Capstone Project

August 18, 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.

```
In [1]: import tensorflow as tf
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        from scipy.io import loadmat
```

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 [2]: # 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 [3]: %matplotlib inline
        #Extract the training and testing images and labels separately from the train and test
        #Extracting features and labels from training and testing data
```

```
        xtrain, ytrain = train['X'],train['y']
        xtest, ytest = test['X'], test['y']

        print("Training Set shapes", xtrain.shape, ytrain.shape)
        print("Test Set shapes", xtest.shape, ytest.shape)
```

```
Training Set shapes (32, 32, 3, 73257) (73257, 1)
Test Set shapes (32, 32, 3, 26032) (26032, 1)
```

```
In [4]: xtrain, ytrain = xtrain.transpose((3,0,1,2)), ytrain[:,0]
        xtest, ytest = xtest.transpose((3,0,1,2)), ytest[:,0]

        print("Training Set shapes", xtrain.shape, ytrain.shape)
        print("Test Set shapes", xtest.shape, ytest.shape)
```

```
Training Set shapes (73257, 32, 32, 3) (73257,)
Test Set shapes (26032, 32, 32, 3) (26032,)
```

1.2.1 Ploting training data examples

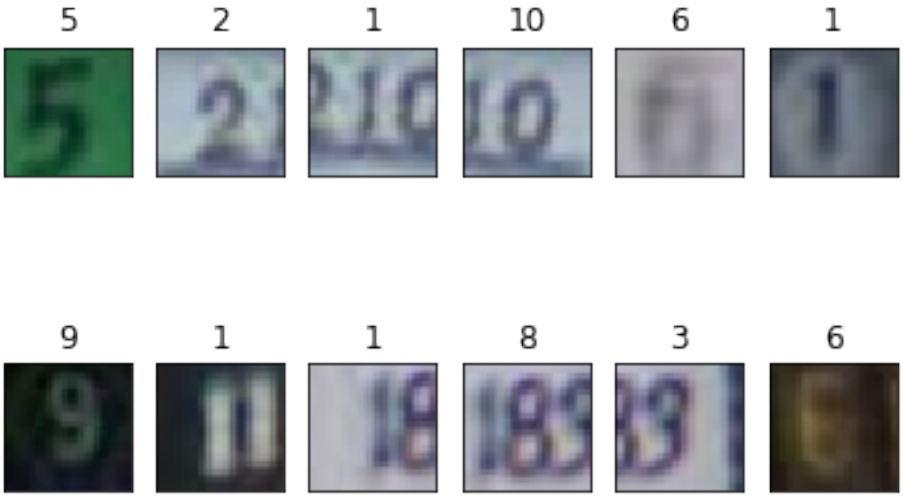
Selecting a sample of images and corresponding labels from the dataset (at least 10), and displaying them in a figure

```
In [5]: def plot_data(images, labels):
            fig, axes = plt.subplots(2, 6)
            for i, ax in enumerate(axes.flat):
                ax.imshow(images[i])
                ax.set_xticks([])
                ax.set_yticks([])
                ax.set_title(labels[i])
```

```
In [6]: plot_data(xtrain, ytrain)
```

1.2.2 Ploting test data examples

```
In [7]: plot_data(xtest, ytest)
```



1.2.3 Converting into greyscale

Converting the training and test images to grayscale by taking the average across all colour channels for each pixel

```
In [5]: def rgb2gray(images):
            return np.expand_dims(np.dot(images, [0.2990, 0.5870, 0.1140]), axis=3)
```

```
In [6]: xtrain = rgb2gray(xtrain).astype(np.float32)
        xtest = rgb2gray(xtest).astype(np.float32)
```

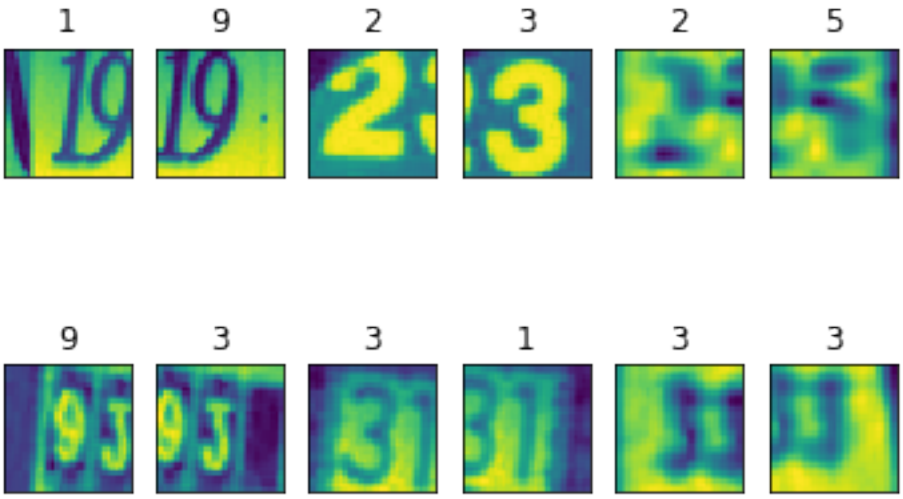
```
In [7]: print("greyscale train shape" , xtrain.shape)
        print("greyscale test shape" , xtest.shape)
```

```
greyscale train shape (73257, 32, 32, 1)
greyscale test shape (26032, 32, 32, 1)
```

```
In [11]: def plot_greydata(images, labels):
             fig, axes = plt.subplots(2, 6)
             for i, ax in enumerate(axes.flat):
                 ax.imshow(images[i,:,:,0])
                 ax.set_xticks([])
                 ax.set_yticks([])
                 ax.set_title(labels[i])
```

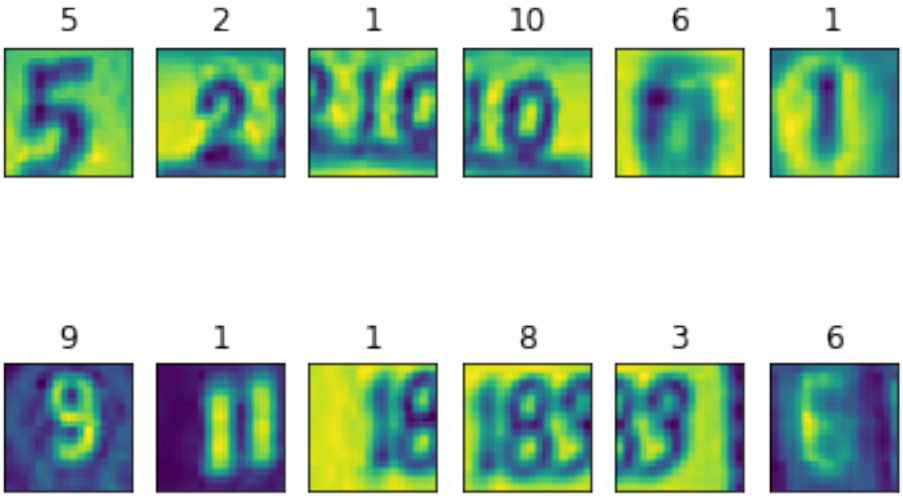
In [12]:
```
print("training data in greyscale")
plot_greydata(xtrain, ytrain)
```

```
training data in greyscale
```

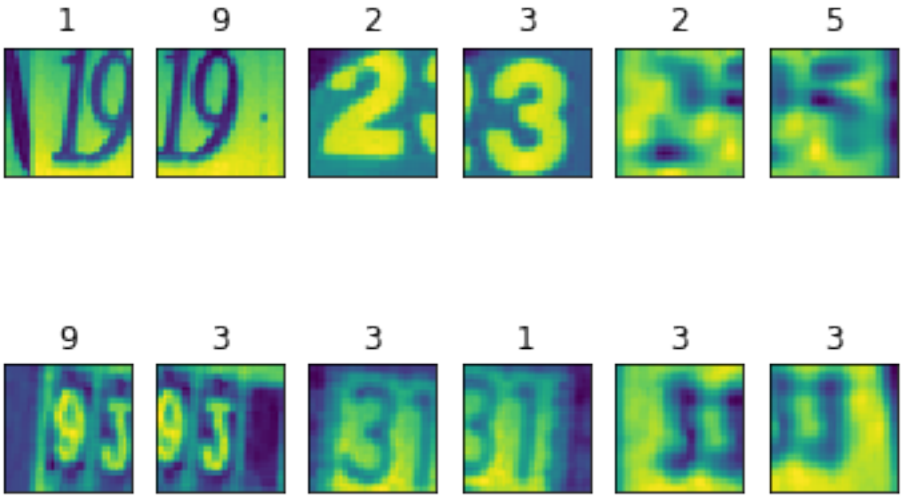


In [13]:
```
print("test data in greyscale")
plot_greydata(xtest, ytest)
```

```
test data in greyscale
```



```
In [8]: #Normalizing Data
        train_mean = np.mean(xtrain, axis=0)
        train_std = np.std(xtrain, axis=0)

        xtrain = (xtrain - train_mean) / train_std
        xtest = (xtest - train_mean)  / train_std

In [9]: xtrain = xtrain/255
        xtest = xtest/255

In [16]: print("training data in greyscale")
         plot_greydata(xtrain, ytrain)

training data in greyscale
```



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 [10]: from tensorflow.keras.models import Sequential
         from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
         from tensorflow.keras.callbacks import ModelCheckpoint
         from tensorflow.keras.layers import Dense, Dropout,BatchNormalization
         from tensorflow.keras import regularizers
```

```
In [11]: cnnxtrain = xtrain
         cnnxtest = xtest
         xtrain = xtrain[..., np.newaxis]
         xtest = xtest[..., np.newaxis]
```

```
In [22]: model = Sequential([
             Flatten(input_shape = xtrain[0].shape),
             Dense(512, activation = 'relu'),
             Dense(256, activation = 'relu'),
             Dense(64, activation = 'relu'),
             Dense(32, activation = 'relu'),
             Dense(11, activation = 'softmax')
         ])
         model.compile(
             optimizer = tf.keras.optimizers.Adam(),
             loss = 'sparse_categorical_crossentropy',
             metrics = ['accuracy'])

         model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_5 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_6 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_7 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_8 (Dense)              (None, 11)                363
=================================================================
Total params: 675,019
Trainable params: 675,019
Non-trainable params: 0
```

```
-----------------------------------------------------------------
```

```
In [20]: def get_checkpoint_every_epoch():
             check = ModelCheckpoint(filepath = 'checkpoints_every_epoch/checkpoint_{epoch:03d}
             return check

         def get_checkpoint_best_only():
             check = ModelCheckpoint(filepath = 'checkpoints_best_only/checkpoint' , monitor="v
             return check
         checkpoint_every_epoch = get_checkpoint_every_epoch()
         checkpoint_best_only = get_checkpoint_best_only()
         callbacks = [checkpoint_every_epoch, checkpoint_best_only]
```

```
In [21]: history= model.fit(xtrain, ytrain, epochs =30,batch_size = 256, validation_split =0.1
```

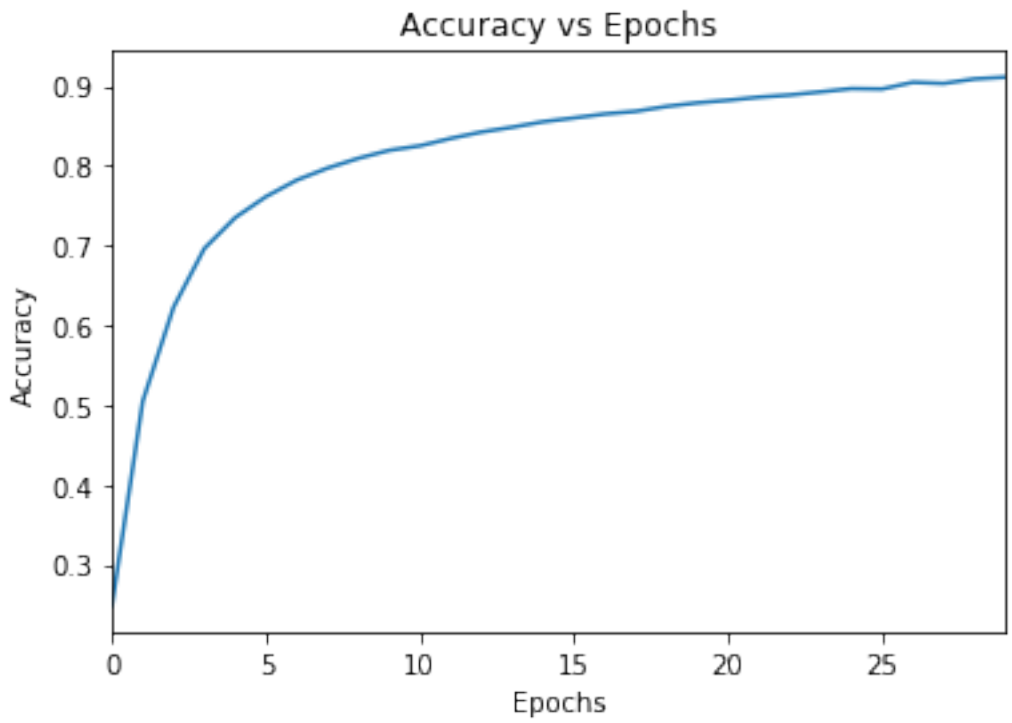
```
Train on 62268 samples, validate on 10989 samples
Epoch 1/30
62268/62268 [==============================] - 25s 408us/sample - loss: 2.0551 - accuracy: 0.24
Epoch 2/30
62268/62268 [==============================] - 23s 364us/sample - loss: 1.4258 - accuracy: 0.50
Epoch 3/30
62268/62268 [==============================] - 23s 365us/sample - loss: 1.1556 - accuracy: 0.62
Epoch 4/30
62268/62268 [==============================] - 23s 368us/sample - loss: 0.9655 - accuracy: 0.69
Epoch 5/30
62268/62268 [==============================] - 23s 363us/sample - loss: 0.8534 - accuracy: 0.73
Epoch 6/30
62268/62268 [==============================] - 23s 362us/sample - loss: 0.7762 - accuracy: 0.76
Epoch 7/30
62268/62268 [==============================] - 23s 361us/sample - loss: 0.7143 - accuracy: 0.78
Epoch 8/30
62268/62268 [==============================] - 22s 359us/sample - loss: 0.6670 - accuracy: 0.79
Epoch 9/30
62268/62268 [==============================] - 22s 348us/sample - loss: 0.6243 - accuracy: 0.80
Epoch 10/30
62268/62268 [==============================] - 22s 348us/sample - loss: 0.5918 - accuracy: 0.81
Epoch 11/30
62268/62268 [==============================] - 22s 348us/sample - loss: 0.5700 - accuracy: 0.82
Epoch 12/30
62268/62268 [==============================] - 22s 353us/sample - loss: 0.5404 - accuracy: 0.83
Epoch 13/30
62268/62268 [==============================] - 21s 345us/sample - loss: 0.5164 - accuracy: 0.84
Epoch 14/30
62268/62268 [==============================] - 21s 343us/sample - loss: 0.4938 - accuracy: 0.84
Epoch 15/30
62268/62268 [==============================] - 21s 343us/sample - loss: 0.4732 - accuracy: 0.85
Epoch 16/30
```

```
62268/62268 [==============================] - 21s 340us/sample - loss: 0.4580 - accuracy: 0.85
Epoch 17/30
62268/62268 [==============================] - 21s 337us/sample - loss: 0.4396 - accuracy: 0.86
Epoch 18/30
62268/62268 [==============================] - 21s 339us/sample - loss: 0.4290 - accuracy: 0.86
Epoch 19/30
62268/62268 [==============================] - 21s 334us/sample - loss: 0.4113 - accuracy: 0.87
Epoch 20/30
62268/62268 [==============================] - 21s 331us/sample - loss: 0.3974 - accuracy: 0.87
Epoch 21/30
62268/62268 [==============================] - 21s 339us/sample - loss: 0.3843 - accuracy: 0.88
Epoch 22/30
62268/62268 [==============================] - 21s 339us/sample - loss: 0.3705 - accuracy: 0.88
Epoch 23/30
62268/62268 [==============================] - 22s 346us/sample - loss: 0.3618 - accuracy: 0.88
Epoch 24/30
62268/62268 [==============================] - 21s 334us/sample - loss: 0.3524 - accuracy: 0.89
Epoch 25/30
62268/62268 [==============================] - 21s 332us/sample - loss: 0.3363 - accuracy: 0.89
Epoch 26/30
62268/62268 [==============================] - 20s 326us/sample - loss: 0.3354 - accuracy: 0.89
Epoch 27/30
62268/62268 [==============================] - 21s 331us/sample - loss: 0.3134 - accuracy: 0.90
Epoch 28/30
62268/62268 [==============================] - 21s 331us/sample - loss: 0.3161 - accuracy: 0.90
Epoch 29/30
62268/62268 [==============================] - 21s 336us/sample - loss: 0.3012 - accuracy: 0.90
Epoch 30/30
62268/62268 [==============================] - 21s 339us/sample - loss: 0.2915 - accuracy: 0.91
```

```
In [22]: frame = pd.DataFrame(history.history)
```

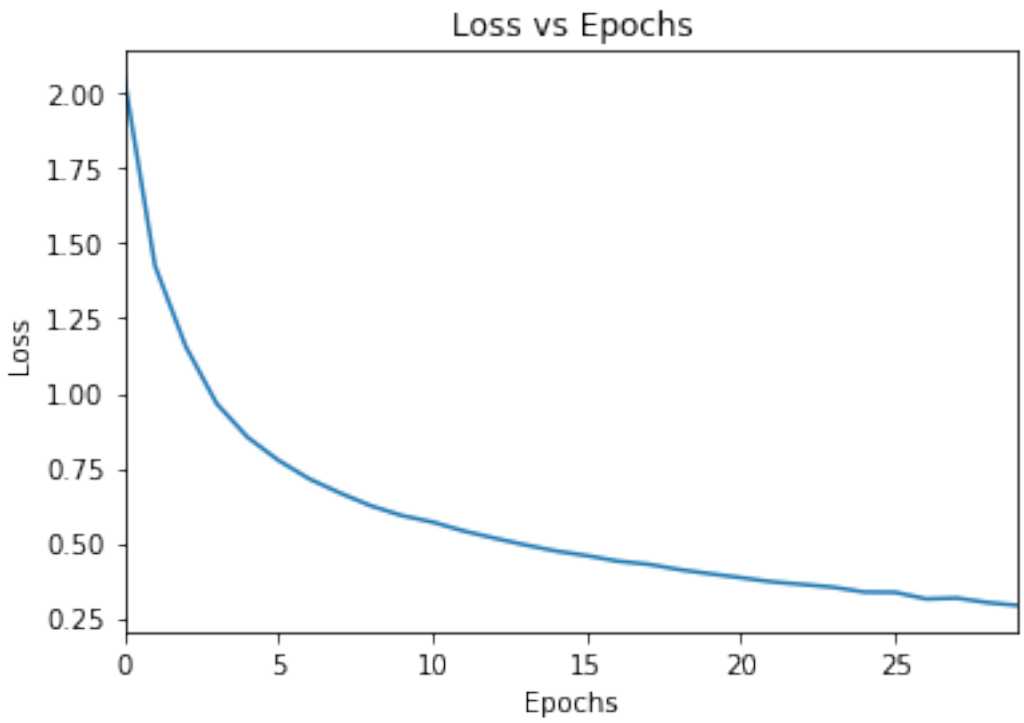
```
In [23]: acc_plot = frame.plot(y="accuracy", title="Accuracy vs Epochs", legend=False)
         acc_plot.set(xlabel="Epochs", ylabel="Accuracy")
```

```
Out[23]: [Text(0, 0.5, 'Accuracy'), Text(0.5, 0, 'Epochs')]
```




```
In [24]: acc_plot = frame.plot(y="loss", title = "Loss vs Epochs",legend=False)
         acc_plot.set(xlabel="Epochs", ylabel="Loss")
```

```
Out[24]: [Text(0, 0.5, 'Loss'), Text(0.5, 0, 'Epochs')]
```




```
In [25]: def evaluate_model(model, scaled_test_images, test_labels):
             test_loss, test_accuracy = model.evaluate(scaled_test_images,test_labels, verbose
             return (test_loss, test_accuracy)

         test_loss, test_accuracy = evaluate_model(model, xtest, ytest)
         print(f"Test loss: {test_loss}")
         print(f"Test accuracy: {test_accuracy}")
```

```
26032/1 - 6s - loss: 0.4905 - accuracy: 0.8178
Test loss: 0.6738043304374779
Test accuracy: 0.8178011775016785
```

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 [12]: cnnmodel = Sequential([
             Conv2D(filters=32, kernel_size=3, input_shape=cnnxtrain[0].shape, activation='relu
             Conv2D(filters=16, kernel_size=3, activation='relu'),
             Conv2D(filters=8, kernel_size=3, activation='relu'),
             MaxPooling2D(pool_size=8),
             Flatten(name='flatten'),
             Dense(64, activation ='relu'),
             BatchNormalization(),
             Dense(32, activation='relu'),
             Dropout(0.3),
             Dense(20, activation='relu'),
             Dense(11, activation='softmax')
         ])
         cnnmodel.compile(
             optimizer = tf.keras.optimizers.Adam(),
             loss = 'sparse_categorical_crossentropy',
             metrics = ['accuracy'])

         print(cnnmodel.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 16)        4624
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 26, 26, 8)         1160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 3, 3, 8)           0
_________________________________________________________________
flatten (Flatten)            (None, 72)                0
_________________________________________________________________
dense (Dense)                (None, 64)                4672
_________________________________________________________________
batch_normalization (BatchNo (None, 64)                256
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
```

```
_________________________________________________________________
dense_2 (Dense)              (None, 20)                660
_________________________________________________________________
dense_3 (Dense)              (None, 11)                231
=================================================================
Total params: 14,003
Trainable params: 13,875
Non-trainable params: 128
_________________________________________________________________
None
```

```
In [13]: def get_early_stopping():
             early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience
             return early_stopping
         def get_checkpoint_best_only():
             check = ModelCheckpoint(filepath = 'cnncheckpoints_best_only/checkpoint' , monito
             return check
```

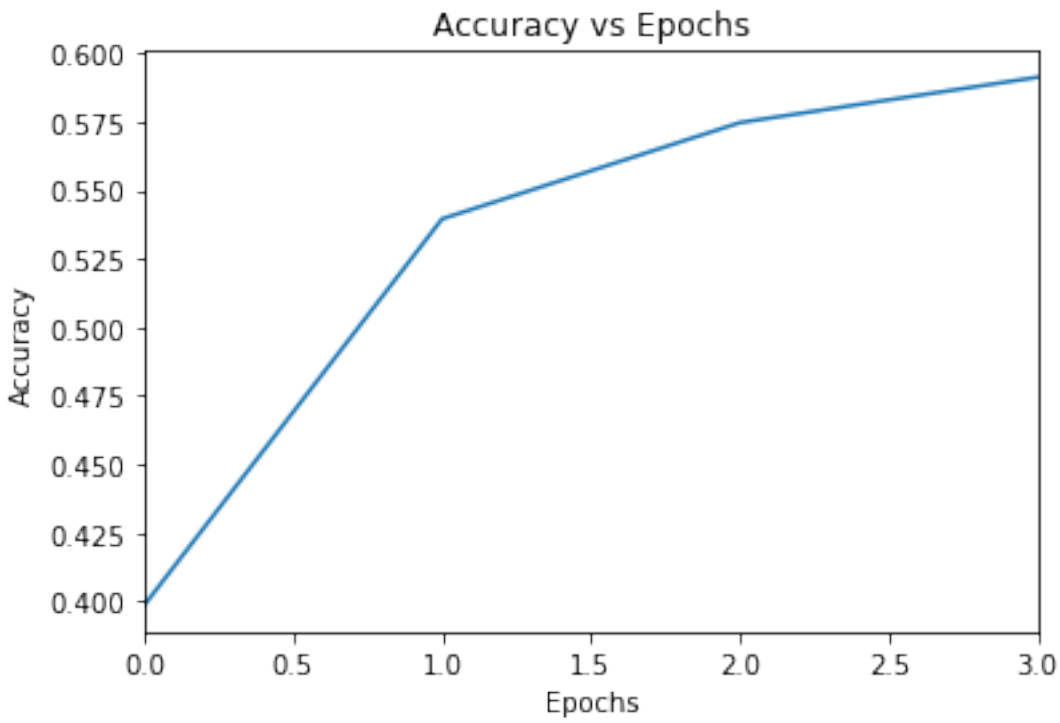
```
In [14]: checkpoint_best_only = get_checkpoint_best_only()
         early_stopping = get_early_stopping()

         callbacks = [checkpoint_best_only, early_stopping]
         history= cnnmodel.fit(cnnxtrain, ytrain, epochs =20,batch_size = 32, validation_split
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/20
62268/62268 [==============================] - 624s 10ms/sample - loss: 1.7306 - accuracy: 0.3
Epoch 2/20
62268/62268 [==============================] - 635s 10ms/sample - loss: 1.3701 - accuracy: 0.5
Epoch 3/20
62268/62268 [==============================] - 615s 10ms/sample - loss: 1.2733 - accuracy: 0.5
Epoch 4/20
62268/62268 [==============================] - 602s 10ms/sample - loss: 1.2270 - accuracy: 0.5
```

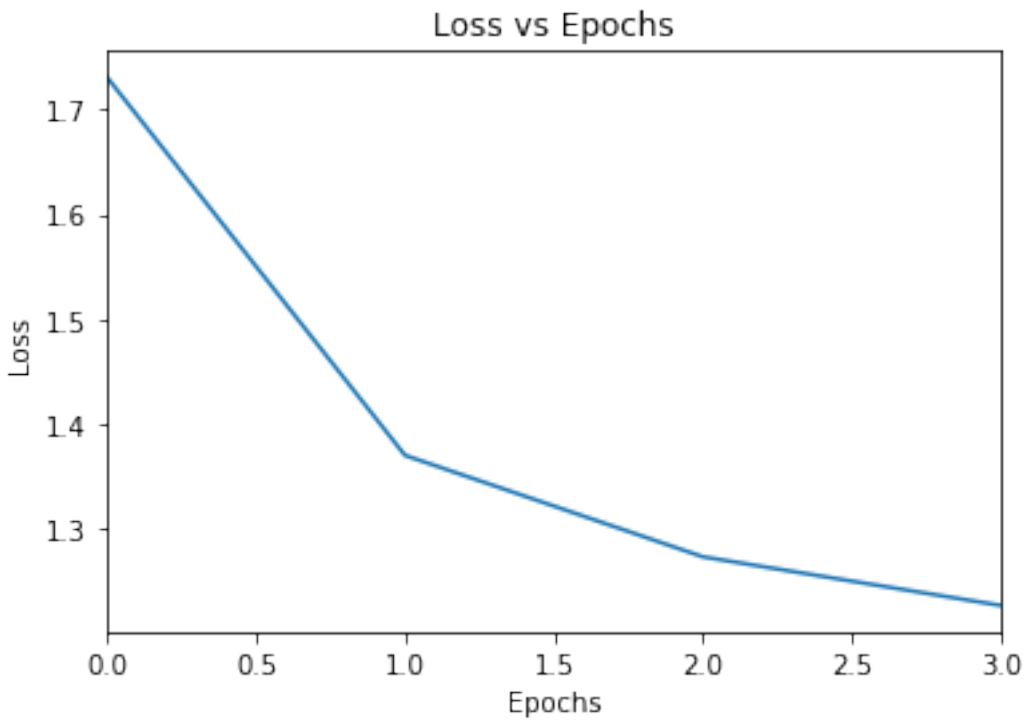
```
In [15]: frame = pd.DataFrame(history.history)
```

```
In [16]: acc_plot = frame.plot(y="accuracy", title="Accuracy vs Epochs", legend=False)
         acc_plot.set(xlabel="Epochs", ylabel="Accuracy")
```

```
Out[16]: [Text(0, 0.5, 'Accuracy'), Text(0.5, 0, 'Epochs')]
```




```
In [17]: acc_plot = frame.plot(y="loss", title = "Loss vs Epochs",legend=False)
         acc_plot.set(xlabel="Epochs", ylabel="Loss")

Out[17]: [Text(0, 0.5, 'Loss'), Text(0.5, 0, 'Epochs')]
```




```
In [18]: def evaluate_model(model, scaled_test_images, test_labels):
             test_loss, test_accuracy = model.evaluate(scaled_test_images,test_labels, verbose
             return (test_loss, test_accuracy)

         test_loss, test_accuracy = evaluate_model(cnnmodel, cnnxtest, ytest)
         print(f"Test loss: {test_loss}")
         print(f"Test accuracy: {test_accuracy}")
```

```
26032/1 - 59s - loss: 3.1707 - accuracy: 0.2270
Test loss: 3.6282062634560104
Test accuracy: 0.22698986530303955
```

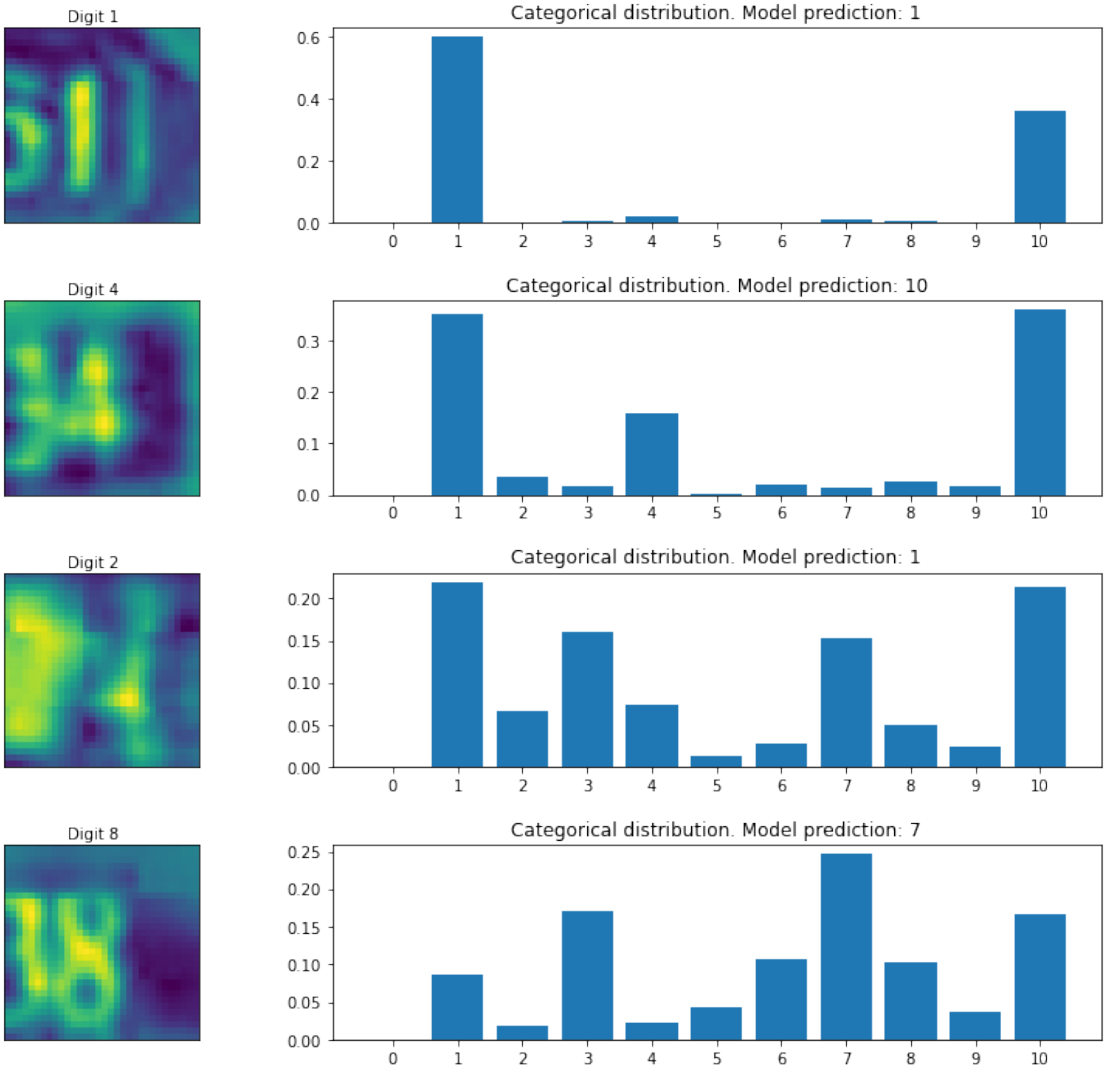
1.5 4. Get model predictions

- 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.

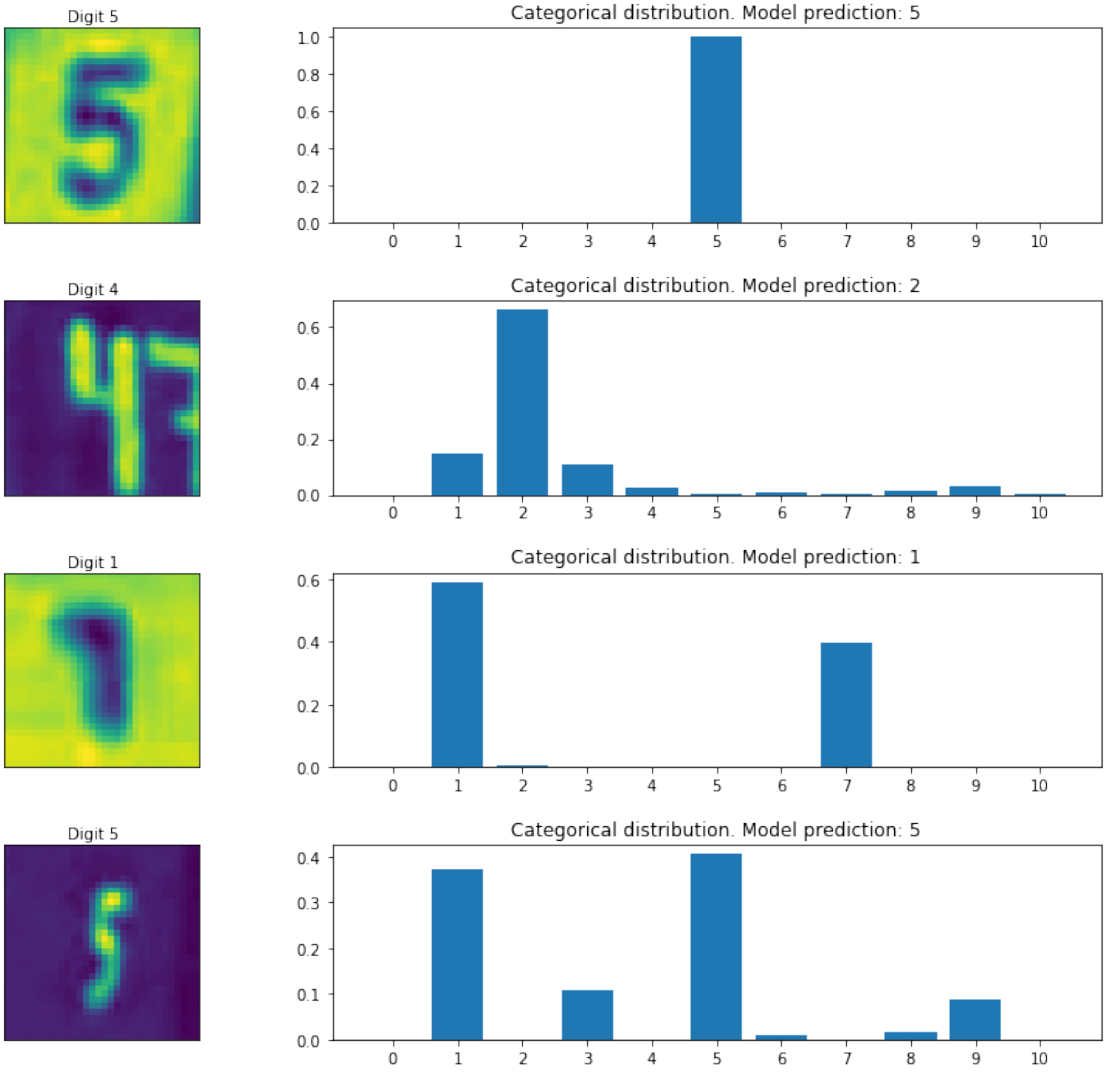
```
In [19]: def plotPrediction(model, data, label):
             num_test_images = data.shape[0]
             random_inx = np.random.choice(num_test_images, 4)
```

```
        random_test_images = data[random_inx, ...]
        random_test_labels = label[random_inx, ...]

        predictions = model.predict(random_test_images)

        fig, axes = plt.subplots(4, 2, figsize=(16, 12))
        fig.subplots_adjust(hspace=0.4, wspace=-0.2)

        for i, (prediction, image, label) in enumerate(zip(predictions, random_test_images
            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)), prediction)
            axes[i, 1].set_xticks(np.arange(len(prediction)))
            axes[i, 1].set_title(f"Categorical distribution. Model prediction: {np.argmax

        plt.show()
    def get_model_best_epoch(model, checkpoint_dir):
        model.load_weights(tf.train.latest_checkpoint(checkpoint_dir, latest_filename=None
        return model
```

In [20]:
```
cnnmodel= get_model_best_epoch(cnnmodel, 'cnncheckpoints_best_only')
plotPrediction(cnnmodel, cnnxtest, ytest)
```




```
In [23]: model= get_model_best_epoch(model, 'checkpoints_best_only')
         plotPrediction(model, xtest, ytest)
```


Digit 5
Categorical distribution. Model prediction: 5
Digit 4
Categorical distribution. Model prediction: 2
Digit 1
Categorical distribution. Model prediction: 1
Digit 5
Categorical distribution. Model prediction: 5

In [ ]:

In [ ]:

In [ ]: