Capstone_Project_2

November 7, 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 (you could download the notebook with File -> Download .ipynb, open the notebook locally, and then File -> Download as -> PDF via LaTeX), and 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
```

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.

The train and test datasets required for this project can be downloaded from here and here. Once unzipped, you will have two files: `train_32x32.mat` and `test_32x32.mat`. You should store these files in Drive for use in this Colab notebook.

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.

[2]:
```
# Run this cell to connect to your Drive folder

# from google.colab import drive
# drive.mount('/content/gdrive')
```

[3]:
```
# Load the dataset from your Drive folder

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.

[4]:
```
x_train = train['X']
y_train = train['y']
x_train = x_train.astype('float64')
y_train = y_train.astype('int64')

x_test = test['X']
x_test = x_test.astype('float64')
y_test = test['y']
y_test = y_test.astype('int64')

# reorder data
x_train = np.moveaxis(x_train, -1, 0)
x_test = np.moveaxis(x_test, -1, 0)
```

[5]:
```
print('Min: {}, Max: {}'.format(x_train.min(), x_train.max()))
```

```
Min: 0.0, Max: 255.0
```

[6]:

```
def normalize_data(x):
    '''
    normalize data so that values are between 0 to 1
    '''
    x = x / 255.0
    return x

x_train = normalize_data(x_train)
x_test = normalize_data(x_test)
```

[7]:

```
import matplotlib.pyplot as plt

# this code displays 5x5 random images in gray scale

n_width = 5
n_height = 5
fig, ax = plt.subplots(nrows=n_height, ncols=n_width)
fig.subplots_adjust(hspace=0.8, wspace=0.1)
fig.suptitle('Display random images', fontsize=16)
flattened_ax = ax.flatten()
for i in flattened_ax:
    rand_idx = np.random.randint(x_train.shape[0])
    i.set_axis_off()
    i.imshow(x_train[rand_idx, :, :, :])
    i.title.set_text("Label: " + str(y_train[rand_idx]))
```

[8]:
```python
def colored_to_gray(x):
    '''
    input shape: n_sample, n_x, x_y, n_channel
    output shape: n_sample, n_x, x_y, 1
    this is a rudementary way of converting a colored image into gray image
    '''
    x = np.mean(x, axis=-1, keepdims=True)
    return x

x_train = colored_to_gray(x_train)
x_test = colored_to_gray(x_test)

print("Shape of Training Data: {}".format(x_train.shape))
print("Shape of Training Labels: {}".format(y_train.shape))
print("Shape of Testing Data: {}".format(x_test.shape))
print("Shape of Testing Labels: {}".format(y_test.shape))
```

```
Shape of Training Data: (73257, 32, 32, 1)
Shape of Training Labels: (73257, 1)
Shape of Testing Data: (26032, 32, 32, 1)
Shape of Testing Labels: (26032, 1)
```

[9]:
```python
# this code displays 5x5 random images in gray scale

n_width = 5
n_height = 5
fig, ax = plt.subplots(nrows=n_height, ncols=n_width)
fig.subplots_adjust(hspace=0.8, wspace=0.1)
fig.suptitle('Display random images', fontsize=16)
flattened_ax = ax.flatten()
for i in flattened_ax:
    rand_idx = np.random.randint(x_train.shape[0])
    i.set_axis_off()
    i.imshow(x_train[rand_idx, :, :], cmap='gray')
    i.title.set_text("Label: " + str(y_train[rand_idx]))
```

[10]:
```
print('Min: {}, Max: {}'.format(x_train.min(), x_train.max()))
```

```
Min: 0.0, Max: 1.0
```

1.2.1 One hot encoder

Convert training and test labels to one hot encoded matrix

[11]:
```
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer()
y_train = lb.fit_transform(y_train)
y_test = lb.fit_transform(y_test)
```

[ ]:
```

```

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.

[12]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
```

[13]:
```
def lr_function(epoch, lr):
    if (epoch) % 10 == 0:
        return lr/5
    else:
        return lr

model_dnn = Sequential([
    Flatten(name='Flatten_Input', input_shape=x_train.shape[1:]),
    Dense(units=2048, activation='relu', name='Dense_1'),
    Dense(units=1024, activation='relu', name='Dense_2'),
    Dense(units=512, activation='relu', name='Dense_3'),
    Dense(units=10, activation='softmax', name='Output')
])

opt = tf.keras.optimizers.Adam(learning_rate=0.005)

model_dnn.compile(optimizer=opt,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

[14]:
```
model_dnn.summary()
```

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Flatten_Input (Flatten)      (None, 1024)              0

_________________________________________________________________
Dense_1 (Dense)              (None, 2048)              2099200

_________________________________________________________________
Dense_2 (Dense)              (None, 1024)              2098176

_________________________________________________________________
Dense_3 (Dense)              (None, 512)               524800

_________________________________________________________________
Output (Dense)               (None, 10)                5130
=================================================================
Total params: 4,727,306
```

```
Trainable params: 4,727,306
Non-trainable params: 0
_________________________________________________________________
```

[15]:

```
callback_list = [
    tf.keras.callbacks.LearningRateScheduler(lr_function, verbose=1),
    tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=3),
    tf.keras.callbacks.
 ↪ModelCheckpoint(filepath='checkpoints_best_only_DNN\checkpoint',
                                       save_weights_only=True,
                                       monitor='val_accuracy',
                                       save_best_only=True)
]

history_dnn = model_dnn.fit(x_train, y_train, epochs=50,
                    validation_split = 0.15,
                    batch_size=256,
                    shuffle=True,
                    callbacks=callback_list)
```

```
Train on 62268 samples, validate on 10989 samples

Epoch 00001: LearningRateScheduler reducing learning rate to
0.0009999999776482583.
Epoch 1/50
62268/62268 [==============================] - 2s 31us/sample - loss: 2.2111 -
accuracy: 0.2028 - val_loss: 1.9165 - val_accuracy: 0.3196

Epoch 00002: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 2/50
62268/62268 [==============================] - 1s 20us/sample - loss: 1.5802 -
accuracy: 0.4545 - val_loss: 1.3337 - val_accuracy: 0.5609

Epoch 00003: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 3/50
62268/62268 [==============================] - 1s 20us/sample - loss: 1.2392 -
accuracy: 0.5955 - val_loss: 1.2227 - val_accuracy: 0.5938

Epoch 00004: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 4/50
62268/62268 [==============================] - 1s 19us/sample - loss: 1.1067 -
accuracy: 0.6442 - val_loss: 1.1556 - val_accuracy: 0.6309

Epoch 00005: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
```

```
Epoch 5/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.9831 -
accuracy: 0.6889 - val_loss: 0.9897 - val_accuracy: 0.6819

Epoch 00006: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 6/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.9125 -
accuracy: 0.7139 - val_loss: 0.8912 - val_accuracy: 0.7129

Epoch 00007: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 7/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.8499 -
accuracy: 0.7315 - val_loss: 0.8745 - val_accuracy: 0.7167

Epoch 00008: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 8/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.8021 -
accuracy: 0.7484 - val_loss: 0.8516 - val_accuracy: 0.7296

Epoch 00009: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 9/50
62268/62268 [==============================] - 1s 17us/sample - loss: 0.7691 -
accuracy: 0.7577 - val_loss: 0.9190 - val_accuracy: 0.7073

Epoch 00010: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 10/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.7307 -
accuracy: 0.7704 - val_loss: 0.8231 - val_accuracy: 0.7366

Epoch 00011: LearningRateScheduler reducing learning rate to
0.0001999999862164259.
Epoch 11/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.6432 -
accuracy: 0.7986 - val_loss: 0.7018 - val_accuracy: 0.7766

Epoch 00012: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 12/50
62268/62268 [==============================] - 1s 19us/sample - loss: 0.6265 -
accuracy: 0.8040 - val_loss: 0.6996 - val_accuracy: 0.7791

Epoch 00013: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
```

```
Epoch 13/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.6170 -
accuracy: 0.8065 - val_loss: 0.6820 - val_accuracy: 0.7851

Epoch 00014: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 14/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.6069 -
accuracy: 0.8110 - val_loss: 0.6740 - val_accuracy: 0.7903

Epoch 00015: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 15/50
62268/62268 [==============================] - 1s 19us/sample - loss: 0.5992 -
accuracy: 0.8129 - val_loss: 0.6786 - val_accuracy: 0.7867

Epoch 00016: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 16/50
62268/62268 [==============================] - 1s 18us/sample - loss: 0.5905 -
accuracy: 0.8156 - val_loss: 0.6716 - val_accuracy: 0.7886

Epoch 00017: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 17/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5831 -
accuracy: 0.8170 - val_loss: 0.6583 - val_accuracy: 0.7921

Epoch 00018: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 18/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5758 -
accuracy: 0.8197 - val_loss: 0.6588 - val_accuracy: 0.7966

Epoch 00019: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 19/50
62268/62268 [==============================] - 1s 18us/sample - loss: 0.5693 -
accuracy: 0.8219 - val_loss: 0.6672 - val_accuracy: 0.7905

Epoch 00020: LearningRateScheduler reducing learning rate to
0.0001999999803956598.
Epoch 20/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5590 -
accuracy: 0.8258 - val_loss: 0.6461 - val_accuracy: 0.7993

Epoch 00021: LearningRateScheduler reducing learning rate to
3.999999607913196e-05.
```

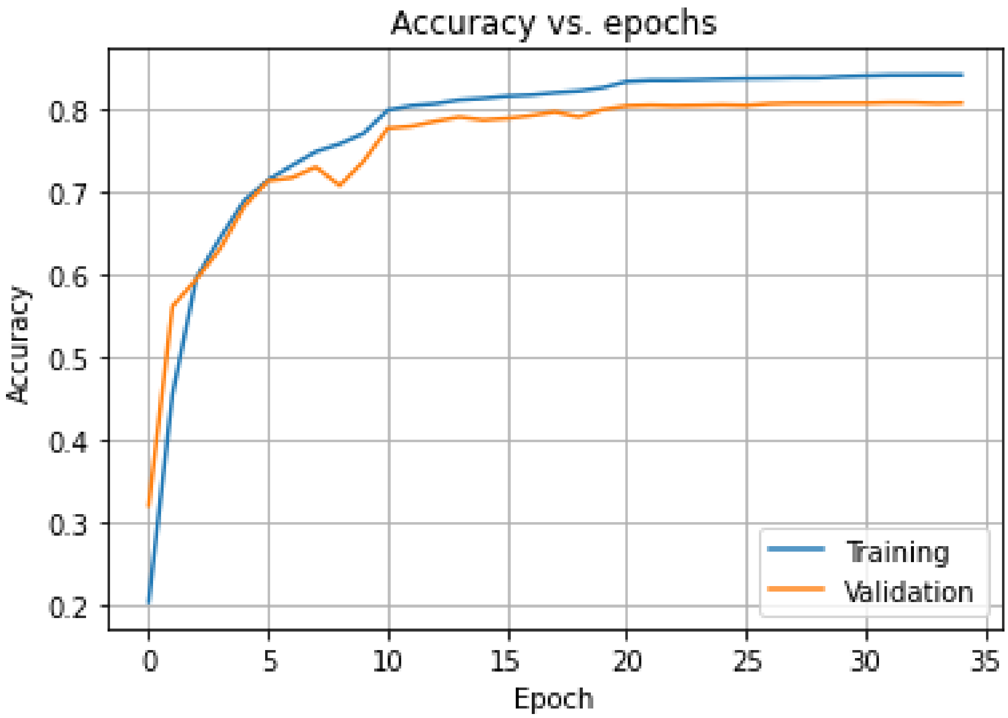
```
Epoch 21/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5355 -
accuracy: 0.8332 - val_loss: 0.6315 - val_accuracy: 0.8037

Epoch 00022: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 22/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5317 -
accuracy: 0.8348 - val_loss: 0.6342 - val_accuracy: 0.8046

Epoch 00023: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 23/50
62268/62268 [==============================] - 1s 17us/sample - loss: 0.5301 -
accuracy: 0.8349 - val_loss: 0.6296 - val_accuracy: 0.8039

Epoch 00024: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 24/50
62268/62268 [==============================] - 1s 17us/sample - loss: 0.5285 -
accuracy: 0.8355 - val_loss: 0.6293 - val_accuracy: 0.8043

Epoch 00025: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 25/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5270 -
accuracy: 0.8363 - val_loss: 0.6299 - val_accuracy: 0.8050

Epoch 00026: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 26/50
62268/62268 [==============================] - 1s 18us/sample - loss: 0.5247 -
accuracy: 0.8369 - val_loss: 0.6277 - val_accuracy: 0.8043

Epoch 00027: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 27/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5232 -
accuracy: 0.8372 - val_loss: 0.6278 - val_accuracy: 0.8064

Epoch 00028: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 28/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5218 -
accuracy: 0.8377 - val_loss: 0.6259 - val_accuracy: 0.8071

Epoch 00029: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
```

```
Epoch 29/50
62268/62268 [==============================] - 1s 17us/sample - loss: 0.5197 -
accuracy: 0.8377 - val_loss: 0.6246 - val_accuracy: 0.8071

Epoch 00030: LearningRateScheduler reducing learning rate to
3.99999953515362e-05.
Epoch 30/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5189 -
accuracy: 0.8392 - val_loss: 0.6270 - val_accuracy: 0.8072

Epoch 00031: LearningRateScheduler reducing learning rate to
7.99999907030724e-06.
Epoch 31/50
62268/62268 [==============================] - 1s 18us/sample - loss: 0.5121 -
accuracy: 0.8402 - val_loss: 0.6222 - val_accuracy: 0.8072

Epoch 00032: LearningRateScheduler reducing learning rate to
7.99999907030724e-06.
Epoch 32/50
62268/62268 [==============================] - 1s 20us/sample - loss: 0.5115 -
accuracy: 0.8411 - val_loss: 0.6220 - val_accuracy: 0.8078

Epoch 00033: LearningRateScheduler reducing learning rate to
7.99999907030724e-06.
Epoch 33/50
62268/62268 [==============================] - 1s 18us/sample - loss: 0.5110 -
accuracy: 0.8412 - val_loss: 0.6221 - val_accuracy: 0.8077

Epoch 00034: LearningRateScheduler reducing learning rate to
7.99999907030724e-06.
Epoch 34/50
62268/62268 [==============================] - 1s 17us/sample - loss: 0.5106 -
accuracy: 0.8413 - val_loss: 0.6227 - val_accuracy: 0.8069

Epoch 00035: LearningRateScheduler reducing learning rate to
7.99999907030724e-06.
Epoch 35/50
62268/62268 [==============================] - 1s 18us/sample - loss: 0.5103 -
accuracy: 0.8412 - val_loss: 0.6217 - val_accuracy: 0.8074
```

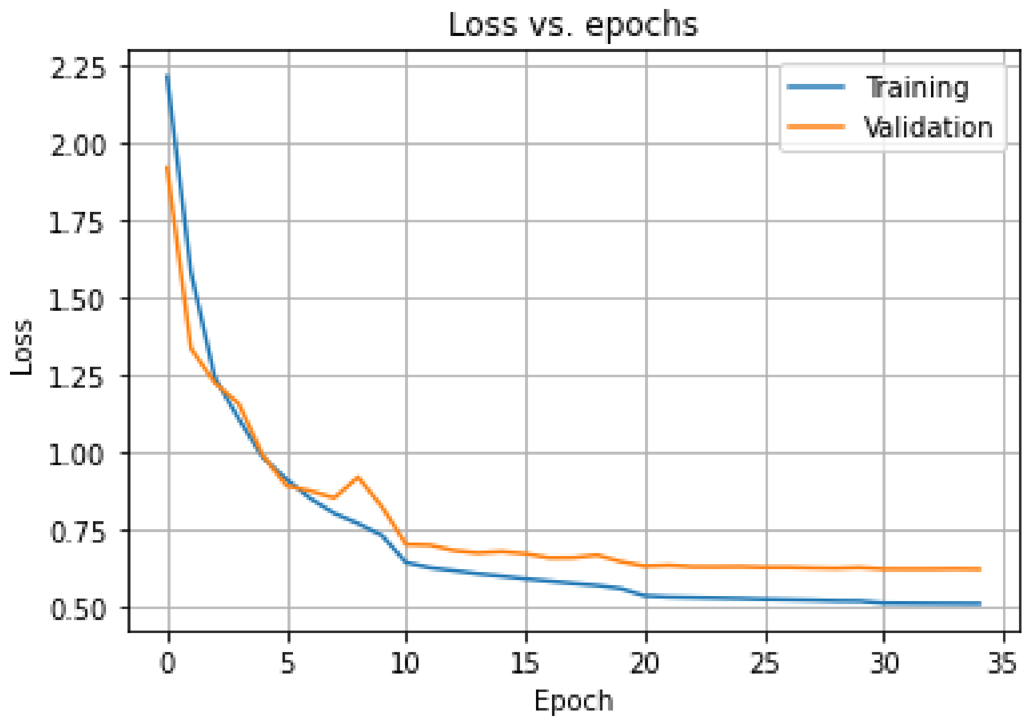
[16]:

```
plt.plot(history_dnn.history['accuracy'])
plt.plot(history_dnn.history['val_accuracy'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.grid()
```

```
plt.show()
```



[17]:
```
plt.plot(history_dnn.history['loss'])
plt.plot(history_dnn.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.grid()
plt.show()
```



[18]:
```
test_loss, test_acc = model_dnn.evaluate(x_test, y_test, verbose=0)
print("Test loss: {:.3f}\nTest accuracy: {:.2f}%".format(test_loss, 100 *␣
↪test_acc))
```

```
Test loss: 0.711
Test accuracy: 78.78%
```

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.

[19]:
```python
model_cnn = Sequential([
    tf.keras.layers.Conv2D(input_shape=x_train.shape[1:], activation='relu',␣
 ↪filters=32, kernel_size=(3,3), padding='same', name='Conv_1'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2), name='Pool_1'),
    tf.keras.layers.BatchNormalization(name='Batch_Norm_1'),
    tf.keras.layers.Conv2D(filters=32, kernel_size=(3,3), padding='valid',␣
 ↪activation='relu', name='Conv_2'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2), name='Pool_2'),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(3,3), padding='valid',␣
 ↪activation='relu', name='Conv_3'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2), name='Pool_3'),
    tf.keras.layers.Flatten(name='Flatten'),
    tf.keras.layers.Dense(units=256, activation='relu', name='Dense_1'),
    tf.keras.layers.Dropout(rate=0.5, name='Dropout_1'),
    tf.keras.layers.Dense(units=10, activation='softmax', name='Output'),
])

opt = tf.keras.optimizers.Adam(learning_rate=0.005)

model_cnn.compile(optimizer=opt, loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

[20]:
```python
model_cnn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Conv_1 (Conv2D)              (None, 32, 32, 32)        320
_________________________________________________________________
Pool_1 (MaxPooling2D)        (None, 16, 16, 32)        0
_________________________________________________________________
Batch_Norm_1 (BatchNormaliza (None, 16, 16, 32)        128
_________________________________________________________________
Conv_2 (Conv2D)              (None, 14, 14, 32)        9248
_________________________________________________________________
Pool_2 (MaxPooling2D)        (None, 7, 7, 32)          0
_________________________________________________________________
Conv_3 (Conv2D)              (None, 5, 5, 16)          4624
_________________________________________________________________
Pool_3 (MaxPooling2D)        (None, 2, 2, 16)          0
_________________________________________________________________
Flatten (Flatten)            (None, 64)                0
_________________________________________________________________
Dense_1 (Dense)              (None, 256)               16640
_________________________________________________________________
Dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
```

```
Output (Dense)               (None, 10)              2570
=================================================================
Total params: 33,530
Trainable params: 33,466
Non-trainable params: 64
_________________________________________________________________
```

[ ]:
```
def lr_function_cnn(epoch, lr):
    if (epoch) % 15 == 0:
        return lr/5
    else:
        return lr

callback_list = [
    tf.keras.callbacks.LearningRateScheduler(lr_function_cnn, verbose=1),
    tf.keras.callbacks.EarlyStopping(monitor='accuracy', patience=3),
    tf.keras.callbacks.
 ↪ModelCheckpoint(filepath='checkpoints_best_only_CNN\checkpoint',
                                       save_weights_only=True,
                                       monitor='val_accuracy',
                                       save_best_only=True)
]

history_cnn = model_cnn.fit(x_train, y_train, epochs=50,
                    validation_split = 0.15,
                    batch_size=128,
                    shuffle=True,
                    callbacks=callback_list)
```

```
Train on 62268 samples, validate on 10989 samples

Epoch 00001: LearningRateScheduler reducing learning rate to
0.0009999999776482583.
Epoch 1/50
62268/62268 [==============================] - 4s 69us/sample - loss: 1.1277 -
accuracy: 0.6303 - val_loss: 0.8595 - val_accuracy: 0.8015

Epoch 00002: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 2/50
62268/62268 [==============================] - 2s 36us/sample - loss: 0.6241 -
accuracy: 0.8120 - val_loss: 0.6251 - val_accuracy: 0.8103

Epoch 00003: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 3/50
62268/62268 [==============================] - 2s 36us/sample - loss: 0.5337 -
accuracy: 0.8401 - val_loss: 0.5291 - val_accuracy: 0.8332
```

```
Epoch 00004: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 4/50
62268/62268 [==============================] - 2s 36us/sample - loss: 0.4862 -
accuracy: 0.8536 - val_loss: 0.4588 - val_accuracy: 0.8608

Epoch 00005: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 5/50
62268/62268 [==============================] - 2s 36us/sample - loss: 0.4541 -
accuracy: 0.8626 - val_loss: 0.4627 - val_accuracy: 0.8613

Epoch 00006: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 6/50
62268/62268 [==============================] - 2s 36us/sample - loss: 0.4292 -
accuracy: 0.8706 - val_loss: 0.4215 - val_accuracy: 0.8746

Epoch 00007: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 7/50
62268/62268 [==============================] - 2s 36us/sample - loss: 0.4140 -
accuracy: 0.8752 - val_loss: 0.4223 - val_accuracy: 0.8747

Epoch 00008: LearningRateScheduler reducing learning rate to
0.0009999999310821295.
Epoch 8/50
41728/62268 [===================>…] - ETA: 0s - loss: 0.3998 -
accuracy: 0.8792
```

[ ]:

```
plt.plot(history_cnn.history['accuracy'])
plt.plot(history_cnn.history['val_accuracy'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.grid()
plt.show()
```

[ ]:

```
plt.plot(history_cnn.history['loss'])
plt.plot(history_cnn.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.grid()
```

```
plt.show()
```

[ ]:
```
test_loss, test_acc = model_cnn.evaluate(x_test, y_test, verbose=0)
print("Test loss: {:.3f}\nTest accuracy: {:.2f}%".format(test_loss, 100 *␣
↪test_acc))
```

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.

[ ]:
```
model_dnn.load_weights('checkpoints_best_only_DNN/checkpoint')
model_cnn.load_weights('checkpoints_best_only_CNN/checkpoint')
```

[ ]:
```
n_width = 5
n_height = 3
fig, ax = plt.subplots(nrows=n_height, ncols=n_width)
fig.subplots_adjust(hspace=1.5, wspace=1)
flattened_ax = ax.flatten()
for i in range(0, n_width):
    rand_idx = np.random.randint(x_test.shape[0])
    flattened_ax[i].set_axis_off()
    flattened_ax[i].imshow(x_test[rand_idx, :, :], cmap='gray')
    flattened_ax[i].title.set_text("Label: " + str(np.argmax(y_test[rand_idx])))

    pred_image = x_test[rand_idx, :, :, :]
    pred_image = pred_image[np.newaxis, ...]

    pred_dnn = model_dnn.predict(pred_image)
    pred_cnn = model_cnn.predict(pred_image)

    flattened_ax[i+n_width].bar(range(0, 10), np.squeeze(pred_dnn))
    flattened_ax[i+n_width].title.set_text("DNN_Pred: \n" + str(np.argmax(np.
↪squeeze(pred_dnn))))
    flattened_ax[i+2*n_width].bar(range(0, 10), np.squeeze(pred_cnn))
    flattened_ax[i+2*n_width].title.set_text("CNN_Pred: \n" + str(np.argmax(np.
↪squeeze(pred_cnn))))
```

[ ]:

[ ]: