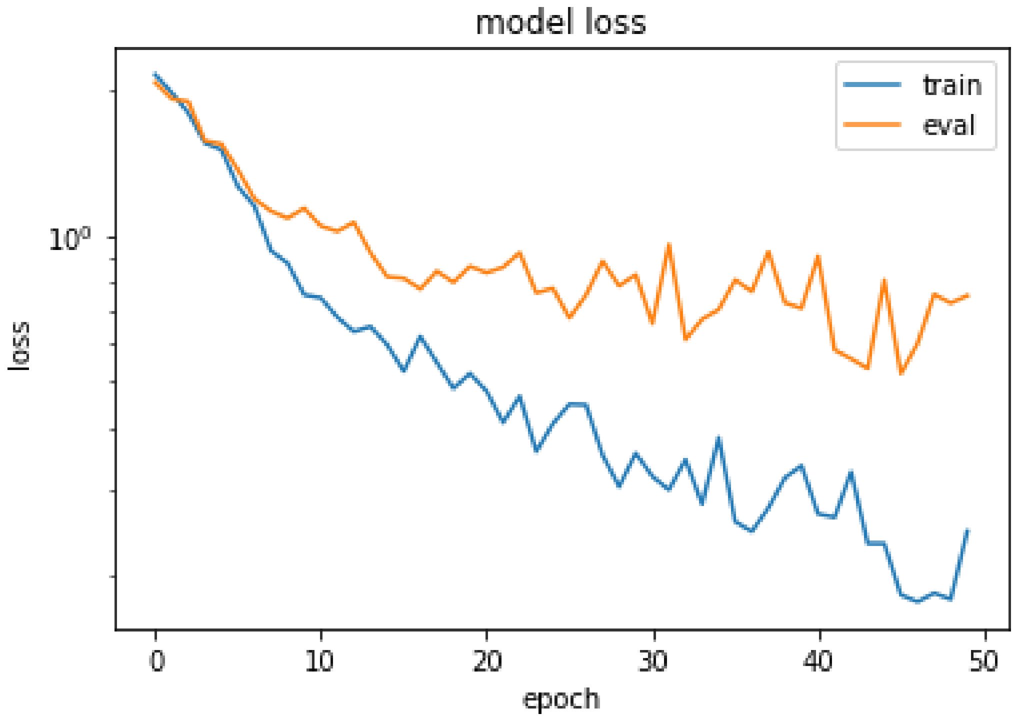
```
0.9388 - val_loss: 0.7252 - val_accuracy: 0.8367
Epoch 50/50
98/98 [==============================] - 1s 12ms/step - loss: 0.2461 - accuracy:
0.9286 - val_loss: 0.7511 - val_accuracy: 0.8367
```

0.5.1 Visualization of training

The model is now ready to use. For a better understanding of the output, one can be visualized how the training was performed. For this matter, the error is used, both in the training data as well as in the test data.

[14]:
```python
plt.semilogy(history.history['loss'])
plt.semilogy(history.history['val_loss'])

plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train','eval'], loc='upper right')
plt.show()
```



0.6 Saving the Neural Network

Last step is to get the outputed neural network saved for further use. The complete network is first saved here in the so-called H5 format.

[15]:
```
## H5-Format
model.save('saved_model/' + ModelNameAndVersion)
```

```
WARNING:absl:Found untraced functions such as
conv2d_layer_call_and_return_conditional_losses, conv2d_layer_call_fn,
conv2d_1_layer_call_and_return_conditional_losses, conv2d_1_layer_call_fn,
conv2d_2_layer_call_and_return_conditional_losses while saving (showing 5 of
12). These functions will not be directly callable after loading.
```

```
INFO:tensorflow:Assets written to: saved_model/counter_det\assets
```

```
INFO:tensorflow:Assets written to: saved_model/counter_det\assets
```