capstoneProject

December 1, 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.

[17]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
from random import randint
from PIL import Image
```

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.

[18]:
```
# Run this cell to load the dataset

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

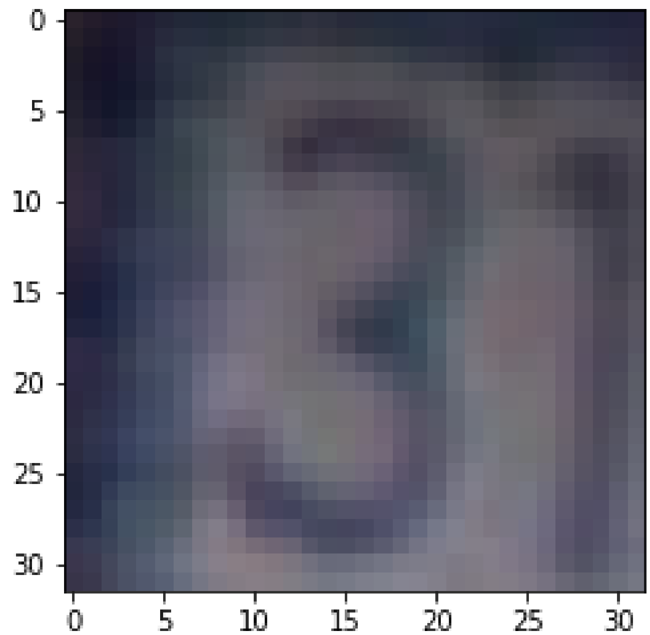
Both `train` and `test` are dictionaries with keys `X` and `y` for the input images and labels respectively.

1.2 1. Inspect and preprocess the dataset

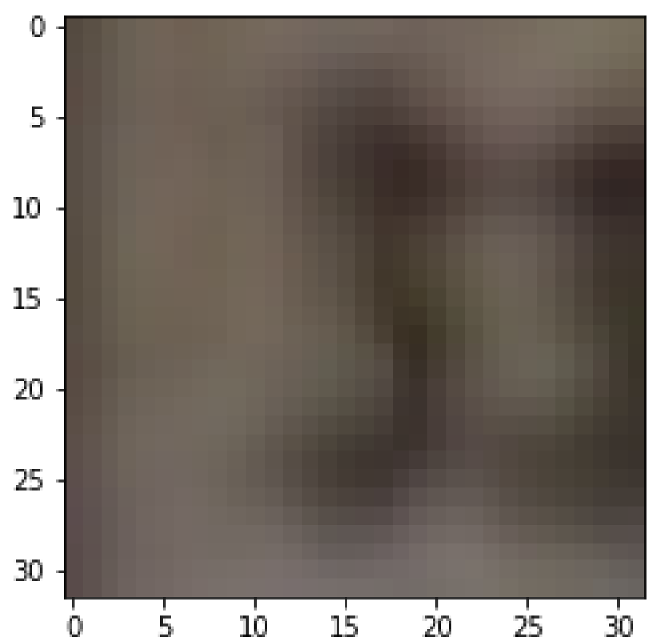
- Extract the training and testing images and labels separately from the train and test dictionaries loaded for you.
- Select a random sample of images and corresponding labels from the dataset (at least 10), and display them in a figure.
- Convert the training and test images to grayscale by taking 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.

[19]:
```
train_img_arr = train['X']
train_img_labels = train['y']
test_img_arr = test['X']
test_img_labels = test['y']
```

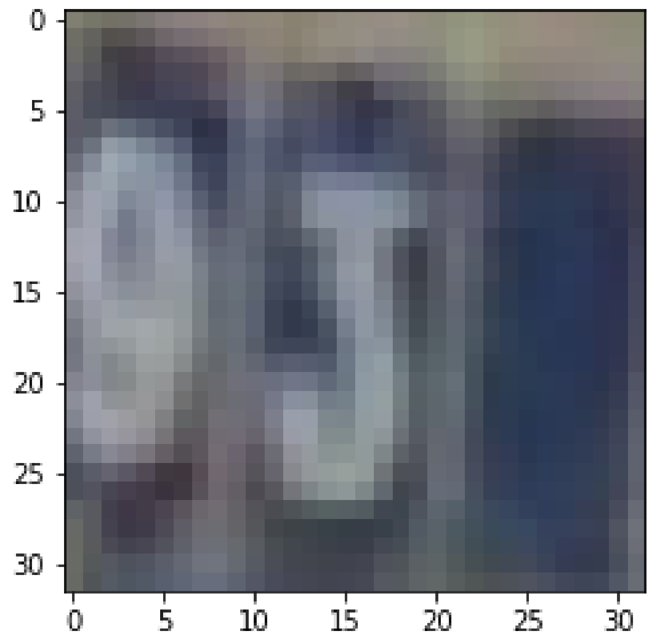
[20]:
```
lenOfTrain = len(train_img_arr)
for x in range(10):
    randomInt = randint(0, lenOfTrain-1)
    plt.imshow(train_img_arr[:,:,:,randomInt])
    plt.show()
    print("img sample label: ", train_img_labels[randomInt])
```



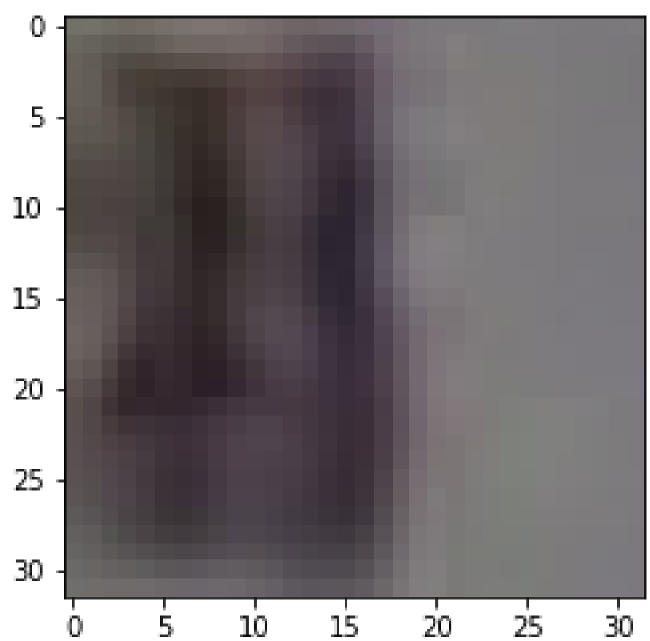
img sample label: [3]



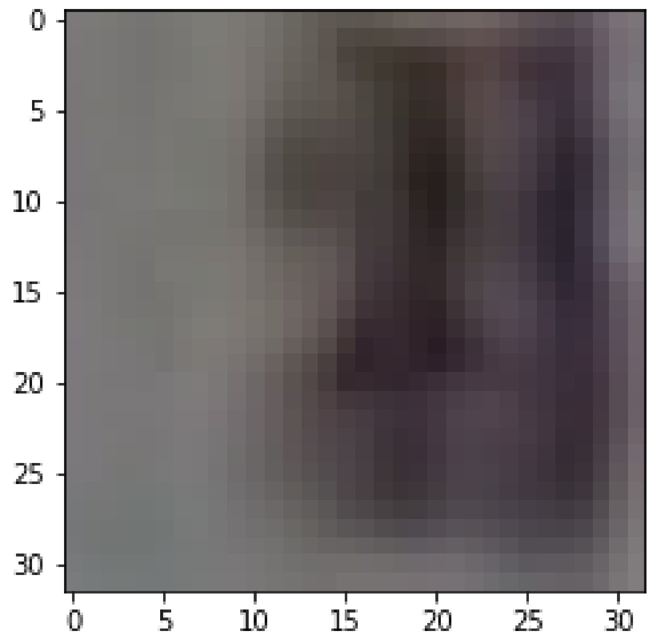
img sample label: [3]



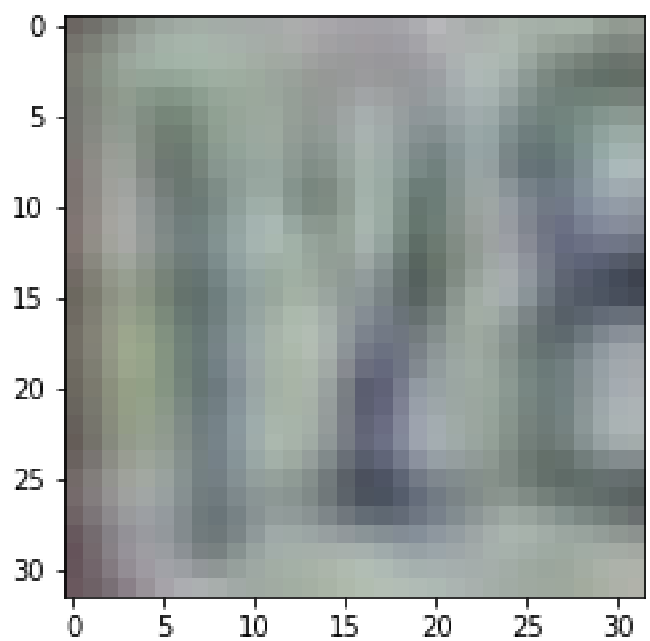
img sample label: [3]



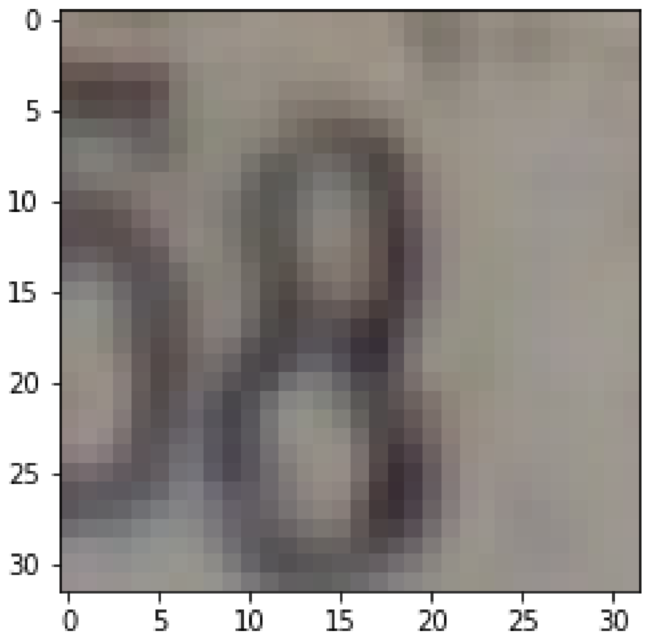
img sample label: [3]



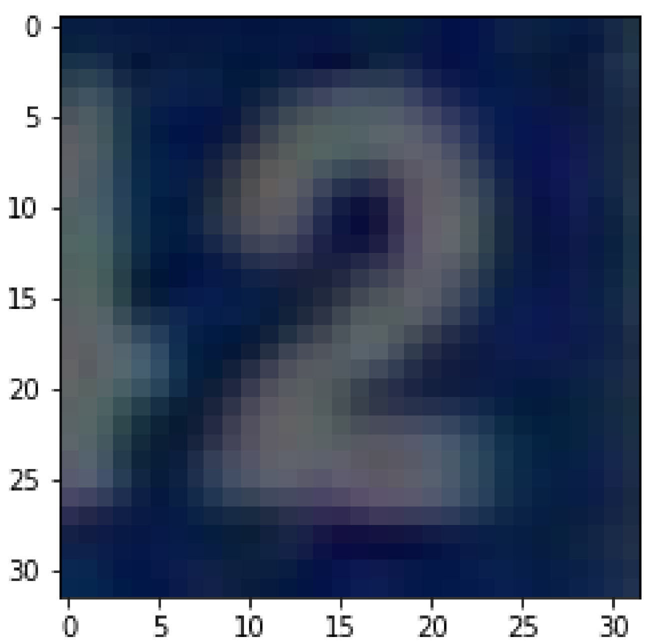
img sample label: [2]



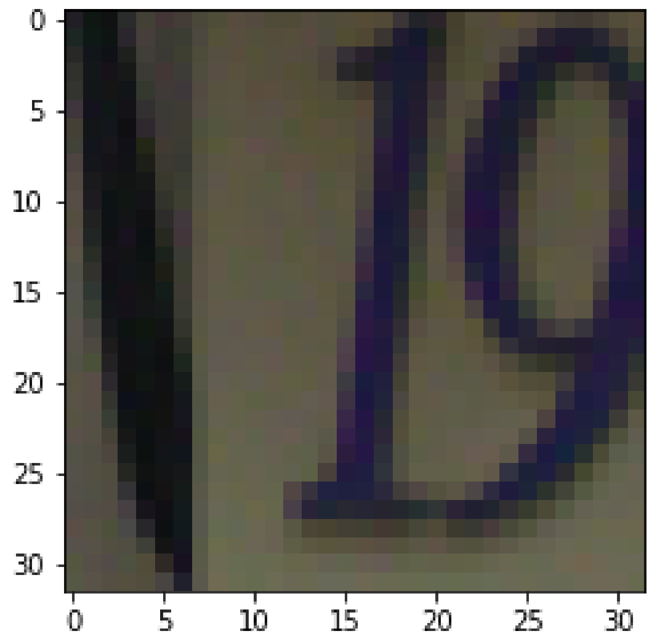
img sample label: [2]



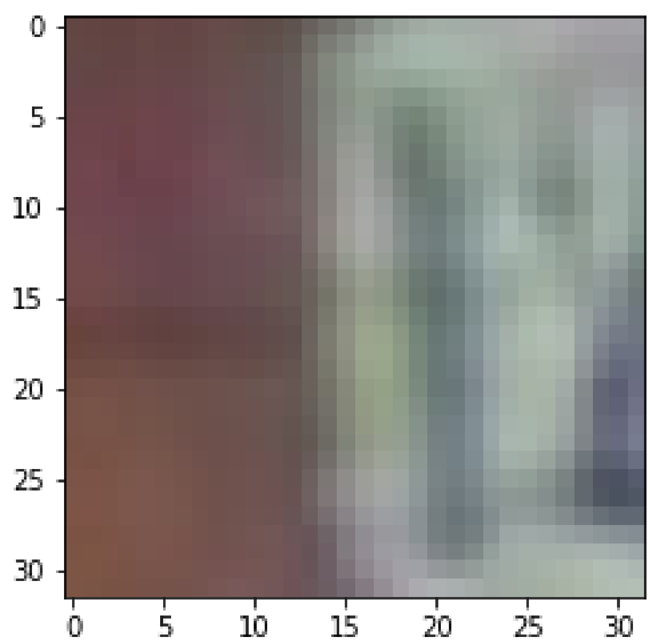
img sample label: [8]



img sample label: [2]



img sample label: [1]



img sample label: [1]

[21]:

```python
# Converts input RBG array to grayscale equivalent

def convert_arr_to_gray(input_array):
    # get chape of train img arr
    original_shape = np.shape(input_array)
    # create gray scale equivalent array with one channel.
    # change the format to have the batch the first parm
    img_arr_gray = np.zeros((original_shape[3], original_shape[0],
 ↪original_shape[1], 1))
    for x in range(np.shape(input_array)[0]):
        for y in range(np.shape(input_array)[1]):
            # convert rbg to grayscale
            gray_arr = ((input_array[x][y][0] + input_array[x][y][1] +
 ↪input_array[x][y][2]) / 3.0) / 255.
            # set it in output arr
            img_arr_gray[:,x,y,0] = gray_arr
    print(img_arr_gray)
    return img_arr_gray

train_img_arr_gray = convert_arr_to_gray(train_img_arr)
print(f"shape: {np.shape(train_img_arr_gray)}")
test_img_arr_gray = convert_arr_to_gray(test_img_arr)
```

```
[[[[0.13202614]
   [0.0745098 ]
   [0.06666667]
   …
   [0.25228758]
   [0.28235294]
   [0.29673203]]

  [[0.13333333]
   [0.07973856]
   [0.06666667]
   …
   [0.16862745]
   [0.23137255]
   [0.28888889]]

  [[0.15555556]
   [0.08496732]
   [0.07320261]
   …
   [0.12679739]
   [0.15816993]
   [0.25359477]]

  …
```

```
 [[0.31633987]
  [0.31372549]
  [0.29542484]
  …
  [0.05098039]
  [0.04444444]
  [0.03921569]]

 [[0.30588235]
  [0.30980392]
  [0.30065359]
  …
  [0.05359477]
  [0.05228758]
  [0.05228758]]

 [[0.31503268]
  [0.31764706]
  [0.30196078]
  …
  [0.06013072]
  [0.05751634]
  [0.04183007]]]


[[[0.28627451]
  [0.29411765]
  [0.27581699]
  …
  [0.30980392]
  [0.30326797]
  [0.30326797]]

 [[0.29150327]
  [0.2875817 ]
  [0.26405229]
  …
  [0.30718954]
  [0.30980392]
  [0.30849673]]

 [[0.28627451]
  [0.27843137]
  [0.21568627]
  …
  [0.31111111]
  [0.30980392]
```

```
  [0.30980392]]

 …

 [[0.01830065]
  [0.01568627]
  [0.00653595]
  …
  [0.0496732 ]
  [0.04575163]
  [0.03006536]]

 [[0.03921569]
  [0.04313725]
  [0.04836601]
  …
  [0.04836601]
  [0.04836601]
  [0.0379085 ]]

 [[0.04575163]
  [0.05098039]
  [0.05882353]
  …
  [0.0627451 ]
  [0.05228758]
  [0.03267974]]]


[[[0.23921569]
  [0.23921569]
  [0.25882353]
  …
  [0.23006536]
  [0.24575163]
  [0.2875817 ]]

 [[0.24052288]
  [0.23137255]
  [0.25098039]
  …
  [0.19869281]
  [0.26405229]
  [0.03529412]]

 [[0.24052288]
  [0.23137255]
  [0.23529412]
```

```
  …
  [0.23529412]
  [0.03137255]
  [0.18039216]]

 …

 [[0.30196078]
  [0.30588235]
  [0.27973856]
  …
  [0.22875817]
  [0.23660131]
  [0.24575163]]

 [[0.30849673]
  [0.29803922]
  [0.29803922]
  …
  [0.22875817]
  [0.22745098]
  [0.23267974]]

 [[0.29542484]
  [0.27712418]
  [0.29673203]
  …
  [0.24313725]
  [0.23660131]
  [0.23921569]]]


…


[[[0.01960784]
  [0.03267974]
  [0.10980392]
  …
  [0.12026144]
  [0.08888889]
  [0.08104575]]

 [[0.02222222]
  [0.03006536]
  [0.10065359]
  …
  [0.04313725]
```

```
  [0.02745098]
  [0.05228758]]

 [[0.05098039]
  [0.0379085 ]
  [0.08888889]
  …
  [0.32418301]
  [0.00915033]
  [0.06013072]]

 …

 [[0.06143791]
  [0.05228758]
  [0.10326797]
  …
  [0.14248366]
  [0.1254902 ]
  [0.11372549]]

 [[0.04183007]
  [0.04313725]
  [0.10718954]
  …
  [0.18300654]
  [0.17385621]
  [0.15424837]]

 [[0.01699346]
  [0.00784314]
  [0.08627451]
  …
  [0.17777778]
  [0.19346405]
  [0.18431373]]]


[[[0.0745098 ]
  [0.13333333]
  [0.19084967]
  …
  [0.2248366 ]
  [0.22222222]
  [0.21960784]]

 [[0.05098039]
  [0.11503268]
```

```
  [0.18431373]
  …
  [0.21699346]
  [0.2130719 ]
  [0.21045752]]

 [[0.00522876]
  [0.07712418]
  [0.16862745]
  …
  [0.22875817]
  [0.21699346]
  [0.20653595]]

 …

 [[0.06013072]
  [0.12156863]
  [0.17254902]
  …
  [0.11372549]
  [0.08627451]
  [0.08627451]]

 [[0.05620915]
  [0.10065359]
  [0.14901961]
  …
  [0.18039216]
  [0.15163399]
  [0.13202614]]

 [[0.07320261]
  [0.10196078]
  [0.1372549 ]
  …
  [0.22222222]
  [0.19738562]
  [0.16732026]]]


[[[0.17385621]
  [0.19346405]
  [0.21176471]
  …
  [0.10196078]
  [0.06666667]
  [0.07843137]]
```

```
  [[0.12941176]
   [0.15294118]
   [0.19084967]
   …
   [0.10588235]
   [0.07843137]
   [0.07843137]]

  [[0.10588235]
   [0.12156863]
   [0.15816993]
   …
   [0.11372549]
   [0.09281046]
   [0.07843137]]

  …

  [[0.22875817]
   [0.21830065]
   [0.20915033]
   …
   [0.09411765]
   [0.04183007]
   [0.06013072]]

  [[0.22614379]
   [0.21699346]
   [0.20915033]
   …
   [0.08235294]
   [0.03006536]
   [0.04444444]]

  [[0.22352941]
   [0.22352941]
   [0.21960784]
   …
   [0.06405229]
   [0.01568627]
   [0.02352941]]]]
shape: (73257, 32, 32, 1)
[[[[0.2627451 ]
   [0.26666667]
   [0.26797386]
   …
   [0.26666667]
```

```
  [0.27058824]
  [0.25228758]]

 [[0.26666667]
  [0.26666667]
  [0.26797386]
  …
  [0.26666667]
  [0.27058824]
  [0.25228758]]

 [[0.26797386]
  [0.26405229]
  [0.27189542]
  …
  [0.27058824]
  [0.27058824]
  [0.25490196]]

 …

 [[0.3254902 ]
  [0.32941176]
  [0.32156863]
  …
  [0.30196078]
  [0.30326797]
  [0.29281046]]

 [[0.31764706]
  [0.32156863]
  [0.31372549]
  …
  [0.30326797]
  [0.30457516]
  [0.29281046]]

 [[0.30980392]
  [0.30980392]
  [0.30196078]
  …
  [0.30065359]
  [0.30065359]
  [0.29542484]]]


[[[0.21960784]
  [0.21699346]
```

```
 [0.2130719 ]
 …
 [0.19738562]
 [0.20522876]
 [0.20784314]]

[[0.24575163]
 [0.24052288]
 [0.2379085 ]
 …
 [0.21699346]
 [0.21437908]
 [0.21437908]]

[[0.27581699]
 [0.27058824]
 [0.26928105]
 …
 [0.24705882]
 [0.23529412]
 [0.22875817]]

…

[[0.29542484]
 [0.27581699]
 [0.25359477]
 …
 [0.07843137]
 [0.08888889]
 [0.12156863]]

[[0.26013072]
 [0.22875817]
 [0.19477124]
 …
 [0.08366013]
 [0.1124183 ]
 [0.15424837]]

[[0.23267974]
 [0.19738562]
 [0.15686275]
 …
 [0.11503268]
 [0.15294118]
 [0.2       ]]]
```

```
[[[0.29150327]
  [0.29673203]
  [0.30849673]
  …
  [0.3254902 ]
  [0.32156863]
  [0.31372549]]

 [[0.29150327]
  [0.29673203]
  [0.30849673]
  …
  [0.33202614]
  [0.32418301]
  [0.31633987]]

 [[0.26797386]
  [0.2745098 ]
  [0.28627451]
  …
  [0.32679739]
  [0.31372549]
  [0.30326797]]

 …

 [[0.25620915]
  [0.25228758]
  [0.24575163]
  …
  [0.21568627]
  [0.21176471]
  [0.21045752]]

 [[0.30718954]
  [0.30588235]
  [0.30588235]
  …
  [0.27189542]
  [0.27189542]
  [0.27189542]]

 [[0.01699346]
  [0.01830065]
  [0.01960784]
  …
  [0.32026144]
```

```
  [0.32156863]
  [0.32287582]]]


…


[[[0.17385621]
  [0.17777778]
  [0.18169935]
  …
  [0.17647059]
  [0.17777778]
  [0.18039216]]

 [[0.2       ]
  [0.20392157]
  [0.20784314]
  …
  [0.18823529]
  [0.19084967]
  [0.19346405]]

 [[0.23267974]
  [0.23660131]
  [0.23921569]
  …
  [0.20784314]
  [0.21045752]
  [0.21437908]]

 …

 [[0.06405229]
  [0.06013072]
  [0.05620915]
  …
  [0.26013072]
  [0.26797386]
  [0.2745098 ]]

 [[0.15686275]
  [0.15686275]
  [0.15947712]
  …
  [0.26405229]
  [0.27058824]
  [0.2745098 ]]
```

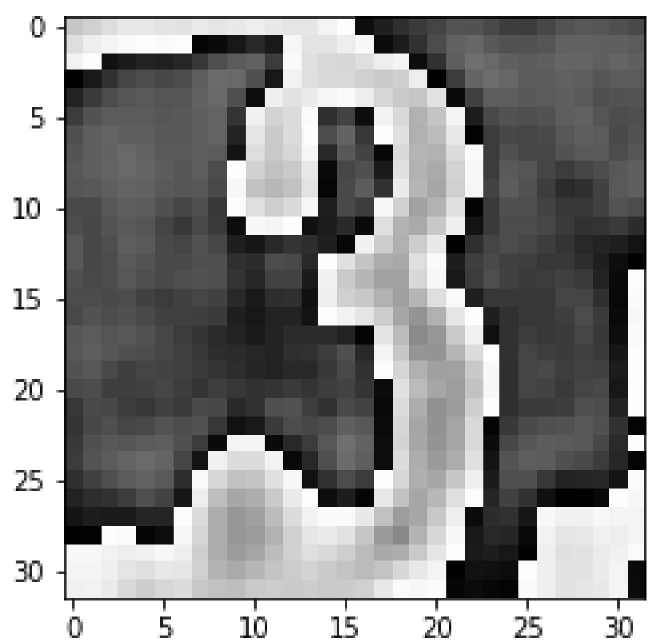
```
 [[0.21699346]
  [0.22091503]
  [0.22745098]
  …
  [0.26797386]
  [0.27189542]
  [0.2745098 ]]]


[[[0.27189542]
  [0.27581699]
  [0.27712418]
  …
  [0.2627451 ]
  [0.26666667]
  [0.27058824]]

 [[0.27320261]
  [0.27712418]
  [0.27843137]
  …
  [0.26143791]
  [0.26666667]
  [0.27058824]]

 [[0.27320261]
  [0.2745098 ]
  [0.27581699]
  …
  [0.27058824]
  [0.28104575]
  [0.28496732]]

 …

 [[0.28104575]
  [0.28627451]
  [0.29542484]
  …
  [0.14117647]
  [0.09542484]
  [0.05490196]]

 [[0.27581699]
  [0.28366013]
  [0.29281046]
  …
```

```
  [0.09150327]
  [0.05751634]
  [0.02614379]]

 [[0.27189542]
  [0.28235294]
  [0.29150327]
  …
  [0.05620915]
  [0.02745098]
  [0.00130719]]]


[[[0.30849673]
  [0.30457516]
  [0.29934641]
  …
  [0.2745098 ]
  [0.2745098 ]
  [0.27581699]]

 [[0.00784314]
  [0.33202614]
  [0.31895425]
  …
  [0.2745098 ]
  [0.26666667]
  [0.26535948]]

 [[0.04444444]
  [0.03006536]
  [0.00653595]
  …
  [0.2745098 ]
  [0.26405229]
  [0.25882353]]

 …

 [[0.05359477]
  [0.05620915]
  [0.05359477]
  …
  [0.28888889]
  [0.28888889]
  [0.28888889]]

 [[0.00261438]
```

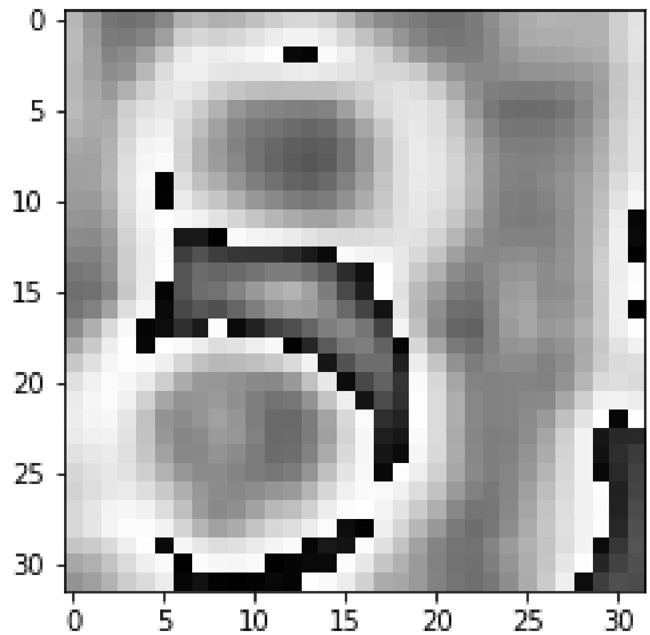
```
 [0.00784314]
 [0.00915033]
 …
 [0.2875817 ]
 [0.29150327]
 [0.29150327]]

[[0.30065359]
 [0.30457516]
 [0.30849673]
 …
 [0.2875817 ]
 [0.29150327]
 [0.29150327]]]]
```

[22]:
```
len_of_train_gray = len(train_img_arr_gray)
print(np.shape(train_img_arr_gray))
for x in range(10):
    randomInt = randint(0, len_of_train_gray-1)
    randomImage = train_img_arr_gray[randomInt,:,:,:]
    plt.imshow(randomImage[:,:,0], cmap='Greys')
    plt.show()
    print("img sample label: ", train_img_labels[randomInt])
```

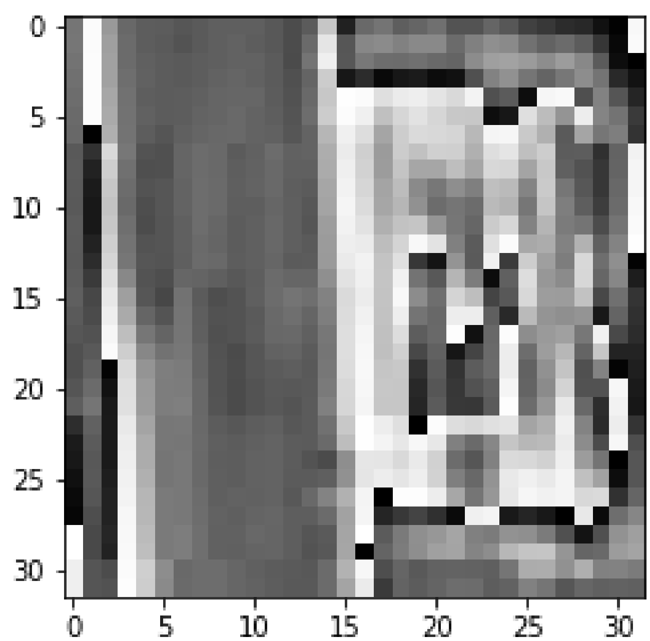
```
(73257, 32, 32, 1)
```



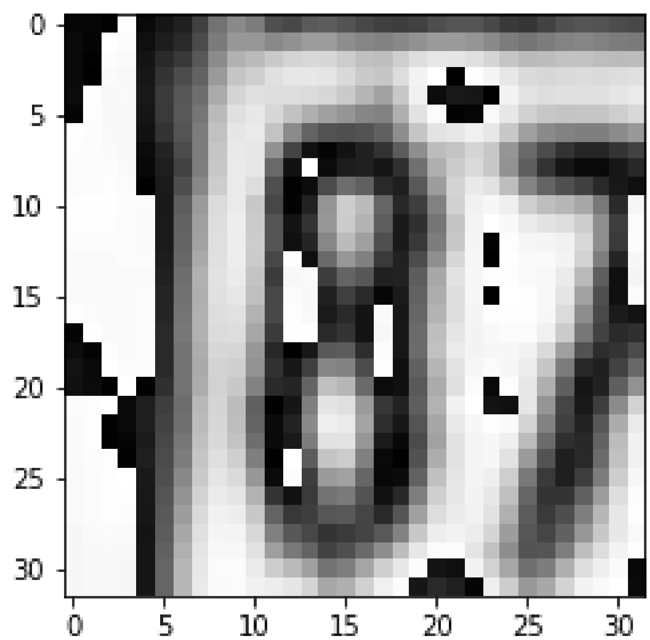

```
img sample label:  [3]
```



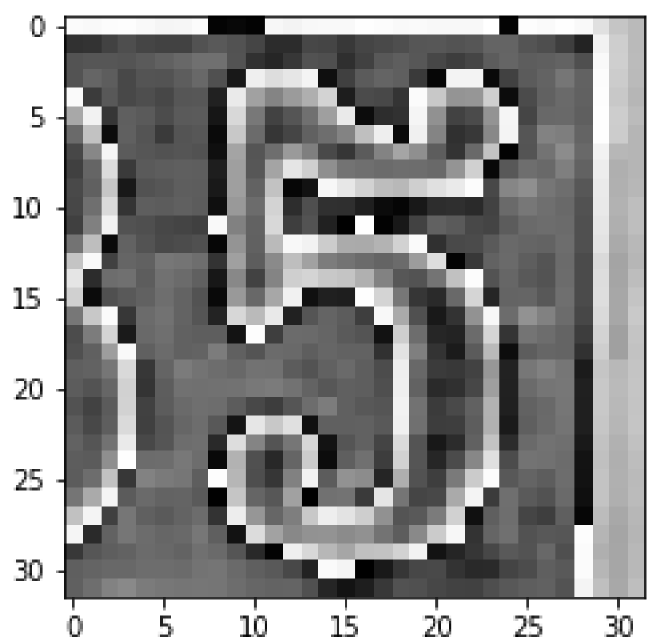
img sample label: [8]



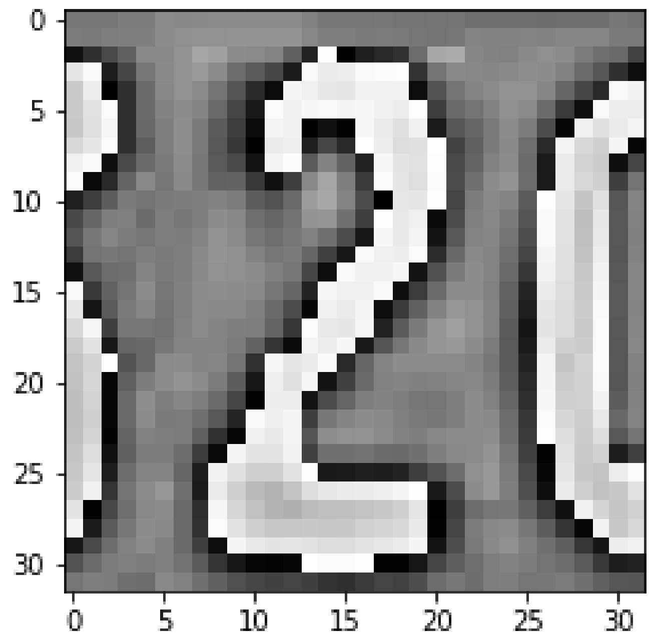
img sample label: [1]



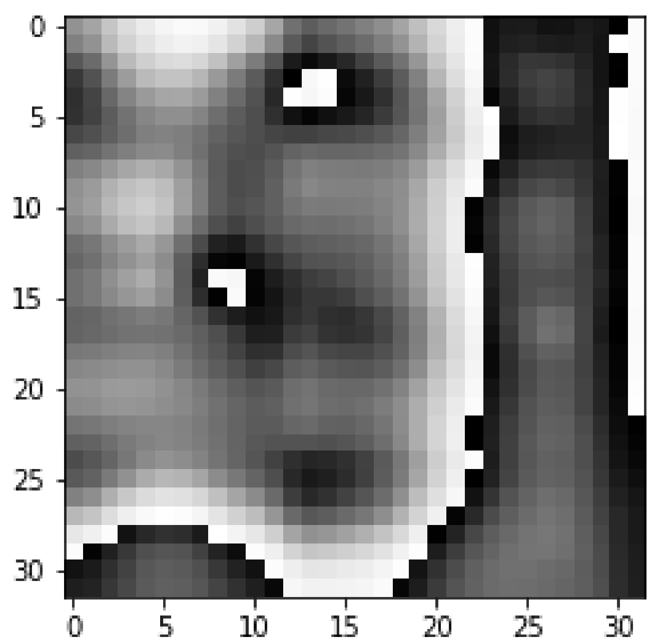
img sample label: [8]



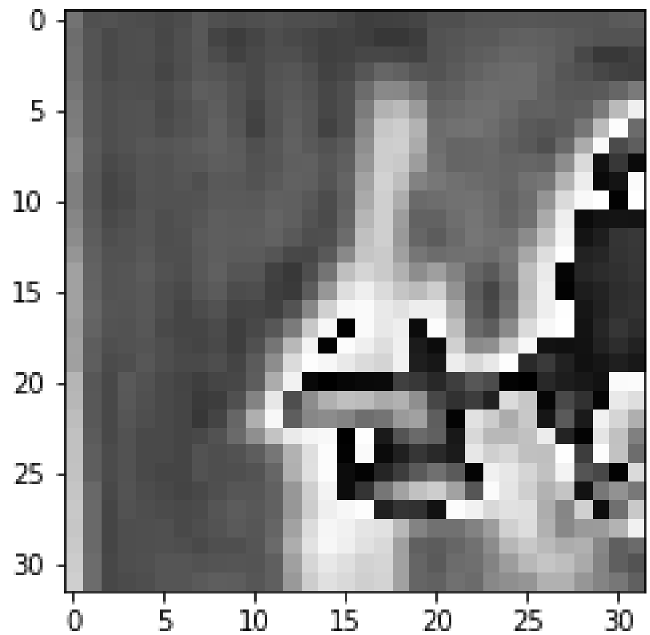
img sample label: [5]



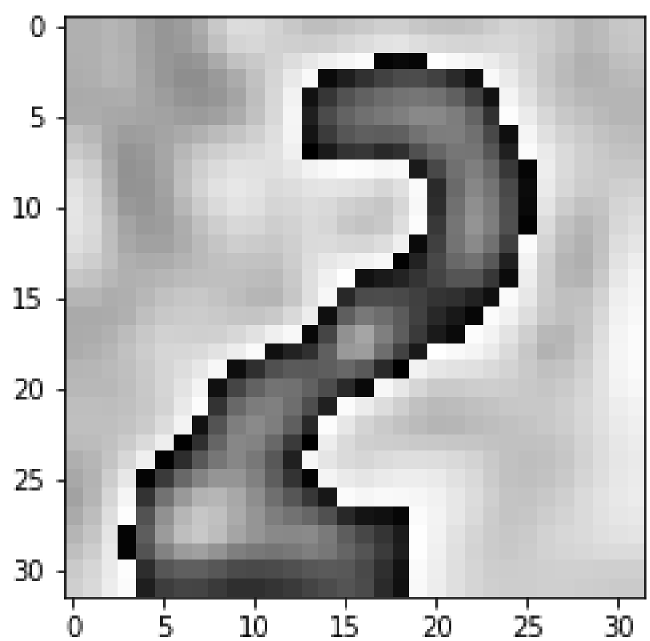

```
img sample label:  [2]
```



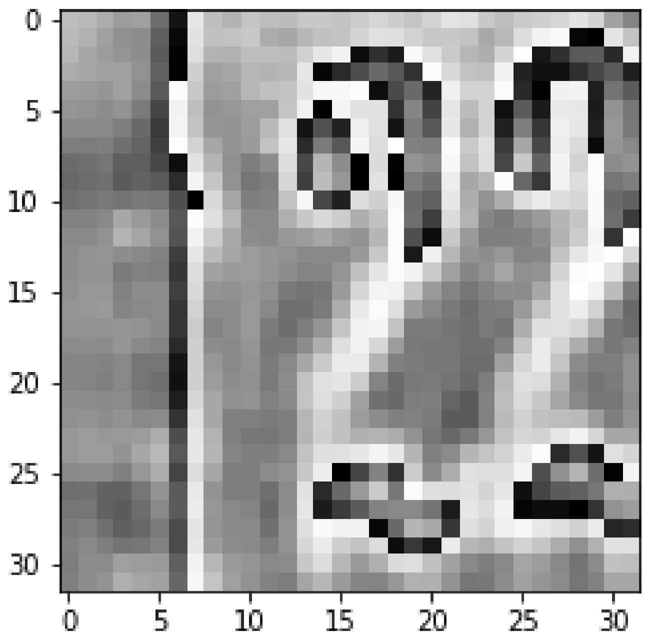

```
img sample label:  [6]
```



img sample label: [4]



img sample label: [2]




```
img sample label:  [2]
```

[ ]:

```

```

[ ]:

```

```

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.

[23]:

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Softmax
```

```python
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

[34]:
```python
# Build the Sequential feedforward neural network model
def get_model(input_shape):
    model = Sequential([
        Flatten(input_shape=input_shape),
        Dense(128, activation='relu', name='layer_1'),
        Dense(64, activation='relu', name='layer_2'),
        Dense(32, activation='relu', name='layer_3'),
        Dense(11, activation='softmax', name='output_layer')
    ])
    return model

input_shape = (np.shape(train_img_arr_gray)[1], np.
 ↪shape(train_img_arr_gray)[2], np.shape(train_img_arr_gray)[3])

model = get_model(input_shape)
```

[35]:
```python
# Print the model summary

model.summary()
```

```
Model: "sequential_3"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 1024)              0

_________________________________________________________________
layer_1 (Dense)              (None, 128)               131200

_________________________________________________________________
layer_2 (Dense)              (None, 64)                8256

_________________________________________________________________
layer_3 (Dense)              (None, 32)                2080

_________________________________________________________________
output_layer (Dense)         (None, 11)                363
=================================================================
Total params: 141,899
Trainable params: 141,899
Non-trainable params: 0

_________________________________________________________________
```

[37]:
```python
#compile the model
acc = tf.keras.metrics.SparseCategoricalAccuracy()
```

```
model.compile(optimizer="Adam", loss='sparse_categorical_crossentropy',␣
↪metrics=[acc])
```

[38]:
```
def get_checkpoint_best_only(checkpoint_best_path):
    """
    Create best checkpoint callback
    """
    checkpoint_best = ModelCheckpoint(filepath=checkpoint_best_path,
                                      save_weights_only=True,
                                      save_freq='epoch',
                                      monitor='sparse_categorical_accuracy',
                                      save_best_only=True,
                                      verbose=1)
    return checkpoint_best

def get_early_stopping():
    """
    Creates EarlyStopping callback that stops training when
    the validation (testing) accuracy has not improved in the last 3 epochs.
    """
    return EarlyStopping(patience=2, monitor='sparse_categorical_accuracy')

checkpoint_best_only_mlp = get_checkpoint_best_only('./
↪checkpoints_mlp_best_only/checkpoint')
early_stopping_mlp = get_early_stopping()
callbacks_mlp = [checkpoint_best_only_mlp, early_stopping_mlp]
```

[39]:
```
# Fit the model
history = model.fit(train_img_arr_gray,
                    train_img_labels,
                    validation_split=0.15,
                    epochs=30,
                    batch_size=32,
                    verbose=2,
                    callbacks=callbacks_mlp)
```

```
Epoch 1/30

Epoch 00001: sparse_categorical_accuracy improved from -inf to 0.18673, saving
model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 2.2466 - sparse_categorical_accuracy: 0.1867 - val_loss:
2.2443 - val_sparse_categorical_accuracy: 0.1881
Epoch 2/30

Epoch 00002: sparse_categorical_accuracy improved from 0.18673 to 0.20515,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 2.1950 - sparse_categorical_accuracy: 0.2051 - val_loss:
2.1313 - val_sparse_categorical_accuracy: 0.2342
```

```
Epoch 3/30

Epoch 00003: sparse_categorical_accuracy improved from 0.20515 to 0.26909,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 2.0418 - sparse_categorical_accuracy: 0.2691 - val_loss:
1.9951 - val_sparse_categorical_accuracy: 0.3028
Epoch 4/30

Epoch 00004: sparse_categorical_accuracy improved from 0.26909 to 0.32707,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.9298 - sparse_categorical_accuracy: 0.3271 - val_loss:
1.9019 - val_sparse_categorical_accuracy: 0.3466
Epoch 5/30

Epoch 00005: sparse_categorical_accuracy improved from 0.32707 to 0.35818,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.8690 - sparse_categorical_accuracy: 0.3582 - val_loss:
1.8827 - val_sparse_categorical_accuracy: 0.3525
Epoch 6/30

Epoch 00006: sparse_categorical_accuracy improved from 0.35818 to 0.37401,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.8294 - sparse_categorical_accuracy: 0.3740 - val_loss:
1.8472 - val_sparse_categorical_accuracy: 0.3671
Epoch 7/30

Epoch 00007: sparse_categorical_accuracy improved from 0.37401 to 0.38933,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.7957 - sparse_categorical_accuracy: 0.3893 - val_loss:
1.8170 - val_sparse_categorical_accuracy: 0.3803
Epoch 8/30

Epoch 00008: sparse_categorical_accuracy improved from 0.38933 to 0.40014,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.7663 - sparse_categorical_accuracy: 0.4001 - val_loss:
1.7945 - val_sparse_categorical_accuracy: 0.3916
Epoch 9/30

Epoch 00009: sparse_categorical_accuracy improved from 0.40014 to 0.41318,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.7423 - sparse_categorical_accuracy: 0.4132 - val_loss:
1.8091 - val_sparse_categorical_accuracy: 0.3883
Epoch 10/30

Epoch 00010: sparse_categorical_accuracy improved from 0.41318 to 0.42097,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.7196 - sparse_categorical_accuracy: 0.4210 - val_loss:
1.7697 - val_sparse_categorical_accuracy: 0.3982
```

```
Epoch 11/30

Epoch 00011: sparse_categorical_accuracy improved from 0.42097 to 0.42943,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6983 - sparse_categorical_accuracy: 0.4294 - val_loss:
1.7699 - val_sparse_categorical_accuracy: 0.4025
Epoch 12/30

Epoch 00012: sparse_categorical_accuracy improved from 0.42943 to 0.43562,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6802 - sparse_categorical_accuracy: 0.4356 - val_loss:
1.7510 - val_sparse_categorical_accuracy: 0.4112
Epoch 13/30

Epoch 00013: sparse_categorical_accuracy improved from 0.43562 to 0.44498,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6633 - sparse_categorical_accuracy: 0.4450 - val_loss:
1.7514 - val_sparse_categorical_accuracy: 0.4128
Epoch 14/30

Epoch 00014: sparse_categorical_accuracy improved from 0.44498 to 0.44747,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6509 - sparse_categorical_accuracy: 0.4475 - val_loss:
1.7235 - val_sparse_categorical_accuracy: 0.4200
Epoch 15/30

Epoch 00015: sparse_categorical_accuracy improved from 0.44747 to 0.45484,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6366 - sparse_categorical_accuracy: 0.4548 - val_loss:
1.7256 - val_sparse_categorical_accuracy: 0.4204
Epoch 16/30

Epoch 00016: sparse_categorical_accuracy improved from 0.45484 to 0.45648,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6256 - sparse_categorical_accuracy: 0.4565 - val_loss:
1.7474 - val_sparse_categorical_accuracy: 0.4183
Epoch 17/30

Epoch 00017: sparse_categorical_accuracy improved from 0.45648 to 0.46274,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6134 - sparse_categorical_accuracy: 0.4627 - val_loss:
1.7137 - val_sparse_categorical_accuracy: 0.4282
Epoch 18/30

Epoch 00018: sparse_categorical_accuracy improved from 0.46274 to 0.46619,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.6060 - sparse_categorical_accuracy: 0.4662 - val_loss:
1.7100 - val_sparse_categorical_accuracy: 0.4299
```

```
Epoch 19/30

Epoch 00019: sparse_categorical_accuracy improved from 0.46619 to 0.46838,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5976 - sparse_categorical_accuracy: 0.4684 - val_loss:
1.7163 - val_sparse_categorical_accuracy: 0.4245
Epoch 20/30

Epoch 00020: sparse_categorical_accuracy improved from 0.46838 to 0.46939,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5894 - sparse_categorical_accuracy: 0.4694 - val_loss:
1.7073 - val_sparse_categorical_accuracy: 0.4300
Epoch 21/30

Epoch 00021: sparse_categorical_accuracy improved from 0.46939 to 0.47344,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5813 - sparse_categorical_accuracy: 0.4734 - val_loss:
1.7173 - val_sparse_categorical_accuracy: 0.4271
Epoch 22/30

Epoch 00022: sparse_categorical_accuracy improved from 0.47344 to 0.47606,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5746 - sparse_categorical_accuracy: 0.4761 - val_loss:
1.7125 - val_sparse_categorical_accuracy: 0.4272
Epoch 23/30

Epoch 00023: sparse_categorical_accuracy improved from 0.47606 to 0.47948,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5713 - sparse_categorical_accuracy: 0.4795 - val_loss:
1.7145 - val_sparse_categorical_accuracy: 0.4255
Epoch 24/30

Epoch 00024: sparse_categorical_accuracy did not improve from 0.47948
1946/1946 - 2s - loss: 1.5655 - sparse_categorical_accuracy: 0.4782 - val_loss:
1.6908 - val_sparse_categorical_accuracy: 0.4336
Epoch 25/30

Epoch 00025: sparse_categorical_accuracy improved from 0.47948 to 0.48280,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5572 - sparse_categorical_accuracy: 0.4828 - val_loss:
1.7005 - val_sparse_categorical_accuracy: 0.4343
Epoch 26/30

Epoch 00026: sparse_categorical_accuracy improved from 0.48280 to 0.48490,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5539 - sparse_categorical_accuracy: 0.4849 - val_loss:
1.7110 - val_sparse_categorical_accuracy: 0.4290
Epoch 27/30
```

```
Epoch 00027: sparse_categorical_accuracy improved from 0.48490 to 0.48662,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5481 - sparse_categorical_accuracy: 0.4866 - val_loss:
1.6906 - val_sparse_categorical_accuracy: 0.4344
Epoch 28/30

Epoch 00028: sparse_categorical_accuracy did not improve from 0.48662
1946/1946 - 2s - loss: 1.5445 - sparse_categorical_accuracy: 0.4857 - val_loss:
1.7195 - val_sparse_categorical_accuracy: 0.4278
Epoch 29/30

Epoch 00029: sparse_categorical_accuracy improved from 0.48662 to 0.48934,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5387 - sparse_categorical_accuracy: 0.4893 - val_loss:
1.6810 - val_sparse_categorical_accuracy: 0.4387
Epoch 30/30

Epoch 00030: sparse_categorical_accuracy improved from 0.48934 to 0.49035,
saving model to ./checkpoints_mlp_best_only/checkpoint
1946/1946 - 2s - loss: 1.5345 - sparse_categorical_accuracy: 0.4903 - val_loss:
1.6972 - val_sparse_categorical_accuracy: 0.4345
```

[40]:

```python
fig = plt.figure(figsize=(12, 5))

fig.add_subplot(121)

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

fig.add_subplot(122)

plt.plot(history.history['sparse_categorical_accuracy'])
plt.plot(history.history['val_sparse_categorical_accuracy'])
plt.title('Accuracy vs. epochs for MLP Neural Network Classifier')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')

plt.show()
```


```
[41]: # Compute and display the loss and accuracy of the trained model on the test␣
      ↪set.
      test_loss, test_accuracy = model.evaluate(test_img_arr_gray, test_img_labels)
      print(f"Test loss: {test_loss}")
      print(f"Test accuracy: {test_accuracy}")
```

```
814/814 [==============================] - 0s 552us/step - loss: 1.7390 -
sparse_categorical_accuracy: 0.4195
Test loss: 1.739003300666809
Test accuracy: 0.41952213644981384
```

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.

[42]:
```
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D,␣
↪BatchNormalization, Dropout
from tensorflow.keras import regularizers
```

[47]:
```
def create_model_cnn(input_shape, wd, rate):
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', padding='SAME',␣
↪input_shape=input_shape),
        MaxPooling2D((2,2)),
        Dense(32, kernel_regularizer=regularizers.l2(wd), activation='relu'),
        BatchNormalization(),
        Dropout(rate),
        Dense(16, kernel_regularizer=regularizers.l2(wd), activation='relu'),
        Flatten(),
        Dense(11, activation='softmax', name='output_layer')])
    return model
# create the CNN model
model_cnn = create_model_cnn(input_shape, 1e-5, 0.3)
# Print the model summary
model_cnn.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 32, 32, 32)        320
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dense_4 (Dense)              (None, 16, 16, 32)        1056
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
dropout_2 (Dropout)          (None, 16, 16, 32)        0
_________________________________________________________________
dense_5 (Dense)              (None, 16, 16, 16)        528
_________________________________________________________________
flatten_5 (Flatten)          (None, 4096)              0
_________________________________________________________________
output_layer (Dense)         (None, 11)                45067
=================================================================
Total params: 47,099
Trainable params: 47,035
Non-trainable params: 64
_________________________________________________________________
```

[48]:
```
acc = tf.keras.metrics.SparseCategoricalAccuracy()
model_cnn.compile(optimizer="Adam", loss='sparse_categorical_crossentropy',␣
↪metrics=[acc])
# callbacks
checkpoint_best_only_cnn = get_checkpoint_best_only('./
↪checkpoints_cnn_best_only/checkpoint')
early_stopping_cnn = get_early_stopping()
callbacks_cnn = [checkpoint_best_only_cnn, early_stopping_cnn]
```

[49]:
```
# Fit the model
history_acc = model_cnn.fit(train_img_arr_gray,
                    train_img_labels,
                    validation_split=0.15,
                    epochs=30,
                    batch_size=64,
                    verbose=2,
                    callbacks=callbacks_cnn)
```

```
Epoch 1/30

Epoch 00001: sparse_categorical_accuracy improved from -inf to 0.67643, saving
model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 25s - loss: 1.0022 - sparse_categorical_accuracy: 0.6764 - val_loss:
0.7903 - val_sparse_categorical_accuracy: 0.7559
Epoch 2/30

Epoch 00002: sparse_categorical_accuracy improved from 0.67643 to 0.79262,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.6800 - sparse_categorical_accuracy: 0.7926 - val_loss:
0.7206 - val_sparse_categorical_accuracy: 0.7826
Epoch 3/30

Epoch 00003: sparse_categorical_accuracy improved from 0.79262 to 0.80647,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 25s - loss: 0.6373 - sparse_categorical_accuracy: 0.8065 - val_loss:
0.7093 - val_sparse_categorical_accuracy: 0.7847
Epoch 4/30

Epoch 00004: sparse_categorical_accuracy improved from 0.80647 to 0.81461,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.6081 - sparse_categorical_accuracy: 0.8146 - val_loss:
0.6431 - val_sparse_categorical_accuracy: 0.8096
Epoch 5/30

Epoch 00005: sparse_categorical_accuracy improved from 0.81461 to 0.82285,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.5855 - sparse_categorical_accuracy: 0.8228 - val_loss:
0.6709 - val_sparse_categorical_accuracy: 0.7987
```

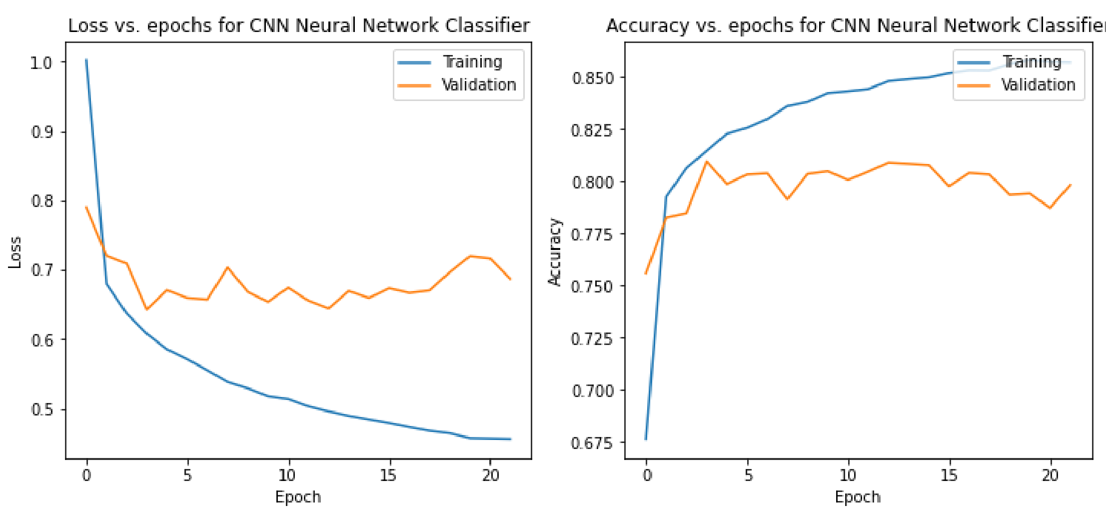
```
Epoch 6/30

Epoch 00006: sparse_categorical_accuracy improved from 0.82285 to 0.82569,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.5714 - sparse_categorical_accuracy: 0.8257 - val_loss:
0.6596 - val_sparse_categorical_accuracy: 0.8033
Epoch 7/30

Epoch 00007: sparse_categorical_accuracy improved from 0.82569 to 0.82986,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.5552 - sparse_categorical_accuracy: 0.8299 - val_loss:
0.6572 - val_sparse_categorical_accuracy: 0.8039
Epoch 8/30

Epoch 00008: sparse_categorical_accuracy improved from 0.82986 to 0.83613,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 27s - loss: 0.5386 - sparse_categorical_accuracy: 0.8361 - val_loss:
0.7037 - val_sparse_categorical_accuracy: 0.7914
Epoch 9/30

Epoch 00009: sparse_categorical_accuracy improved from 0.83613 to 0.83814,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 25s - loss: 0.5294 - sparse_categorical_accuracy: 0.8381 - val_loss:
0.6684 - val_sparse_categorical_accuracy: 0.8036
Epoch 10/30

Epoch 00010: sparse_categorical_accuracy improved from 0.83814 to 0.84234,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.5181 - sparse_categorical_accuracy: 0.8423 - val_loss:
0.6539 - val_sparse_categorical_accuracy: 0.8049
Epoch 11/30

Epoch 00011: sparse_categorical_accuracy improved from 0.84234 to 0.84323,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 27s - loss: 0.5140 - sparse_categorical_accuracy: 0.8432 - val_loss:
0.6745 - val_sparse_categorical_accuracy: 0.8007
Epoch 12/30

Epoch 00012: sparse_categorical_accuracy improved from 0.84323 to 0.84427,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 27s - loss: 0.5035 - sparse_categorical_accuracy: 0.8443 - val_loss:
0.6556 - val_sparse_categorical_accuracy: 0.8047
Epoch 13/30

Epoch 00013: sparse_categorical_accuracy improved from 0.84427 to 0.84816,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 27s - loss: 0.4961 - sparse_categorical_accuracy: 0.8482 - val_loss:
0.6445 - val_sparse_categorical_accuracy: 0.8090
```

```
Epoch 14/30

Epoch 00014: sparse_categorical_accuracy improved from 0.84816 to 0.84910,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 28s - loss: 0.4897 - sparse_categorical_accuracy: 0.8491 - val_loss:
0.6697 - val_sparse_categorical_accuracy: 0.8084
Epoch 15/30

Epoch 00015: sparse_categorical_accuracy improved from 0.84910 to 0.84992,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.4845 - sparse_categorical_accuracy: 0.8499 - val_loss:
0.6595 - val_sparse_categorical_accuracy: 0.8078
Epoch 16/30

Epoch 00016: sparse_categorical_accuracy improved from 0.84992 to 0.85196,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 26s - loss: 0.4793 - sparse_categorical_accuracy: 0.8520 - val_loss:
0.6737 - val_sparse_categorical_accuracy: 0.7977
Epoch 17/30

Epoch 00017: sparse_categorical_accuracy improved from 0.85196 to 0.85333,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 27s - loss: 0.4736 - sparse_categorical_accuracy: 0.8533 - val_loss:
0.6670 - val_sparse_categorical_accuracy: 0.8041
Epoch 18/30

Epoch 00018: sparse_categorical_accuracy did not improve from 0.85333
973/973 - 27s - loss: 0.4683 - sparse_categorical_accuracy: 0.8532 - val_loss:
0.6705 - val_sparse_categorical_accuracy: 0.8033
Epoch 19/30

Epoch 00019: sparse_categorical_accuracy improved from 0.85333 to 0.85635,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 27s - loss: 0.4648 - sparse_categorical_accuracy: 0.8563 - val_loss:
0.6967 - val_sparse_categorical_accuracy: 0.7936
Epoch 20/30

Epoch 00020: sparse_categorical_accuracy improved from 0.85635 to 0.85813,
saving model to ./checkpoints_cnn_best_only/checkpoint
973/973 - 27s - loss: 0.4575 - sparse_categorical_accuracy: 0.8581 - val_loss:
0.7199 - val_sparse_categorical_accuracy: 0.7942
Epoch 21/30

Epoch 00021: sparse_categorical_accuracy did not improve from 0.85813
973/973 - 27s - loss: 0.4569 - sparse_categorical_accuracy: 0.8573 - val_loss:
0.7166 - val_sparse_categorical_accuracy: 0.7872
Epoch 22/30
```

```
Epoch 00022: sparse_categorical_accuracy did not improve from 0.85813
973/973 - 27s - loss: 0.4563 - sparse_categorical_accuracy: 0.8571 - val_loss:
0.6871 - val_sparse_categorical_accuracy: 0.7983
```

[50]:

```
fig = plt.figure(figsize=(12, 5))

fig.add_subplot(121)

plt.plot(history_acc.history['loss'])
plt.plot(history_acc.history['val_loss'])
plt.title('Loss vs. epochs for CNN Neural Network Classifier')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')

fig.add_subplot(122)

plt.plot(history_acc.history['sparse_categorical_accuracy'])
plt.plot(history_acc.history['val_sparse_categorical_accuracy'])
plt.title('Accuracy vs. epochs for CNN Neural Network Classifier')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')

plt.show()
```



[51]:

```
# Compute and display the loss and accuracy of the trained model on the test␣
↪set.
test_loss, test_accuracy = model_cnn.evaluate(test_img_arr_gray,␣
↪test_img_labels)
```

```
print(f"Test loss: {test_loss}")
print(f"Test accuracy: {test_accuracy}")
```

```
814/814 [==============================] - 3s 4ms/step - loss: 0.7880 -
sparse_categorical_accuracy: 0.7708
Test loss: 0.788002610206604
Test accuracy: 0.7708205580711365
```

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.

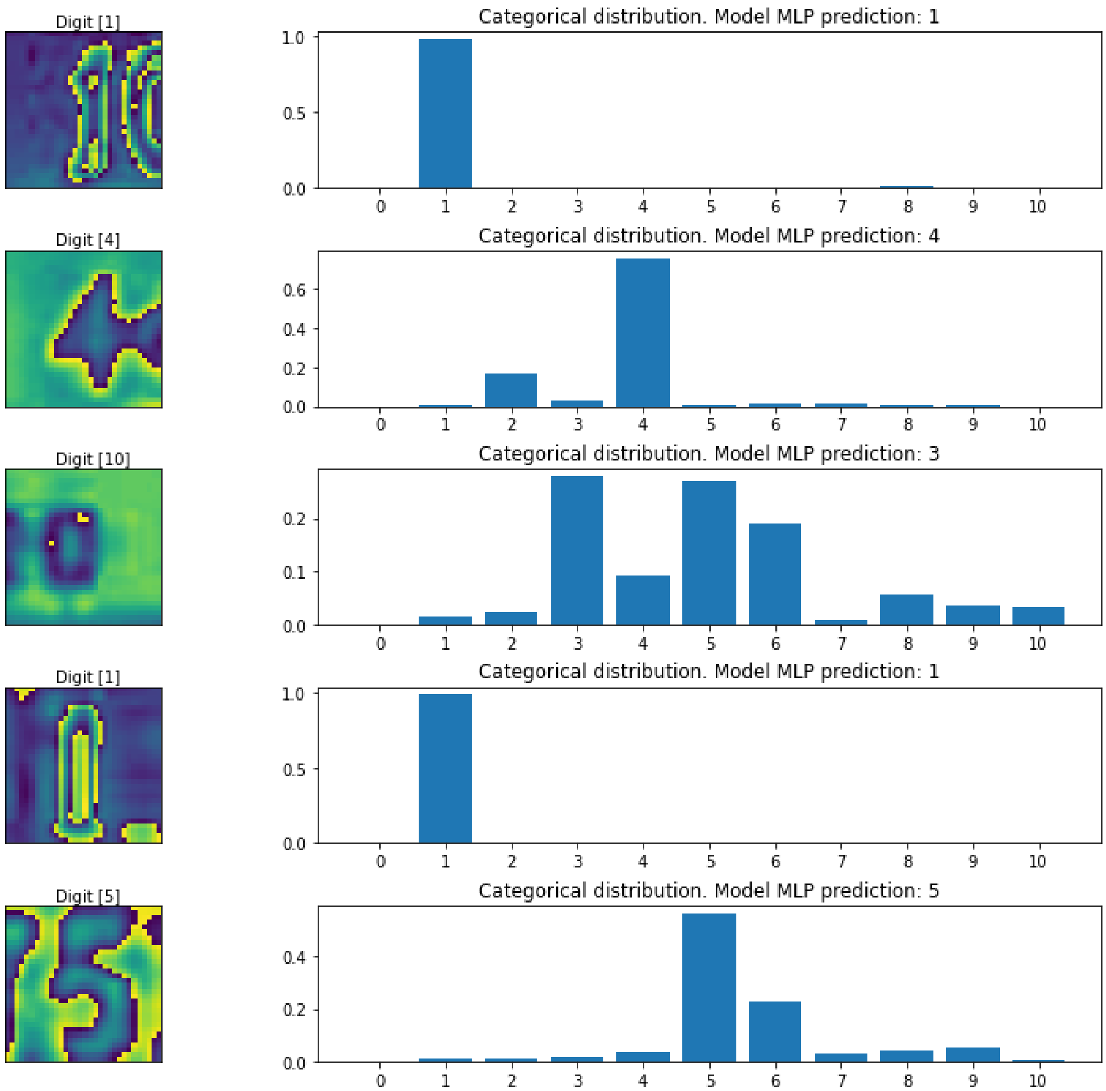
[52]:
```
# get best weights for the MLP model
model.load_weights('checkpoints_mlp_best_only/checkpoint')
# get the beset weights for the CNN model
model_cnn.load_weights('checkpoints_cnn_best_only/checkpoint')
```

[52]:
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x16282ae50>
```

[53]:
```
# get MLP and CNN model predictions on randomly selected test images

num_test_images = test_img_arr_gray.shape[0]

random_inx = np.random.choice(num_test_images, 5)
random_test_images_gray = test_img_arr_gray[random_inx, ...]
random_test_labels = test_img_labels[random_inx, ...]

predictions_mlp = model.predict(random_test_images_gray)
predictions_cnn = model_cnn.predict(random_test_images_gray)
```

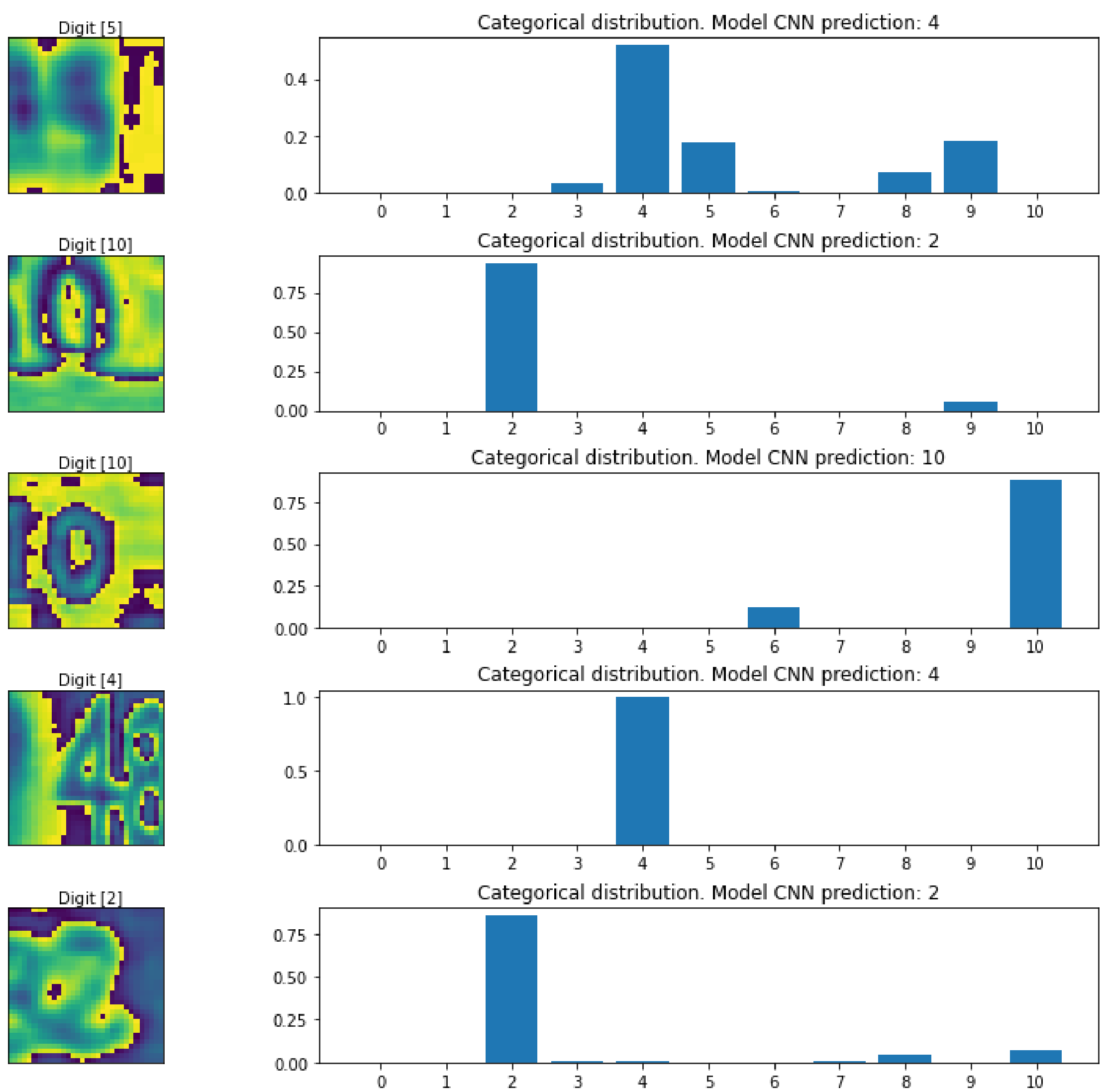
[80]:
```
# MLP predictions
fig, axes = plt.subplots(5, 2, figsize=(16, 12))

fig.subplots_adjust(hspace=0.4, wspace=-0.2)

for i, (prediction_mlp, image, label) in enumerate(zip(predictions_mlp,␣
↪random_test_images_gray, random_test_labels)):
    axes[i, 0].imshow(np.squeeze(image))
    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(len(prediction_mlp)), prediction_mlp)
    axes[i, 1].set_xticks(np.arange(len(prediction_mlp)))
```

```
    axes[i, 1].set_title(f"Categorical distribution. Model MLP prediction: {np.
 ↪argmax(prediction_mlp)}")

plt.show()
```



[55]:
```
# CNN predictions
fig, axes = plt.subplots(5, 2, figsize=(16, 12))

fig.subplots_adjust(hspace=0.4, wspace=-0.2)

for i, (prediction_cnn, image, label) in enumerate(zip(predictions_cnn,␣
 ↪random_test_images_gray, random_test_labels)):
    axes[i, 0].imshow(np.squeeze(image))
```

```
    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(len(prediction_cnn)), prediction_cnn)
    axes[i, 1].set_xticks(np.arange(len(prediction_cnn)))
    axes[i, 1].set_title(f"Categorical distribution. Model CNN prediction: {np.
↪argmax(prediction_cnn)}")

plt.show()
```



[82]:

```
```

[ ]:

```
```