Capstone Project-release

November 27, 2020

1 Capstone Project

1.1 Image classifier for the SVHN dataset

1.1.1 Instructions

In this notebook, you will create a neural network that classifies real-world images digits. You will use concepts from throughout this course in building, training, testing, validating and saving your Tensorflow classifier model.

This project is peer-assessed. Within this notebook you will find instructions in each section for how to complete the project. Pay close attention to the instructions as the peer review will be carried out according to a grading rubric that checks key parts of the project instructions. Feel free to add extra cells into the notebook as required.

1.1.2 How to submit

When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able to view the outputs of your notebook. Save the notebook as a pdf (File -> Download as -> PDF via LaTeX). You should then submit this pdf for review.

1.1.3 Let's get started!

We'll start by running some imports, and loading the dataset. For this project you are free to make further imports throughout the notebook as you wish.

[1]:
```
import tensorflow as tf
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt
```

1.1.4 check available GPU (my PC)

[2]:
```
print("Num GPUs Available: ", len(tf.config.experimental.
 ↪list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

[3]:
```
#tf.debugging.set_log_device_placement(True)
```

For the capstone project, you will use the SVHN dataset. This is an image dataset of over 600,000 digit images in all, and is a harder dataset than MNIST as the numbers appear in the context of natural scene images. SVHN is obtained from house numbers in Google Street View images.

- Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng. "Reading Digits in Natural Images with Unsupervised Feature Learning". NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.

Your goal is to develop an end-to-end workflow for building, training, validating, evaluating and saving a neural network that classifies a real-world image into one of ten classes.

[4]:
```
# Run this cell to load the dataset

train = loadmat('data/train_32x32.mat')
test = loadmat('data/test_32x32.mat')
```

Both `train` and `test` are dictionaries with keys `X` and `y` for the input images and labels respectively.

1.2 1. Inspect and preprocess the dataset

- Extract the training and testing images and labels separately from the train and test dictionaries loaded for you.
- Select a random sample of images and corresponding labels from the dataset (at least 10), and display them in a figure.
- Convert the training and test images to grayscale by taking the average across all colour channels for each pixel. *Hint: retain the channel dimension, which will now have size 1.*
- Select a random sample of the grayscale images and corresponding labels from the dataset (at least 10), and display them in a figure.

prepare datasets (replace 10 value with 0)

[5]:
```
x_train, y_train = train['X'] , np.where(train['y'] == 10, 0, train['y'])
```

[6]:
```
x_test, y_test = test['X'],  np.where(test['y'] == 10, 0, test['y'])
```

make appropriate shape (Batch, 32,32,3)

[7]:
```
x_train = np.moveaxis(x_train, -1, 0)
x_train.shape
```

[7]:
```
(73257, 32, 32, 3)
```

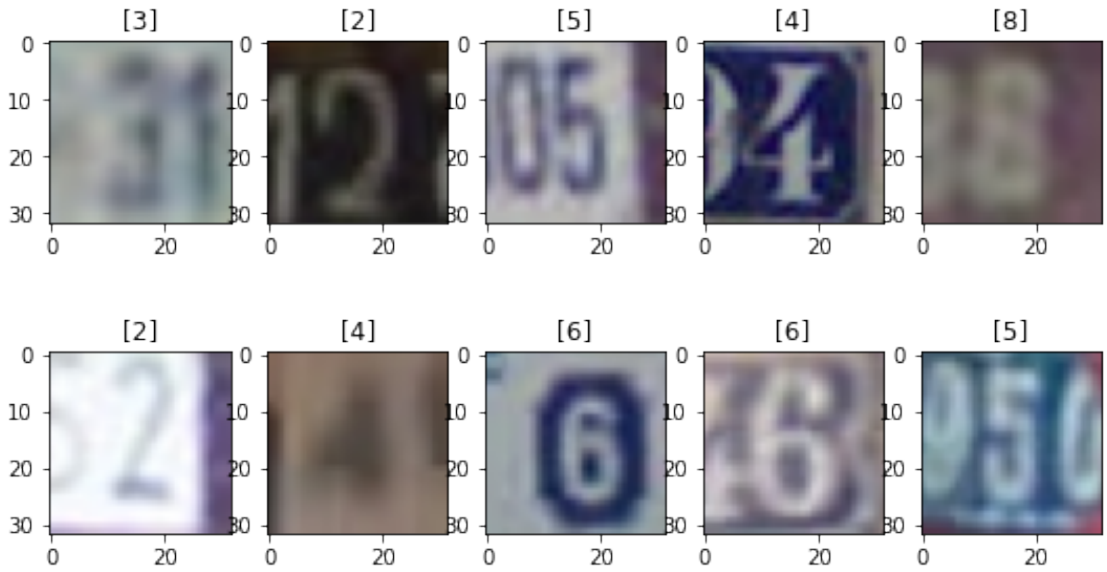
[8]:
```
x_test = np.moveaxis(x_test, -1, 0)
x_test.shape
```

[8]:
```
(26032, 32, 32, 3)
```

pick up 10 random images and plot

[9]:
```
rnd_idxs = np.random.randint(0, x_train.shape[0], 10)
```

[10]:
```
fig, axs = plt.subplots(nrows=2, ncols=5, figsize=(9, 5))
for ax, i in zip(axs.flat, rnd_idxs):
    ax.set_title(str(y_train[i]))
    ax.imshow(x_train[i])
```



convert to grayscale (my mean of 3 channels)

[11]:
```
x_train_gray = np.mean(x_train, axis = -1, keepdims=True) / 255
x_train_gray.shape
```

[11]: (73257, 32, 32, 1)

[12]:
```
x_test_gray = np.mean(x_test, axis = -1, keepdims=True) / 255
x_test_gray.shape
```

[12]: (26032, 32, 32, 1)

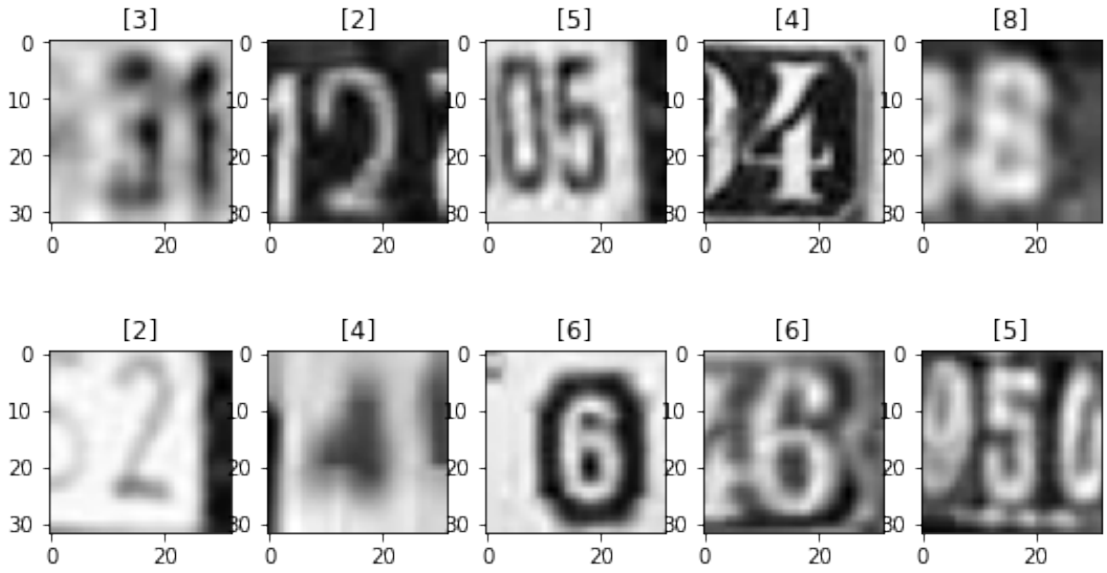
[13]:
```
x_train_gray.max(), x_train_gray.min()
```

[13]: (1.0, 0.0)

plot result

[14]:
```
fig, axs = plt.subplots(nrows=2, ncols=5, figsize=(9, 5))
for ax, i in zip(axs.flat, rnd_idxs):
```

```
    ax.set_title(str(y_train[i]))
    ax.imshow(np.squeeze(x_train_gray[i]), cmap = "gray")
```



1.3 2. MLP neural network classifier

- Build an MLP classifier model using the Sequential API. Your model should use only Flatten and Dense layers, with the final layer having a 10-way softmax output.
- You should design and build the model yourself. Feel free to experiment with different MLP architectures. *Hint: to achieve a reasonable accuracy you won't need to use more than 4 or 5 layers.*
- Print out the model summary (using the summary() method)
- Compile and train the model (we recommend a maximum of 30 epochs), making use of both training and validation sets during the training run.
- Your model should track at least one appropriate metric, and use at least two callbacks during training, one of which should be a ModelCheckpoint callback.
- As a guide, you should aim to achieve a final categorical cross entropy training loss of less than 1.0 (the validation loss might be higher).
- Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets.
- Compute and display the loss and accuracy of the trained model on the test set.

[15]:
```
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Sequential
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

set number of epochs (30)

[16]:
```
n_epochs = 30 #10
```

define MLP model

[17]:
```
def get_MLP_model(input_shape):
    model = Sequential([
        Flatten(input_shape=input_shape),
        Dense(256, activation='relu',  kernel_initializer='he_uniform'),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        Dense(16, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

function print model accuracy

[18]:
```
def print_accuracy(model, x_test, y_test):
    _, acc = model.evaluate(x=x_test, y=y_test, verbose=0)
    print('accuracy: {acc:0.3f}'.format(acc=acc))
```

[19]:
```
model_MLP = get_MLP_model(x_train_gray[0].shape)
model_MLP.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 256)               262400
_________________________________________________________________
dense_1 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_3 (Dense)              (None, 16)                1040
_________________________________________________________________
dense_4 (Dense)              (None, 10)                170
=================================================================
Total params: 304,762
Trainable params: 304,762
Non-trainable params: 0
_________________________________________________________________
```

set model callbacks

[20]:
```
checkpoint_path_best_mlp = "checkpoints_best_for_MLP/checkpoint"
checkpoint_MLP = ModelCheckpoint(filepath = checkpoint_path_best_mlp,
```

```
                                save_best_only=True,
                                save_weights_only=True,
                                save_freq='epoch',
                                monitor = 'val_accuracy',
                                verbose = 1)

early_stop = EarlyStopping(monitor = 'val_accuracy',
                                patience=3, verbose=1)
```

train model and save history

[21]:
```
history_MLP = model_MLP.fit(x_train_gray, y_train, epochs=n_epochs,␣
↪batch_size=32,
             validation_split=0.15, callbacks=[checkpoint_MLP, early_stop],␣
↪verbose = 1)
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/30
61952/62268 [============================>.] - ETA: 0s - loss: 2.1884 -
accuracy: 0.2070
Epoch 00001: val_accuracy improved from -inf to 0.26818, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 72us/sample - loss: 2.1877 -
accuracy: 0.2074 - val_loss: 2.0381 - val_accuracy: 0.2682
Epoch 2/30
61472/62268 [============================>.] - ETA: 0s - loss: 1.7343 -
accuracy: 0.3898
Epoch 00002: val_accuracy improved from 0.26818 to 0.55101, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.7310 -
accuracy: 0.3910 - val_loss: 1.3524 - val_accuracy: 0.5510
Epoch 3/30
61856/62268 [============================>.] - ETA: 0s - loss: 1.3030 -
accuracy: 0.5714
Epoch 00003: val_accuracy improved from 0.55101 to 0.58222, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.3028 -
accuracy: 0.5714 - val_loss: 1.2703 - val_accuracy: 0.5822
Epoch 4/30
61696/62268 [============================>.] - ETA: 0s - loss: 1.2204 -
accuracy: 0.6029
Epoch 00004: val_accuracy improved from 0.58222 to 0.61816, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 65us/sample - loss: 1.2200 -
accuracy: 0.6030 - val_loss: 1.1892 - val_accuracy: 0.6182
Epoch 5/30
61728/62268 [============================>.] - ETA: 0s - loss: 1.1724 -
accuracy: 0.6213
```

```
Epoch 00005: val_accuracy improved from 0.61816 to 0.63245, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.1723 -
accuracy: 0.6214 - val_loss: 1.1460 - val_accuracy: 0.6325
Epoch 6/30
61344/62268 [============================>.] - ETA: 0s - loss: 1.1367 -
accuracy: 0.6339
Epoch 00006: val_accuracy improved from 0.63245 to 0.64692, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.1352 -
accuracy: 0.6346 - val_loss: 1.1042 - val_accuracy: 0.6469
Epoch 7/30
62080/62268 [============================>.] - ETA: 0s - loss: 1.0930 -
accuracy: 0.6509
Epoch 00007: val_accuracy improved from 0.64692 to 0.66011, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0930 -
accuracy: 0.6510 - val_loss: 1.0697 - val_accuracy: 0.6601
Epoch 8/30
62176/62268 [============================>.] - ETA: 0s - loss: 1.0719 -
accuracy: 0.6589
Epoch 00008: val_accuracy improved from 0.66011 to 0.66148, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0719 -
accuracy: 0.6589 - val_loss: 1.0461 - val_accuracy: 0.6615
Epoch 9/30
62016/62268 [============================>.] - ETA: 0s - loss: 1.0580 -
accuracy: 0.6637
Epoch 00009: val_accuracy improved from 0.66148 to 0.67040, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0576 -
accuracy: 0.6638 - val_loss: 1.0290 - val_accuracy: 0.6704
Epoch 10/30
61472/62268 [============================>.] - ETA: 0s - loss: 1.0389 -
accuracy: 0.6693
Epoch 00010: val_accuracy did not improve from 0.67040
62268/62268 [==============================] - 4s 63us/sample - loss: 1.0399 -
accuracy: 0.6691 - val_loss: 1.1051 - val_accuracy: 0.6468
Epoch 11/30
61664/62268 [============================>.] - ETA: 0s - loss: 1.0342 -
accuracy: 0.6709
Epoch 00011: val_accuracy did not improve from 0.67040
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0341 -
accuracy: 0.6710 - val_loss: 1.0583 - val_accuracy: 0.6588
Epoch 12/30
61856/62268 [============================>.] - ETA: 0s - loss: 1.0274 -
accuracy: 0.6726
Epoch 00012: val_accuracy improved from 0.67040 to 0.68132, saving model to
```

```
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0273 -
accuracy: 0.6726 - val_loss: 0.9994 - val_accuracy: 0.6813
Epoch 13/30
61792/62268 [============================>.] - ETA: 0s - loss: 1.0136 -
accuracy: 0.6780
Epoch 00013: val_accuracy did not improve from 0.68132
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0132 -
accuracy: 0.6782 - val_loss: 1.0076 - val_accuracy: 0.6810
Epoch 14/30
61664/62268 [============================>.] - ETA: 0s - loss: 1.0047 -
accuracy: 0.6806
Epoch 00014: val_accuracy did not improve from 0.68132
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0043 -
accuracy: 0.6808 - val_loss: 0.9987 - val_accuracy: 0.6808
Epoch 15/30
61696/62268 [============================>.] - ETA: 0s - loss: 1.0060 -
accuracy: 0.6797
Epoch 00015: val_accuracy improved from 0.68132 to 0.68741, saving model to
checkpoints_best_for_MLP/checkpoint
62268/62268 [==============================] - 4s 65us/sample - loss: 1.0052 -
accuracy: 0.6801 - val_loss: 0.9908 - val_accuracy: 0.6874
Epoch 16/30
61664/62268 [============================>.] - ETA: 0s - loss: 1.0033 -
accuracy: 0.6810
Epoch 00016: val_accuracy did not improve from 0.68741
62268/62268 [==============================] - 4s 64us/sample - loss: 1.0043 -
accuracy: 0.6807 - val_loss: 1.0010 - val_accuracy: 0.6790
Epoch 17/30
61952/62268 [============================>.] - ETA: 0s - loss: 0.9936 -
accuracy: 0.6842
Epoch 00017: val_accuracy did not improve from 0.68741
62268/62268 [==============================] - 4s 64us/sample - loss: 0.9939 -
accuracy: 0.6840 - val_loss: 1.0318 - val_accuracy: 0.6664
Epoch 18/30
61472/62268 [============================>.] - ETA: 0s - loss: 0.9907 -
accuracy: 0.6857
Epoch 00018: val_accuracy did not improve from 0.68741
62268/62268 [==============================] - 4s 63us/sample - loss: 0.9901 -
accuracy: 0.6859 - val_loss: 0.9949 - val_accuracy: 0.6833
Epoch 00018: early stopping
```

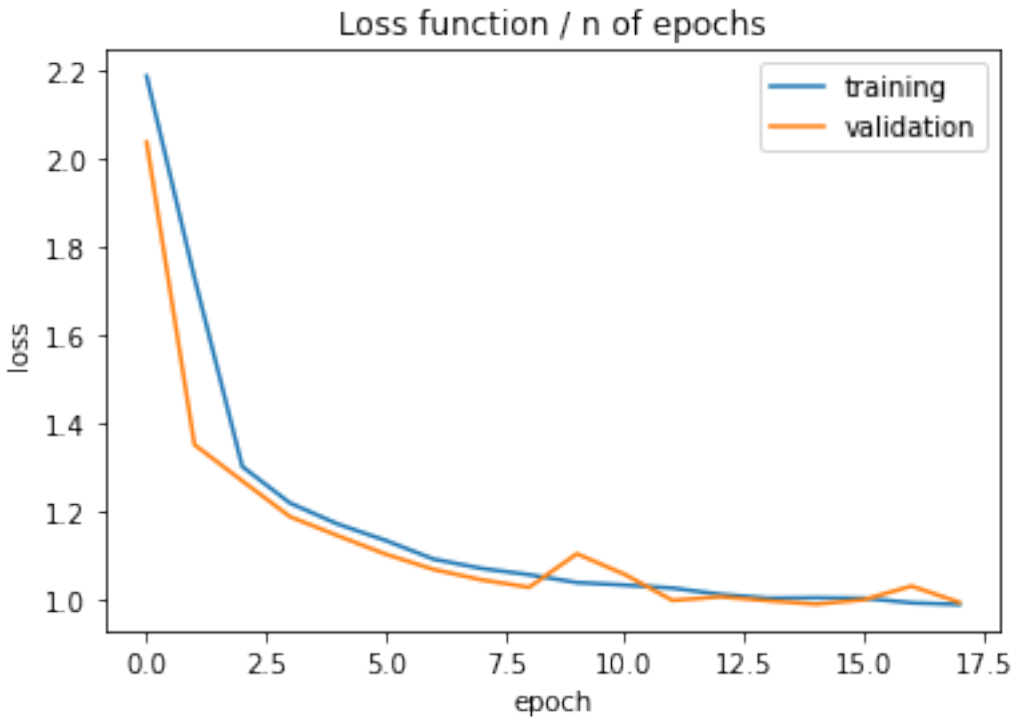
print accuracy and plot history

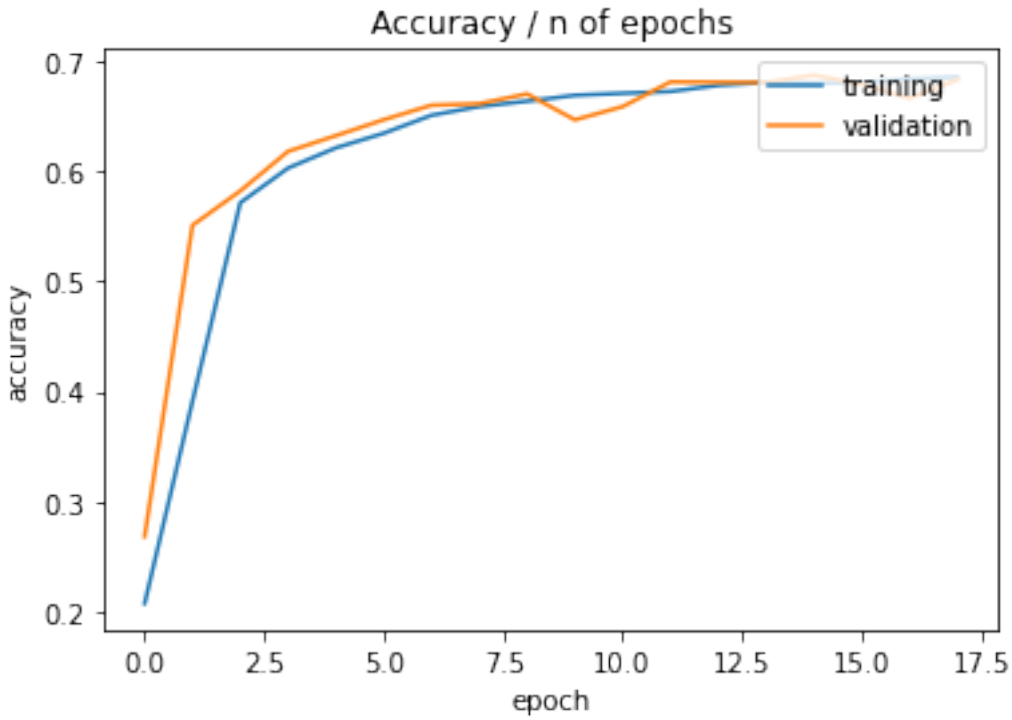
[22]:
```
print_accuracy(model_MLP, x_test_gray, y_test)
```

```
accuracy: 0.654
```

[23]:
```
def plot_history(history):
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Loss function / n of epochs')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['training', 'validation'], loc='upper right')
    plt.show()

    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Accuracy / n of epochs')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['training', 'validation'], loc='upper right')
    plt.show()
```

[24]:
```
plot_history(history_MLP)
```





1.4 3. CNN neural network classifier

- Build a CNN classifier model using the Sequential API. Your model should use the Conv2D, MaxPool2D, BatchNormalization, Flatten, Dense and Dropout layers. The final layer should again have a 10-way softmax output.
- You should design and build the model yourself. Feel free to experiment with different CNN architectures. *Hint: to achieve a reasonable accuracy you won't need to use more than 2 or 3 convolutional layers and 2 fully connected layers.)*
- The CNN model should use fewer trainable parameters than your MLP model.
- Compile and train the model (we recommend a maximum of 30 epochs), making use of both training and validation sets during the training run.
- Your model should track at least one appropriate metric, and use at least two callbacks during training, one of which should be a ModelCheckpoint callback.
- You should aim to beat the MLP model performance with fewer parameters!
- Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets.
- Compute and display the loss and accuracy of the trained model on the test set.

[25]:
```
from tensorflow.keras.layers import Conv2D, BatchNormalization, MaxPooling2D,␣
↪Dropout
```

define CNN model

[26]:
```
def get_CNN_model(input_shape):
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', padding='SAME',␣
↪input_shape=input_shape,  kernel_initializer='he_uniform'),
        MaxPooling2D((2,2), padding='SAME'),
        BatchNormalization(),
        Conv2D(16, (3,3), activation='relu', padding='SAME'),
        MaxPooling2D((2,2), padding='SAME'),
        BatchNormalization(),
        Flatten(),
        Dense(64, activation='relu'),
        Dropout(0.5),
        Dense(32, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])
    return model
```

set model callbacks

[27]:
```
checkpoint_path_best_cnn = "checkpoints_best_for_CNN/checkpoint"
checkpoint_CNN = ModelCheckpoint(filepath = checkpoint_path_best_cnn,
                                 save_best_only=True,
                                 save_weights_only=True,
                                 save_freq='epoch',
                                 monitor = 'val_accuracy',
                                 verbose = 1)
```

[28]:
```
model_CNN = get_CNN_model(x_train_gray[0].shape)
model_CNN.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
batch_normalization (BatchNo (None, 16, 16, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 16)        4624
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 16)          0
_________________________________________________________________
batch_normalization_1 (Batch (None, 8, 8, 16)          64
_________________________________________________________________
```

```
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_5 (Dense)              (None, 64)                65600
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_6 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_7 (Dense)              (None, 10)                330
=================================================================
Total params: 73,146
Trainable params: 73,050
Non-trainable params: 96
_________________________________________________________________
```

train model and save history

[29]:
```
history_CNN = model_CNN.fit(x_train_gray, y_train, epochs=n_epochs,␣
↪batch_size=64,
            validation_split=0.15, callbacks=[checkpoint_CNN, early_stop],␣
↪verbose = 1)
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/30
61376/62268 [============================>.] - ETA: 0s - loss: 1.4084 -
accuracy: 0.5143
Epoch 00001: val_accuracy improved from -inf to 0.79616, saving model to
checkpoints_best_for_CNN/checkpoint
62268/62268 [==============================] - 5s 80us/sample - loss: 1.3999 -
accuracy: 0.5177 - val_loss: 0.6776 - val_accuracy: 0.7962
Epoch 2/30
61760/62268 [============================>.] - ETA: 0s - loss: 0.7250 -
accuracy: 0.7721
Epoch 00002: val_accuracy improved from 0.79616 to 0.84066, saving model to
checkpoints_best_for_CNN/checkpoint
62268/62268 [==============================] - 3s 53us/sample - loss: 0.7243 -
accuracy: 0.7722 - val_loss: 0.5663 - val_accuracy: 0.8407
Epoch 3/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.6159 -
accuracy: 0.8077
Epoch 00003: val_accuracy did not improve from 0.84066
62268/62268 [==============================] - 3s 55us/sample - loss: 0.6158 -
accuracy: 0.8077 - val_loss: 0.5228 - val_accuracy: 0.8405
Epoch 4/30
61568/62268 [============================>.] - ETA: 0s - loss: 0.5635 -
accuracy: 0.8256
Epoch 00004: val_accuracy improved from 0.84066 to 0.86932, saving model to
checkpoints_best_for_CNN/checkpoint
```

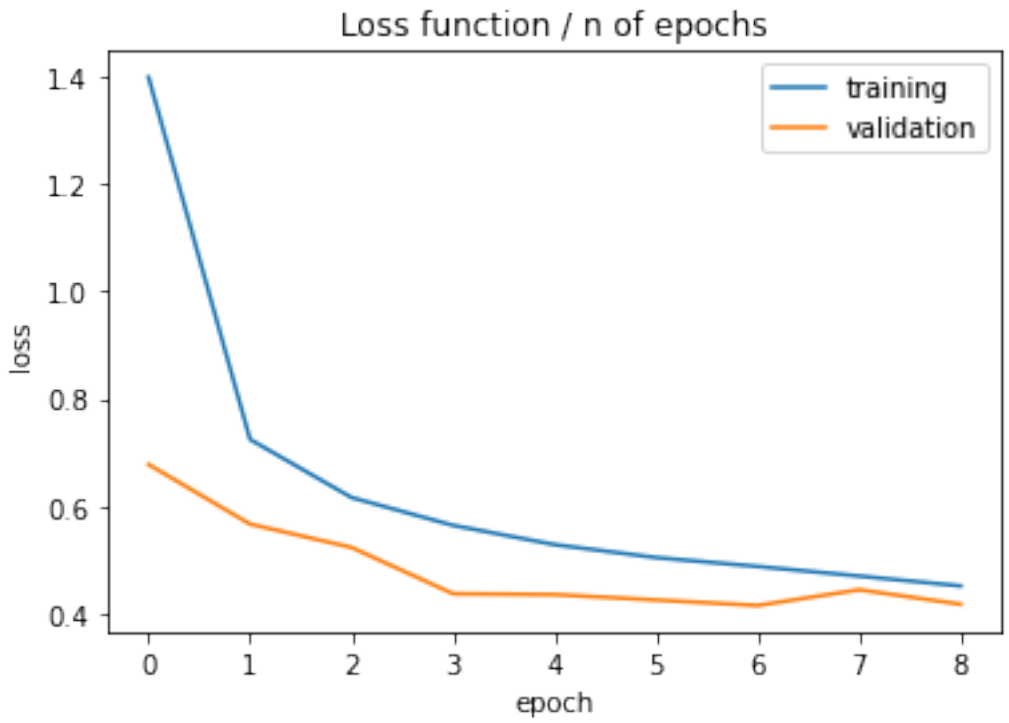
```
62268/62268 [==============================] - 3s 54us/sample - loss: 0.5639 -
accuracy: 0.8255 - val_loss: 0.4364 - val_accuracy: 0.8693
Epoch 5/30
61312/62268 [============================>.] - ETA: 0s - loss: 0.5284 -
accuracy: 0.8365
Epoch 00005: val_accuracy improved from 0.86932 to 0.87196, saving model to
checkpoints_best_for_CNN/checkpoint
62268/62268 [==============================] - 3s 54us/sample - loss: 0.5280 -
accuracy: 0.8365 - val_loss: 0.4348 - val_accuracy: 0.8720
Epoch 6/30
61888/62268 [============================>.] - ETA: 0s - loss: 0.5037 -
accuracy: 0.8425
Epoch 00006: val_accuracy improved from 0.87196 to 0.87851, saving model to
checkpoints_best_for_CNN/checkpoint
62268/62268 [==============================] - 3s 56us/sample - loss: 0.5038 -
accuracy: 0.8424 - val_loss: 0.4247 - val_accuracy: 0.8785
Epoch 7/30
61760/62268 [============================>.] - ETA: 0s - loss: 0.4875 -
accuracy: 0.8496
Epoch 00007: val_accuracy did not improve from 0.87851
62268/62268 [==============================] - 3s 55us/sample - loss: 0.4873 -
accuracy: 0.8496 - val_loss: 0.4145 - val_accuracy: 0.8782
Epoch 8/30
61504/62268 [============================>.] - ETA: 0s - loss: 0.4690 -
accuracy: 0.8537
Epoch 00008: val_accuracy did not improve from 0.87851
62268/62268 [==============================] - 3s 55us/sample - loss: 0.4694 -
accuracy: 0.8536 - val_loss: 0.4438 - val_accuracy: 0.8680
Epoch 9/30
61824/62268 [============================>.] - ETA: 0s - loss: 0.4512 -
accuracy: 0.8591
Epoch 00009: val_accuracy did not improve from 0.87851
62268/62268 [==============================] - 4s 58us/sample - loss: 0.4510 -
accuracy: 0.8591 - val_loss: 0.4168 - val_accuracy: 0.8765
Epoch 00009: early stopping
```

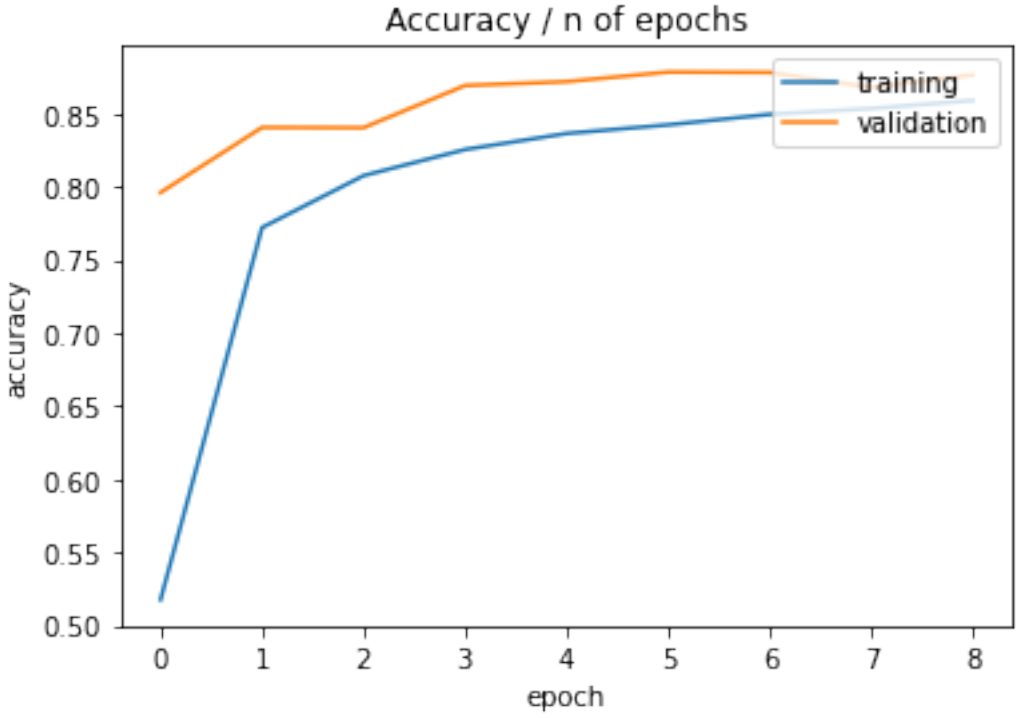
print accuracy and plot history

[30]:
```
print_accuracy(model_CNN, x_test_gray, y_test)
```

```
accuracy: 0.863
```

[31]:
```
plot_history(history_CNN)
```


Loss function / n of epochs
training
validation
loss
epoch
1.4
1.2
1.0
0.8
0.6
0.4
0
1
2
3
4
5
6
7
8


Accuracy / n of epochs
training
validation
accuracy
epoch
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0
1
2
3
4
5
6
7
8

1.5 4. Get model predictions

- Load the best weights for the MLP and CNN models that you saved during the training run.
- Randomly select 5 images and corresponding labels from the test set and display the images with their labels.
- Alongside the image and label, show each model's predictive distribution as a bar chart, and the final model prediction given by the label with maximum probability.

load the best weights of the MLP and CNN

[32]:
```
model_MLP = get_MLP_model(x_train_gray[0].shape)
model_MLP.load_weights(checkpoint_path_best_mlp)
print_accuracy(model_MLP, x_test_gray, y_test)
```

```
accuracy: 0.653
```

[33]:
```
model_CNN = get_CNN_model(x_train_gray[0].shape)
model_CNN.load_weights(checkpoint_path_best_cnn)
print_accuracy(model_CNN, x_test_gray, y_test)
```

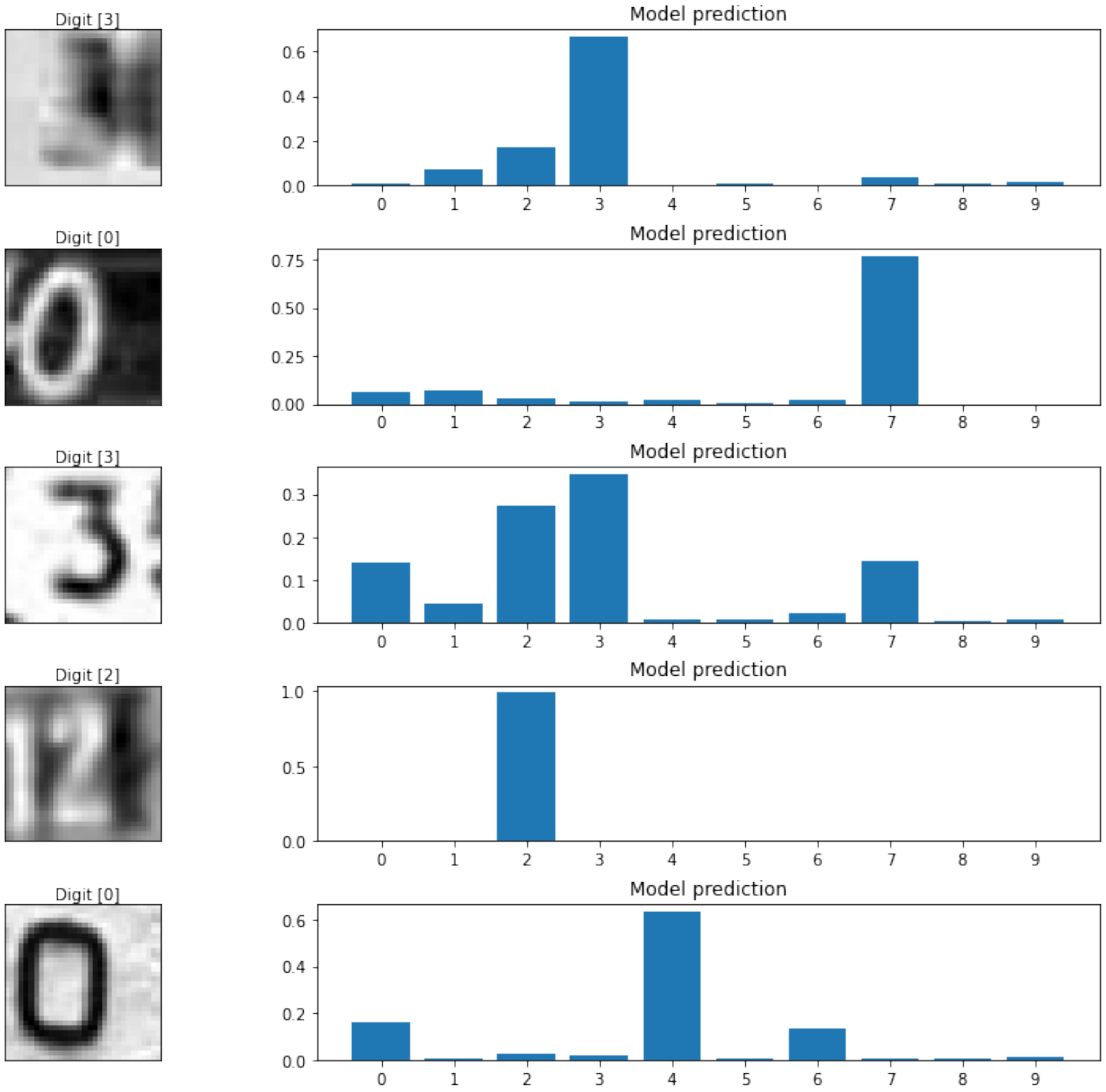
```
accuracy: 0.862
```

function for print pedictions

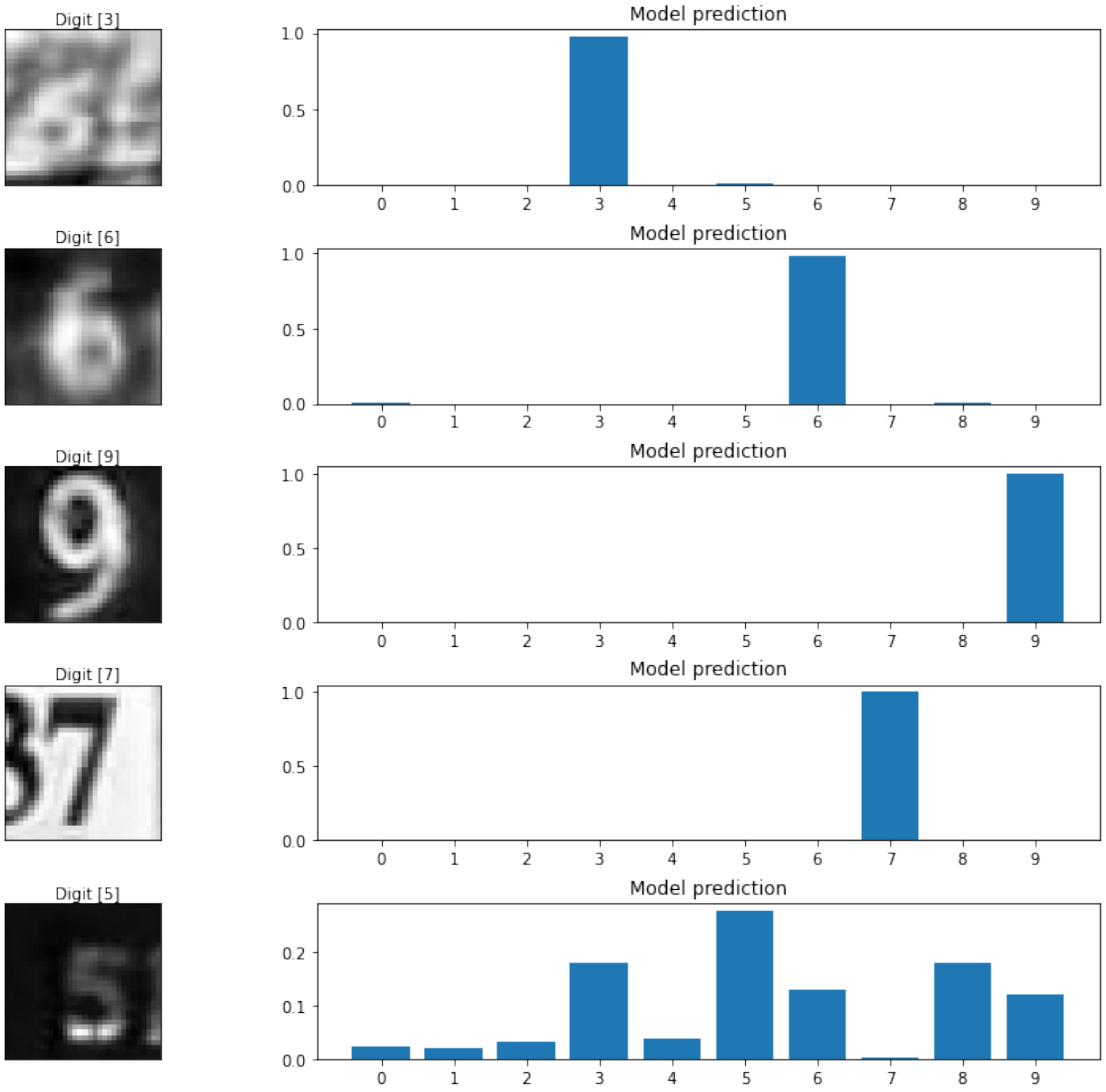
[34]:
```
def print_predictions(model, x_test, y_test):
    idxs = np.random.choice(x_test.shape[0], 5)
    images = x_test[idxs, ...]
    labels = y_test[idxs, ...]

    preds = model.predict(images)

    fig, axes = plt.subplots(5, 2, figsize=(16, 12))
    fig.subplots_adjust(hspace=0.4, wspace=-0.2)

    for i, (pred, image, label) in enumerate(zip(preds, images, labels)):
        axes[i, 0].imshow(np.squeeze(image), cmap='gray')
        axes[i, 0].get_xaxis().set_visible(False)
        axes[i, 0].get_yaxis().set_visible(False)
        axes[i, 0].text(10., -1.5, f'Digit {label}')
        axes[i, 1].bar(np.arange(10), pred)
        #print(pred)
        axes[i, 1].set_xticks(np.arange(10))
        axes[i, 1].set_title("Model prediction")

    plt.show()
```

1.5.1 print CNN and MLP models predictions

[35]:
```
print_predictions(model_MLP, x_test_gray, y_test)
```



[36]:
```
print_predictions(model_CNN, x_test_gray, y_test)
```


Digit [3]
Model prediction
Digit [6]
Model prediction
Digit [9]
Model prediction
Digit [7]
Model prediction
Digit [5]
Model prediction

[ ]: