Capstone Project

August 8, 2021

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.

```
In [2]: 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.

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.

```
In [3]: # 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.

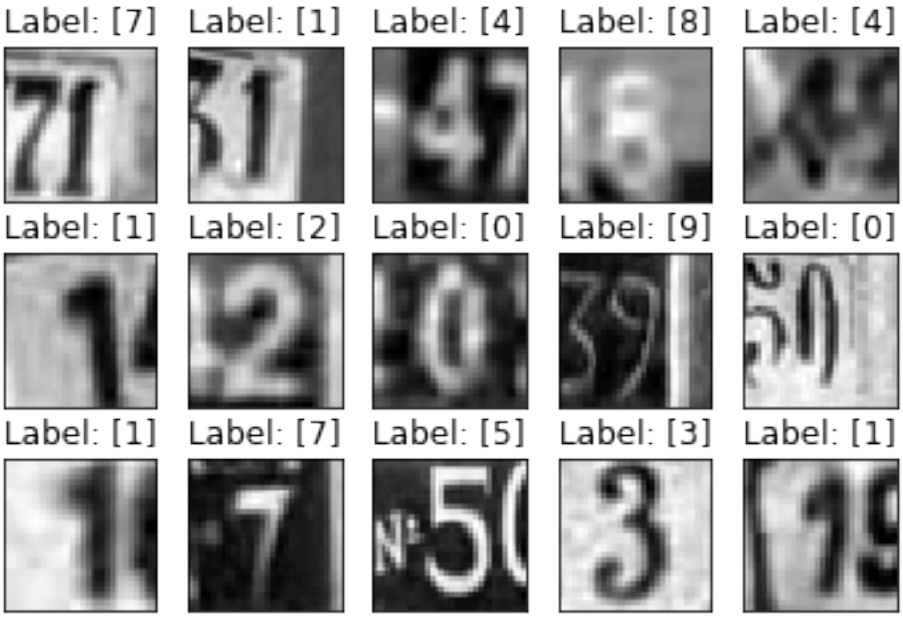
```
In [4]: x_train = np.moveaxis(train['X'], -1, 0)  # make first axis correspond to instances
        y_train = train['y'] - 1  # labels [0, 10)
        x_test = np.moveaxis(test['X'], -1, 0)
        y_test = test['y'] - 1
```

In [6]:
```
import random
from numpy import floor
import matplotlib.pyplot as plt

def display_random_images(image_number: int = 15,
                          data: np.array = x_train, labels: np.array = y_train):
    random_indices = random.sample(range(labels.shape[0]), image_number)
    fig, axes = plt.subplots(int(np.floor(image_number/5)), 5)
    for n_axes, random_index in enumerate(random_indices):
        ax = axes[int(floor(n_axes/5))][n_axes % 5]
        if data.shape[3] == 3:
            ax.imshow(data[random_index,:,:,:], cmap = None)
        else:
            ax.imshow(np.squeeze(data[random_index,:,:,:]), cmap = 'binary')
        ax.set_title(f"Label: {(labels[random_index]+1)%10}")
        ax.set_xticks([])
        ax.set_yticks([])
    return random_indices
```

In [7]:
```
_= display_random_images(15)
```

In [8]:
```
x_train_greyscale = x_train.mean(axis = 3, keepdims=True)/255.0
x_test_greyscale = x_test.mean(axis = 3, keepdims = True)/255.0
```

In [9]:
```
_=display_random_images(15, x_train_greyscale, y_train)
```



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.

```
In [11]: import tensorflow as tf
         from tensorflow.keras.models import Sequential
         from tensorflow.keras.layers import Flatten, Dense
         from tensorflow.keras.callbacks import EarlyStopping

In [41]: model_checkpoint_cnn_callback = ModelCheckpoint('best_weights', monitor="val_loss",
                                                          verbose=0, save_best_only=True, save_
         model_checkpoint_mlp_callback = ModelCheckpoint('best_weights_mlp', monitor="val_loss
                                                          verbose=0, save_best_only=True, save_
```

```
        early_stopping_callback = EarlyStopping(patience = 5, monitor = 'val_accuracy')
```

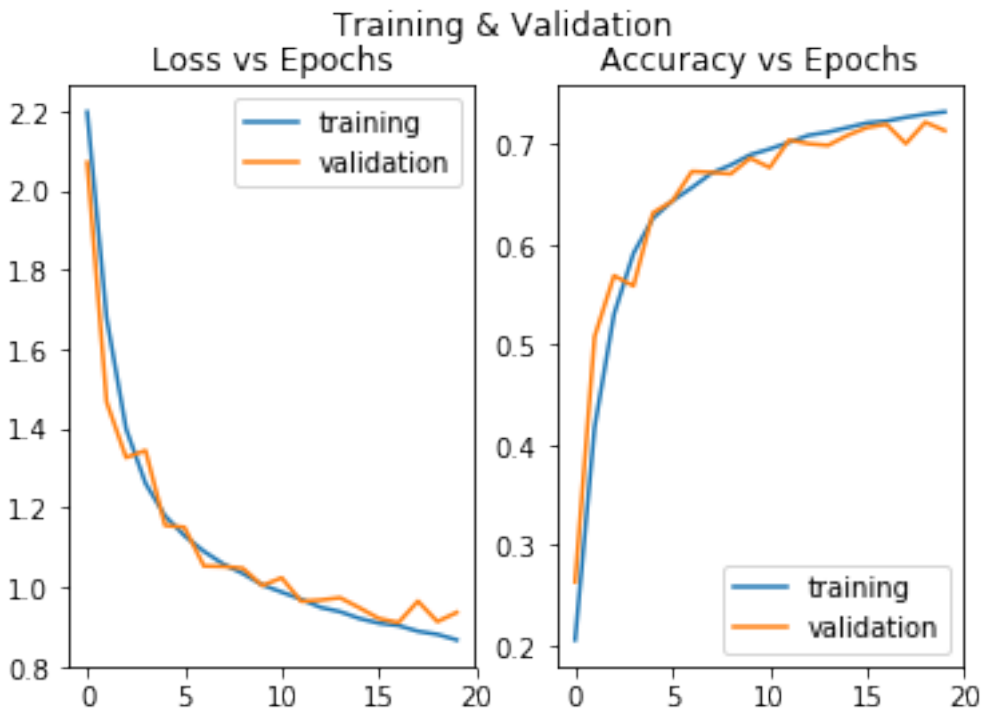
```
In [12]: model_mlp = Sequential([
             Flatten(input_shape=(32, 32, 1)),
             Dense(128, activation='relu'),
             Dense(32, activation='relu'),
             Dense(32, activation='relu'),
             Dense(10, activation ='softmax')
         ])
```

```
In [45]: model_mlp.summary()

Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_8 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_31 (Dense)             (None, 128)               131200
_________________________________________________________________
dense_32 (Dense)             (None, 32)                4128
_________________________________________________________________
dense_33 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_34 (Dense)             (None, 10)                330
=================================================================
Total params: 136,714
Trainable params: 136,714
Non-trainable params: 0
_________________________________________________________________
```

```
In [46]: model_mlp.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', metrics=
         history_mlp = model_mlp.fit(x_train_greyscale, y_train,
                                     epochs=20, validation_split = 0.2,
                                     callbacks=[early_stopping_callback, model_checkpoint_mlp_

Train on 58605 samples, validate on 14652 samples
Epoch 1/20
58605/58605 [==============================] - 33s 561us/sample - loss: 2.1975 - accuracy: 0.20
Epoch 2/20
58605/58605 [==============================] - 32s 546us/sample - loss: 1.6781 - accuracy: 0.41
Epoch 3/20
58605/58605 [==============================] - 32s 546us/sample - loss: 1.3991 - accuracy: 0.53
Epoch 4/20
58605/58605 [==============================] - 32s 549us/sample - loss: 1.2613 - accuracy: 0.59
Epoch 5/20
58605/58605 [==============================] - 32s 548us/sample - loss: 1.1798 - accuracy: 0.62
```

```
Epoch 6/20
58605/58605 [==============================] - 32s 548us/sample - loss: 1.1298 - accuracy: 0.64
Epoch 7/20
58605/58605 [==============================] - 32s 551us/sample - loss: 1.0901 - accuracy: 0.65
Epoch 8/20
58605/58605 [==============================] - 32s 548us/sample - loss: 1.0581 - accuracy: 0.67
Epoch 9/20
58605/58605 [==============================] - 32s 550us/sample - loss: 1.0344 - accuracy: 0.67
Epoch 10/20
58605/58605 [==============================] - 32s 550us/sample - loss: 1.0058 - accuracy: 0.68
Epoch 11/20
58605/58605 [==============================] - 32s 545us/sample - loss: 0.9875 - accuracy: 0.69
Epoch 12/20
58605/58605 [==============================] - 32s 545us/sample - loss: 0.9694 - accuracy: 0.70
Epoch 13/20
58605/58605 [==============================] - 32s 539us/sample - loss: 0.9490 - accuracy: 0.70
Epoch 14/20
58605/58605 [==============================] - 32s 541us/sample - loss: 0.9382 - accuracy: 0.71
Epoch 15/20
58605/58605 [==============================] - 32s 550us/sample - loss: 0.9210 - accuracy: 0.71
Epoch 16/20
58605/58605 [==============================] - 32s 543us/sample - loss: 0.9092 - accuracy: 0.71
Epoch 17/20
58605/58605 [==============================] - 32s 544us/sample - loss: 0.9032 - accuracy: 0.71
Epoch 18/20
58605/58605 [==============================] - 31s 534us/sample - loss: 0.8888 - accuracy: 0.71
Epoch 19/20
58605/58605 [==============================] - 32s 543us/sample - loss: 0.8810 - accuracy: 0.73
Epoch 20/20
58605/58605 [==============================] - 31s 532us/sample - loss: 0.8670 - accuracy: 0.73
```

In [47]:
```
fig, ax = plt.subplots(1, 2)
ax[0].plot(history_mlp.history['loss'], label = 'training')
ax[0].plot(history_mlp.history['val_loss'], label = 'validation')
ax[0].legend()
ax[0].set_title('Loss vs Epochs')

ax[1].plot(history_mlp.history['accuracy'], label = 'training')
ax[1].plot(history_mlp.history['val_accuracy'], label = 'validation')
ax[1].set_title('Accuracy vs Epochs')
ax[1].legend()
_=plt.suptitle('Training & Validation')
```




```
In [49]: test_loss, test_acc = model_mlp.evaluate(x_test_greyscale, y_test, verbose=0)
         print(f"Test set loss: {test_loss:.6f}, test set accuracy: {test_acc:.6f}")
```

```
Test set loss: 1.050020, test set accuracy: 0.684158
```

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.

```
In [13]: from tensorflow.keras.layers import (Conv2D, MaxPooling2D, BatchNormalization, Flatte
         from tensorflow.keras.callbacks import (EarlyStopping, ModelCheckpoint)

In [14]: model_cnn = Sequential([
             Conv2D(input_shape=(32, 32, 1),
                    filters=16, kernel_size=(3, 3),
                    padding='SAME', strides=2, activation='relu'),
             BatchNormalization(),
             MaxPooling2D((3, 3)),
             Conv2D(filters=16, kernel_size=(3, 3),
                    padding='SAME', strides=2, activation='relu'),
             BatchNormalization(),
             MaxPooling2D((3, 3)),
             Conv2D(filters=16, kernel_size=(3, 3),
                    padding='SAME', strides=2, activation='relu'),
             BatchNormalization(),
             Flatten(),
             Dense(128, activation='relu'),
             BatchNormalization(),
             Dropout(0.5),
             Dense(128, activation='relu'),
             BatchNormalization(),
             Dropout(0.5),
             Dense(10, activation='softmax')
         ])

In [33]: model_cnn.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', metrics=
         model_cnn.summary()

Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 16, 16, 16)        160
_________________________________________________________________
batch_normalization (BatchNo (None, 16, 16, 16)        64
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 5, 5, 16)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 3, 3, 16)          2320
_________________________________________________________________
batch_normalization_1 (Batch (None, 3, 3, 16)          64
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 1, 1, 16)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 1, 16)          2320
_________________________________________________________________
batch_normalization_2 (Batch (None, 1, 1, 16)          64
```

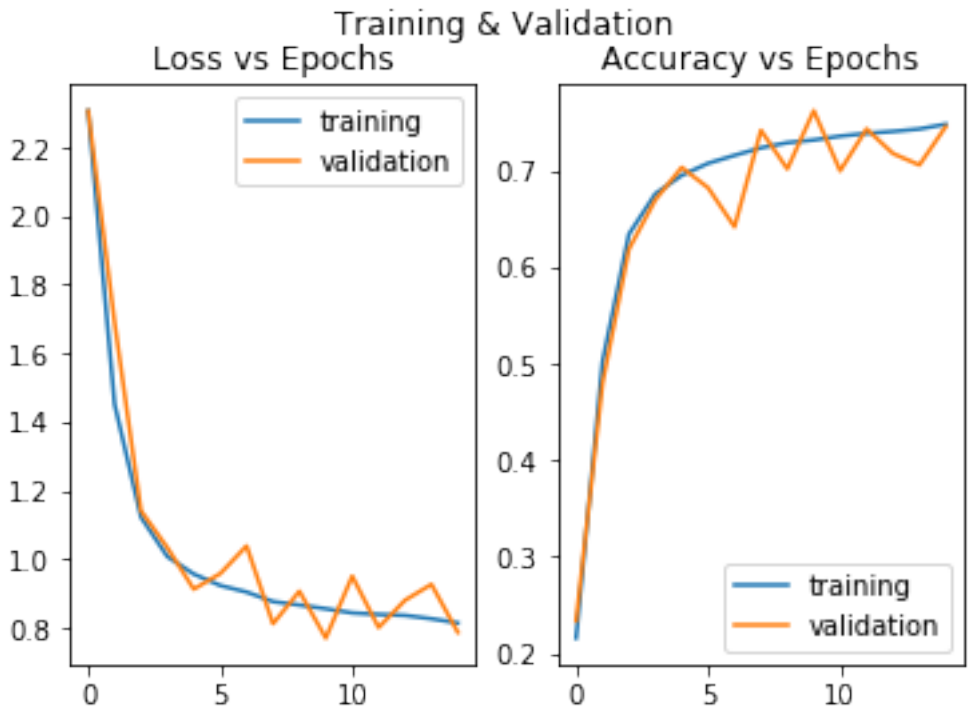
```
_________________________________________________________________
flatten_6 (Flatten)          (None, 16)                0
_________________________________________________________________
dense_24 (Dense)             (None, 128)               2176
_________________________________________________________________
batch_normalization_3 (Batch (None, 128)               512
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_25 (Dense)             (None, 128)               16512
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_26 (Dense)             (None, 10)                1290
=================================================================
Total params: 25,994
Trainable params: 25,386
Non-trainable params: 608
_________________________________________________________________
```

```
In [34]: history_cnn = model_cnn.fit(x_train_greyscale, y_train,
                                     epochs=20, validation_split = 0.2,
                                     callbacks=[model_checkpoint_cnn_callback,
                                                early_stopping_callback])
```

```
Train on 58605 samples, validate on 14652 samples
Epoch 1/20
58605/58605 [==============================] - 132s 2ms/sample - loss: 2.3075 - accuracy: 0.215
Epoch 2/20
58605/58605 [==============================] - 120s 2ms/sample - loss: 1.4514 - accuracy: 0.502
Epoch 3/20
58605/58605 [==============================] - 121s 2ms/sample - loss: 1.1217 - accuracy: 0.634
Epoch 4/20
58605/58605 [==============================] - 122s 2ms/sample - loss: 1.0066 - accuracy: 0.675
Epoch 5/20
58605/58605 [==============================] - 121s 2ms/sample - loss: 0.9556 - accuracy: 0.694
Epoch 6/20
58605/58605 [==============================] - 121s 2ms/sample - loss: 0.9224 - accuracy: 0.707
Epoch 7/20
58605/58605 [==============================] - 129s 2ms/sample - loss: 0.9031 - accuracy: 0.715
Epoch 8/20
58605/58605 [==============================] - 132s 2ms/sample - loss: 0.8753 - accuracy: 0.723
Epoch 9/20
58605/58605 [==============================] - 115s 2ms/sample - loss: 0.8653 - accuracy: 0.728
Epoch 10/20
```

```
58605/58605 [==============================] - 101s 2ms/sample - loss: 0.8545 - accuracy: 0.73
Epoch 11/20
58605/58605 [==============================] - 100s 2ms/sample - loss: 0.8423 - accuracy: 0.735
Epoch 12/20
58605/58605 [==============================] - 101s 2ms/sample - loss: 0.8380 - accuracy: 0.738
Epoch 13/20
58605/58605 [==============================] - 101s 2ms/sample - loss: 0.8350 - accuracy: 0.740
Epoch 14/20
58605/58605 [==============================] - 101s 2ms/sample - loss: 0.8253 - accuracy: 0.742
Epoch 15/20
58605/58605 [==============================] - 107s 2ms/sample - loss: 0.8130 - accuracy: 0.747
```

```
In [50]: fig, ax = plt.subplots(1, 2)
         ax[0].plot(history_cnn.history['loss'], label = 'training')
         ax[0].plot(history_cnn.history['val_loss'], label = 'validation')
         ax[0].legend()
         ax[0].set_title('Loss vs Epochs')

         ax[1].plot(history_cnn.history['accuracy'], label = 'training')
         ax[1].plot(history_cnn.history['val_accuracy'], label = 'validation')
         ax[1].set_title('Accuracy vs Epochs')
         ax[1].legend()
         _=plt.suptitle('Training & Validation')
```



```
In [51]: test_loss, test_acc = model_cnn.evaluate(x_test_greyscale, y_test, verbose=0)
         print(f"Test set loss: {test_loss:.6f}, test set accuracy: {test_acc:.6f}")

Test set loss: 0.796171, test set accuracy: 0.746159
```

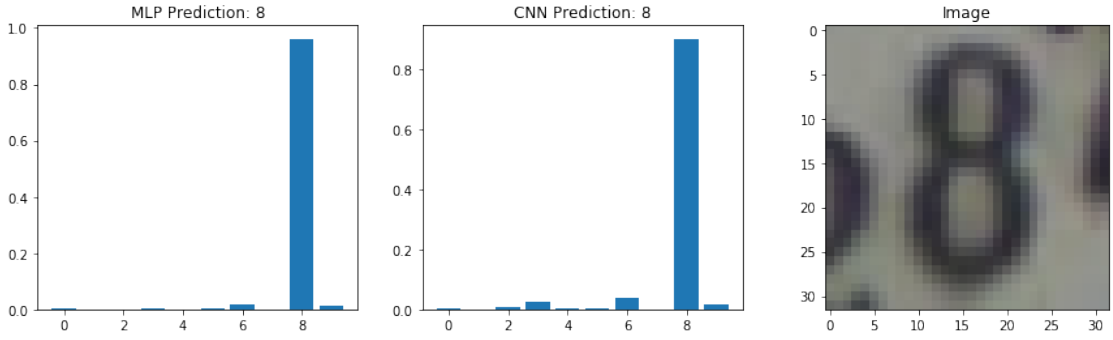
1.5 4. Get model predictions

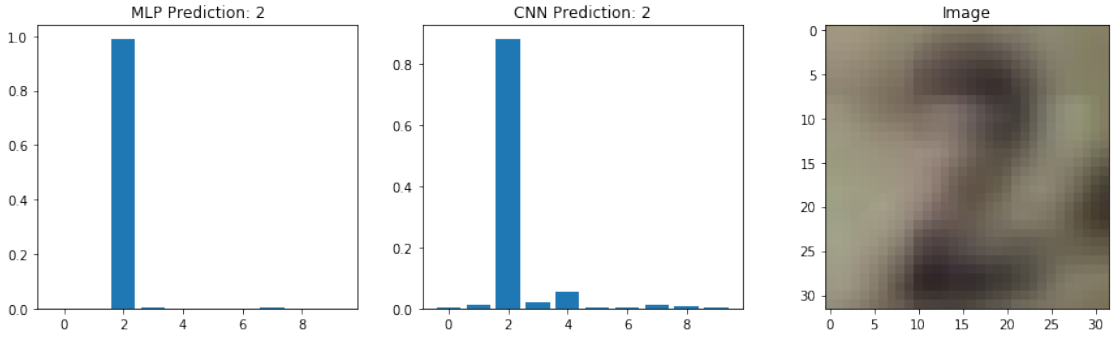
- Load the best weights for the MLP and CNN models that you saved during the training run.
- Randomly select 5 images and corresponding labels from the test set and display the images with their labels.
- Alongside the image and label, show each model's predictive distribution as a bar chart, and the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

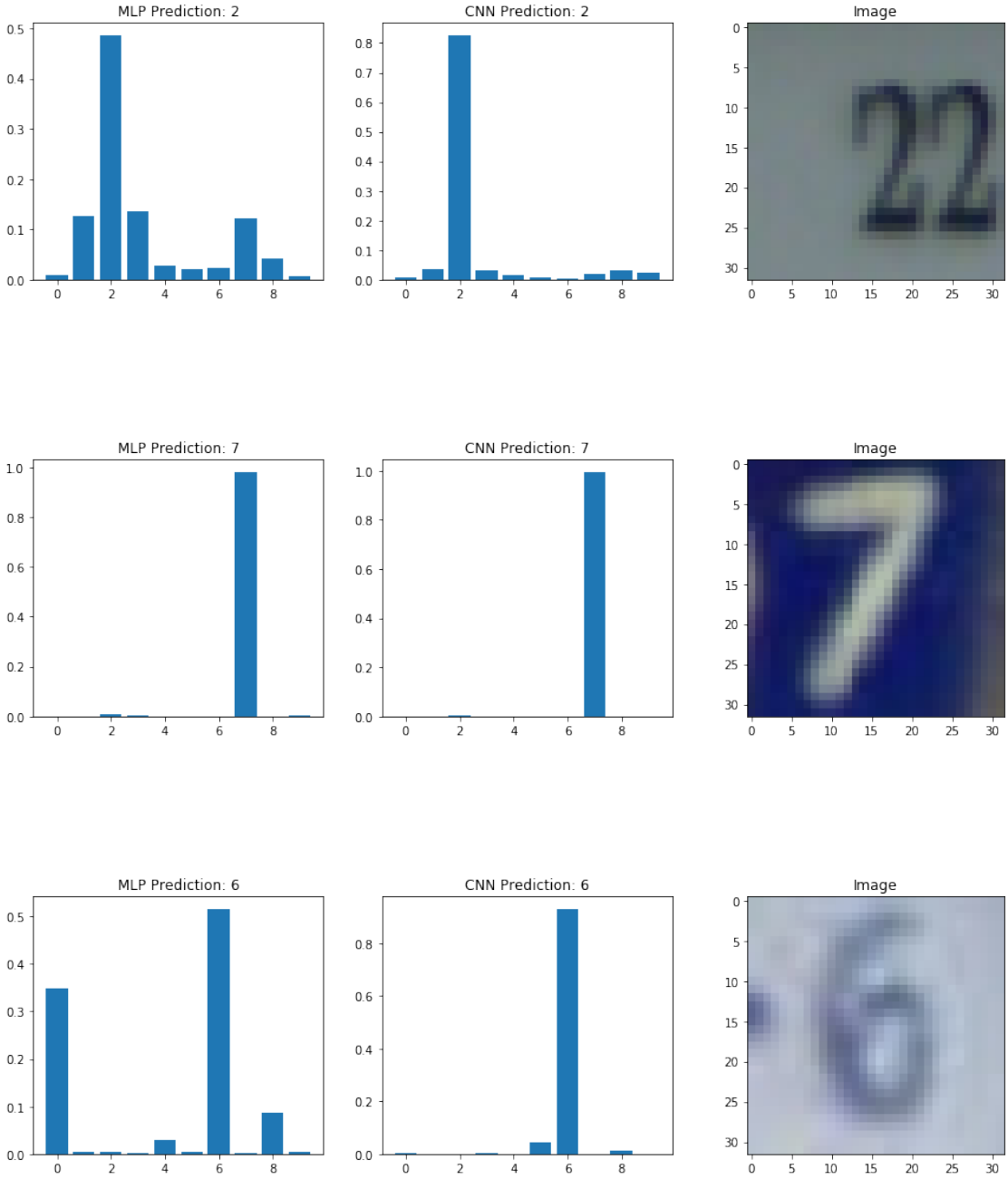
```
In [15]: model_cnn.load_weights('best_weights')
         model_mlp.load_weights('best_weights_mlp')

Out[15]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f0f552f9ef0>

In [10]: random_indices = display_random_images(10, x_test, y_test)  # we'll take the first 5
```


```
In [16]: mlp_preds = model_mlp.predict(x_test_greyscale[random_indices,:,:,:])
         cnn_preds = model_cnn.predict(x_test_greyscale[random_indices,:,:,:])

In [17]: for i in range(5):
             mlp_pred = mlp_preds[i]
             cnn_pred = cnn_preds[i]

             mlp_final_pred = mlp_pred.argmax()+1
```

```
cnn_final_pred = cnn_pred.argmax()+1

image_index = random_indices[i]
fig, ax = plt.subplots(1, 3)
fig.set_figwidth(15)

ax[0].bar(x =[1,2,3,4,5,6,7,8,9,0], height = mlp_pred)
ax[1].bar(x =[1,2,3,4,5,6,7,8,9,0], height = cnn_pred)
ax[2].imshow(x_test[image_index,:,:,:], cmap = None)

ax[0].set_title(f'MLP Prediction: {mlp_final_pred}')
ax[1].set_title(f'CNN Prediction: {cnn_final_pred}')
ax[2].set_title('Image')
```






MLP Prediction: 2
CNN Prediction: 2
Image
MLP Prediction: 7
CNN Prediction: 7
Image
MLP Prediction: 6
CNN Prediction: 6
Image