Capstone Project

July 31, 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 [18]: import tensorflow as tf
         from scipy.io import loadmat
         import matplotlib.pyplot as plt
         import numpy as np
         import os
         import random

         from scipy.io import loadmat
         from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
         from tensorflow.keras.models import Sequential
         from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, BatchNormali
```

 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

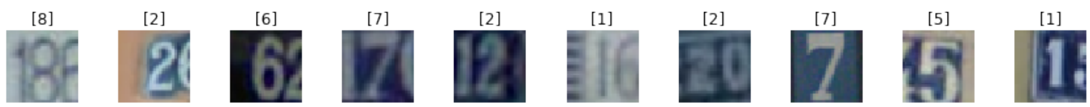
- Extract the training and testing images and labels separately from the train and test dictionaries loaded for you.
- Select a random sample of images and corresponding labels from the dataset (at least 10), and display them in a figure.
- Convert the training and test images to grayscale by taking 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]: X_train = train['X']
        X_test = test['X']
        y_train = train['y']
        y_test = test['y']
```

```
In [4]: X_train = np.moveaxis(X_train, -1, 0)
        X_test = np.moveaxis(X_test, -1 , 0)
```

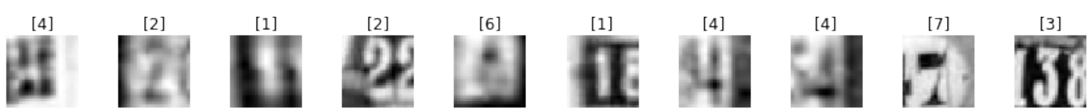
```
In [5]: y_train = np.where(y_train==10, 0, y_train)
        y_test = np.where(y_test==10, 0, y_test)
```

```
In [6]: indices = random.sample( range(0,X_train.shape[0]), 10 )
        fig, ax = plt.subplots(1, 10, figsize=(15,1))

        for i in range(10):
            ax[i].set_axis_off()
            ax[i].imshow(X_train[indices[i]])
            ax[i].set_title(y_train[indices[i]])
```




```
In [7]: X_train_greyscale = np.mean(X_train, -1, keepdims=True)
        X_test_greyscale = np.mean(X_test, -1, keepdims=True)
```

```
In [8]: indices = random.sample( range(0,X_train_greyscale.shape[0]), 10 )
        fig, ax = plt.subplots(1, 10, figsize=(15,1))

        for i in range(10):
            ax[i].set_axis_off()
            ax[i].imshow(X_train_greyscale[indices[i],:,:,0], cmap='gray')
            ax[i].set_title(y_train[indices[i]])
```



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 [9]: def get_MLP_model():

            Model = Sequential([
                Flatten(input_shape=X_train[0].shape),
                Dense(1024, activation='relu'),
                Dense(512, activation='relu'),
                Dense(256, activation='relu'),
                Dense(128, activation='relu'),
                Dense(10, activation='softmax')
            ])

            return(Model)

        model = get_MLP_model()
        model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              3146752
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_3 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
Total params: 3,837,066
Trainable params: 3,837,066
Non-trainable params: 0
_________________________________________________________________
```

```
In [10]: checkpoint_path = 'Checkpoint/best_model'
         checkpoint = ModelCheckpoint(checkpoint_path,
```

```
                             save_best_only=True,
                             save_weights_only=True,
                             verbose=2,
                             save_freq='epoch',
                             monitor='val_accuracy',
                             mode='max')
```

In [11]:
```
EarlyStop = EarlyStopping(monitor='val_accuracy', mode='max', patience=3)
```

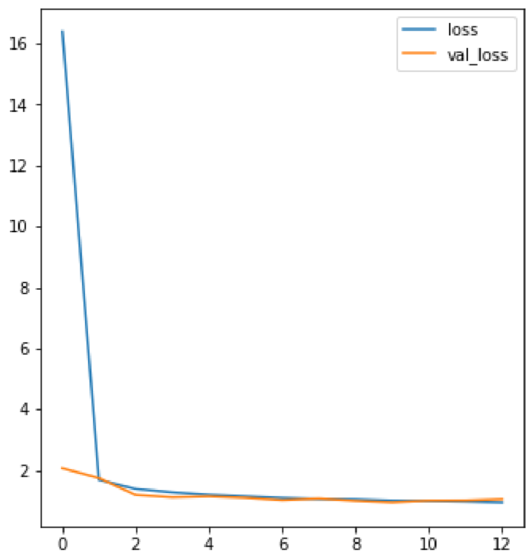
In [12]:
```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accu
```

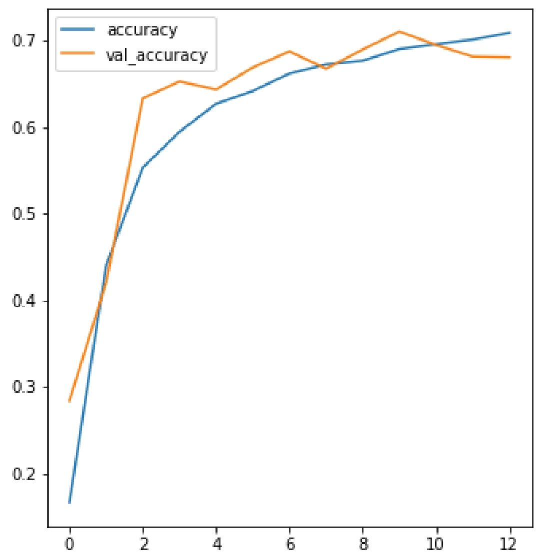
In [13]:
```
history = model.fit(X_train, y_train, epochs=30,
                    batch_size=64, verbose=1,
                    validation_split=0.15,
                    callbacks=[checkpoint, EarlyStop])
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/30
62208/62268 [============================>.] - ETA: 0s - loss: 16.4113 - accuracy: 0.1665
Epoch 00001: val_accuracy improved from -inf to 0.28410, saving model to Checkpoint/best_model
62268/62268 [==============================] - 170s 3ms/sample - loss: 16.3976 - accuracy: 0.16
Epoch 2/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.6834 - accuracy: 0.4403
Epoch 00002: val_accuracy improved from 0.28410 to 0.42051, saving model to Checkpoint/best_mod
62268/62268 [==============================] - 168s 3ms/sample - loss: 1.6834 - accuracy: 0.440
Epoch 3/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.3903 - accuracy: 0.5532
Epoch 00003: val_accuracy improved from 0.42051 to 0.63336, saving model to Checkpoint/best_mod
62268/62268 [==============================] - 167s 3ms/sample - loss: 1.3900 - accuracy: 0.553
Epoch 4/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.2773 - accuracy: 0.5946
Epoch 00004: val_accuracy improved from 0.63336 to 0.65274, saving model to Checkpoint/best_mod
62268/62268 [==============================] - 168s 3ms/sample - loss: 1.2771 - accuracy: 0.594
Epoch 5/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.1976 - accuracy: 0.6271
Epoch 00005: val_accuracy did not improve from 0.65274
62268/62268 [==============================] - 168s 3ms/sample - loss: 1.1978 - accuracy: 0.627
Epoch 6/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.1466 - accuracy: 0.6418
Epoch 00006: val_accuracy improved from 0.65274 to 0.66894, saving model to Checkpoint/best_mod
62268/62268 [==============================] - 168s 3ms/sample - loss: 1.1467 - accuracy: 0.641
Epoch 7/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.0930 - accuracy: 0.6620
Epoch 00007: val_accuracy improved from 0.66894 to 0.68751, saving model to Checkpoint/best_mod
62268/62268 [==============================] - 167s 3ms/sample - loss: 1.0931 - accuracy: 0.662
Epoch 8/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.0566 - accuracy: 0.6725
Epoch 00008: val_accuracy did not improve from 0.68751
62268/62268 [==============================] - 167s 3ms/sample - loss: 1.0570 - accuracy: 0.672
```

```
Epoch 9/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.0457 - accuracy: 0.6767
Epoch 00009: val_accuracy improved from 0.68751 to 0.68969, saving model to Checkpoint/best_mod
62268/62268 [==============================] - 167s 3ms/sample - loss: 1.0456 - accuracy: 0.676
Epoch 10/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.0043 - accuracy: 0.6902
Epoch 00010: val_accuracy improved from 0.68969 to 0.71053, saving model to Checkpoint/best_mod
62268/62268 [==============================] - 168s 3ms/sample - loss: 1.0042 - accuracy: 0.690
Epoch 11/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.9911 - accuracy: 0.6957
Epoch 00011: val_accuracy did not improve from 0.71053
62268/62268 [==============================] - 167s 3ms/sample - loss: 0.9908 - accuracy: 0.695
Epoch 12/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.9711 - accuracy: 0.7012
Epoch 00012: val_accuracy did not improve from 0.71053
62268/62268 [==============================] - 166s 3ms/sample - loss: 0.9710 - accuracy: 0.701
Epoch 13/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.9430 - accuracy: 0.7091
Epoch 00013: val_accuracy did not improve from 0.71053
62268/62268 [==============================] - 165s 3ms/sample - loss: 0.9431 - accuracy: 0.709
```

In [14]:
```python
def plot_metrics(h):
    fig, axs = plt.subplots(1,2, figsize=(12, 6))
    axs[0].plot(h['loss'], label='loss')
    axs[0].plot(h['val_loss'], label='val_loss')
    axs[0].legend()

    axs[1].plot(h['accuracy'], label='accuracy')
    axs[1].plot(h['val_accuracy'], label='val_accuracy')
    axs[1].legend()

plot_metrics(history.history)
```





In [15]:
```
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=False)
print(f"Test Loss is {test_loss}")
print(f"Test Accuracy is {test_accuracy}")
```

```
Test Loss is 1.157867325729729
Test Accuracy is 0.659342348575592
```

In [ ]:

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 [19]: def get_CNN_model(input_shape):

             Model = Sequential([
                 Conv2D(16, (3,3), padding='same', activation='relu', input_shape=input_shape)
                 BatchNormalization(),
                 MaxPooling2D((2,2)),
                 Conv2D(16, (3,3), padding='same', activation='relu'),
                 BatchNormalization(),
                 MaxPooling2D((2,2)),
                 Conv2D(16, (3,3), padding='same', activation='relu'),
                 BatchNormalization(),
                 MaxPooling2D((2,2)),
                 Flatten(),
                 Dense(256, activation='relu'),
                 Dropout(0.2),
                 Dense(128, activation='relu'),
                 Dropout(0.2),
                 Dense(10, activation='softmax')
             ])

             return(Model)

         CNN_model = get_CNN_model(X_train[0].shape)
         CNN_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 16)        448
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 16)        64
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 16)        2320
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 16)        64
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 16)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 16)          2320
_________________________________________________________________
batch_normalization_3 (Batch (None, 8, 8, 16)          64
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 16)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
```

```
_________________________________________________________________
dense_5 (Dense)              (None, 256)               65792
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 105,258
Trainable params: 105,162
Non-trainable params: 96
_________________________________________________________________
```

```
In [20]: checkpoint_path_CNN = 'Checkpoint/best_model_CNN'
         checkpoint_CNN = ModelCheckpoint(checkpoint_path_CNN,
                                          save_best_only=True,
                                          save_weights_only=True,
                                          verbose=2,
                                          save_freq='epoch',
                                          monitor='val_accuracy',
                                          mode='max')
```

```
In [21]: EarlyStop_CNN = EarlyStopping(monitor='val_accuracy', mode='max', patience=3)
```

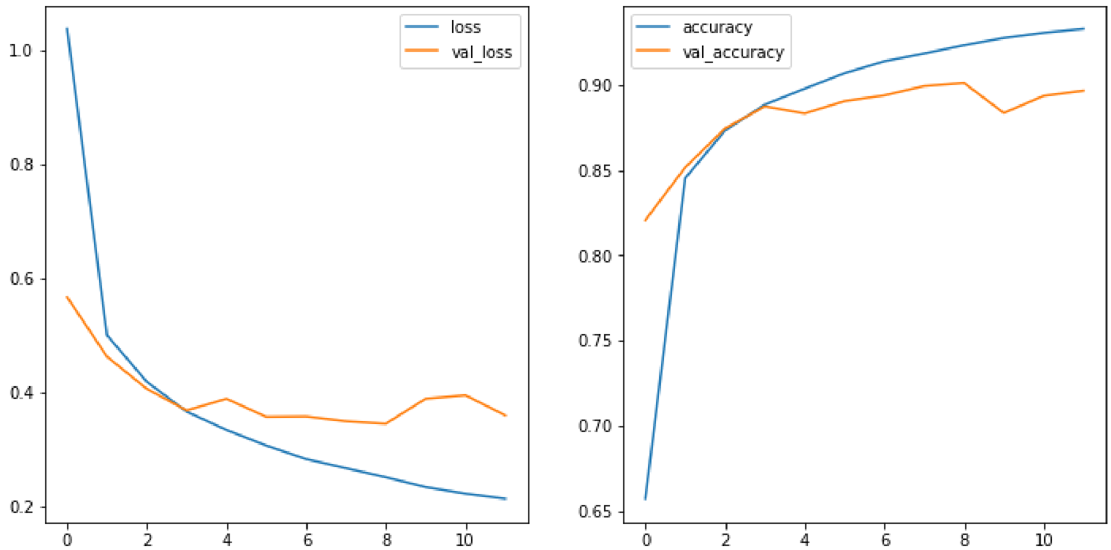
```
In [22]: CNN_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=[
```

```
In [23]: history_CNN = CNN_model.fit(X_train, y_train, epochs=30,
                                     batch_size=64, verbose=1,
                                     validation_split=0.15,
                                     callbacks=[checkpoint_CNN, EarlyStop_CNN])
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/30
62208/62268 [============================>.] - ETA: 0s - loss: 1.0386 - accuracy: 0.6570
Epoch 00001: val_accuracy improved from -inf to 0.82073, saving model to Checkpoint/best_model_
62268/62268 [==============================] - 334s 5ms/sample - loss: 1.0381 - accuracy: 0.657
Epoch 2/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.4998 - accuracy: 0.8454
Epoch 00002: val_accuracy improved from 0.82073 to 0.85167, saving model to Checkpoint/best_moc
62268/62268 [==============================] - 332s 5ms/sample - loss: 0.4998 - accuracy: 0.845
Epoch 3/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.4180 - accuracy: 0.8733
Epoch 00003: val_accuracy improved from 0.85167 to 0.87460, saving model to Checkpoint/best_moc
62268/62268 [==============================] - 329s 5ms/sample - loss: 0.4180 - accuracy: 0.873
Epoch 4/30
```

```
62208/62268 [============================>.] - ETA: 0s - loss: 0.3655 - accuracy: 0.8886
Epoch 00004: val_accuracy improved from 0.87460 to 0.88752, saving model to Checkpoint/best_moc
62268/62268 [==============================] - 331s 5ms/sample - loss: 0.3655 - accuracy: 0.888
Epoch 5/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.3333 - accuracy: 0.8979
Epoch 00005: val_accuracy did not improve from 0.88752
62268/62268 [==============================] - 331s 5ms/sample - loss: 0.3333 - accuracy: 0.897
Epoch 6/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.3059 - accuracy: 0.9071
Epoch 00006: val_accuracy improved from 0.88752 to 0.89062, saving model to Checkpoint/best_moc
62268/62268 [==============================] - 343s 6ms/sample - loss: 0.3059 - accuracy: 0.907
Epoch 7/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.2823 - accuracy: 0.9140
Epoch 00007: val_accuracy improved from 0.89062 to 0.89398, saving model to Checkpoint/best_moc
62268/62268 [==============================] - 367s 6ms/sample - loss: 0.2822 - accuracy: 0.914
Epoch 8/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.2663 - accuracy: 0.9185
Epoch 00008: val_accuracy improved from 0.89398 to 0.89944, saving model to Checkpoint/best_moc
62268/62268 [==============================] - 366s 6ms/sample - loss: 0.2662 - accuracy: 0.918
Epoch 9/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.2503 - accuracy: 0.9235
Epoch 00009: val_accuracy improved from 0.89944 to 0.90126, saving model to Checkpoint/best_moc
62268/62268 [==============================] - 366s 6ms/sample - loss: 0.2503 - accuracy: 0.923
Epoch 10/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.2332 - accuracy: 0.9277
Epoch 00010: val_accuracy did not improve from 0.90126
62268/62268 [==============================] - 366s 6ms/sample - loss: 0.2331 - accuracy: 0.927
Epoch 11/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.2215 - accuracy: 0.9307
Epoch 00011: val_accuracy did not improve from 0.90126
62268/62268 [==============================] - 336s 5ms/sample - loss: 0.2214 - accuracy: 0.930
Epoch 12/30
62208/62268 [============================>.] - ETA: 0s - loss: 0.2126 - accuracy: 0.9330
Epoch 00012: val_accuracy did not improve from 0.90126
62268/62268 [==============================] - 332s 5ms/sample - loss: 0.2126 - accuracy: 0.933
```

In [26]:
```
def plot_metrics(h):
    fig, axs = plt.subplots(1,2, figsize=(12, 6))
    axs[0].plot(h['loss'], label='loss')
    axs[0].plot(h['val_loss'], label='val_loss')
    axs[0].legend()

    axs[1].plot(h['accuracy'], label='accuracy')
    axs[1].plot(h['val_accuracy'], label='val_accuracy')
    axs[1].legend()

plot_metrics(history_CNN.history)
```




```
In [25]: test_loss_CNN, test_accuracy_CNN = CNN_model.evaluate(X_test, y_test, verbose=False)
         print(f"Test Loss is {test_loss_CNN}")
         print(f"Test Accuracy is {test_accuracy_CNN}")
```

```
Test Loss is 0.3870960692803535
Test Accuracy is 0.8946681022644043
```

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 [33]: Best_MLP = get_MLP_model()
         Best_MLP.load_weights('Checkpoint/best_model')
```

```
Out[33]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f39b61b3828>
```

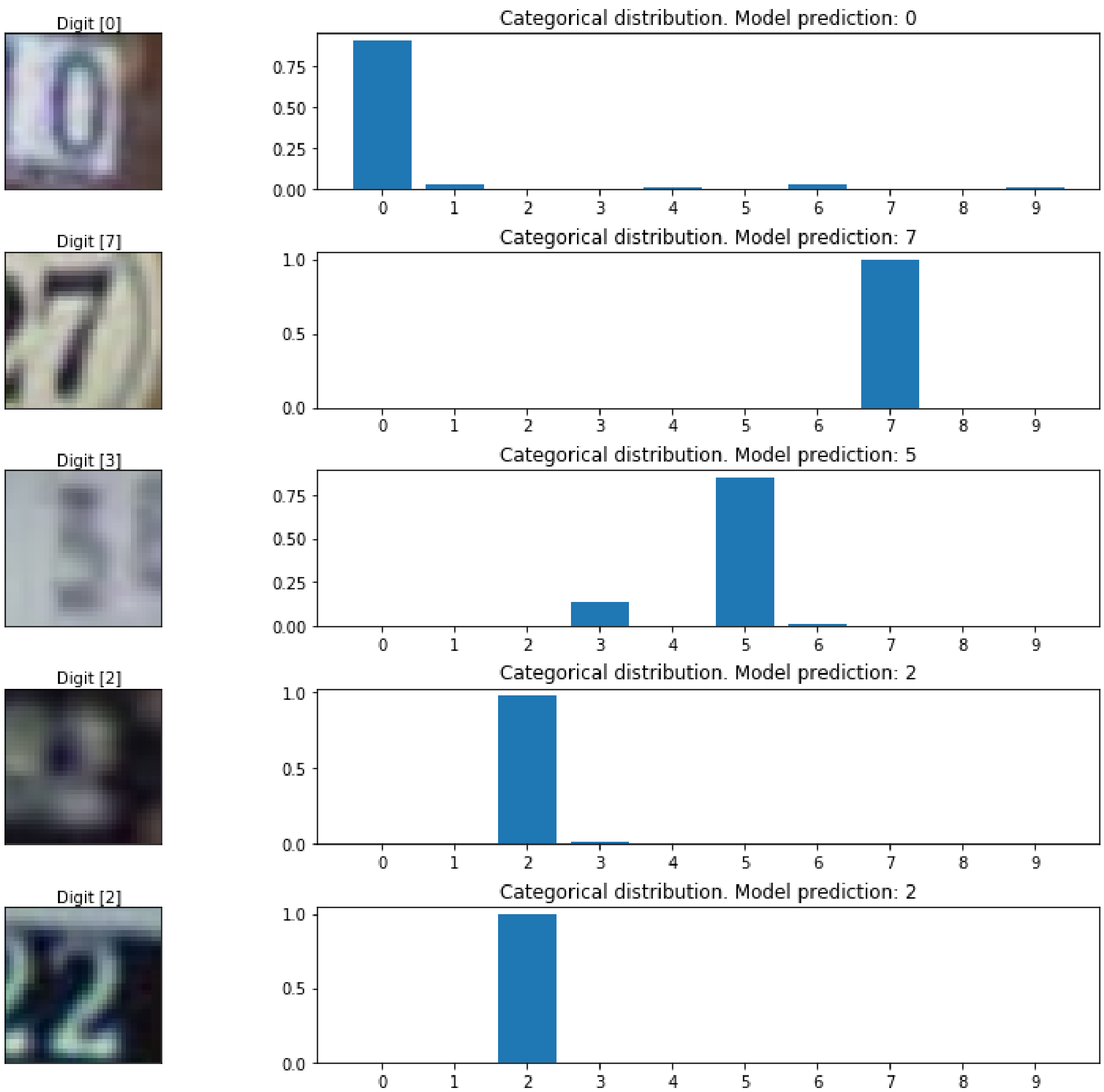
```
In [34]: Best_CNN = get_CNN_model(X_train[0].shape)
         Best_CNN.load_weights('Checkpoint/best_model_CNN')
```

```
Out[34]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f39b3ee67b8>
```

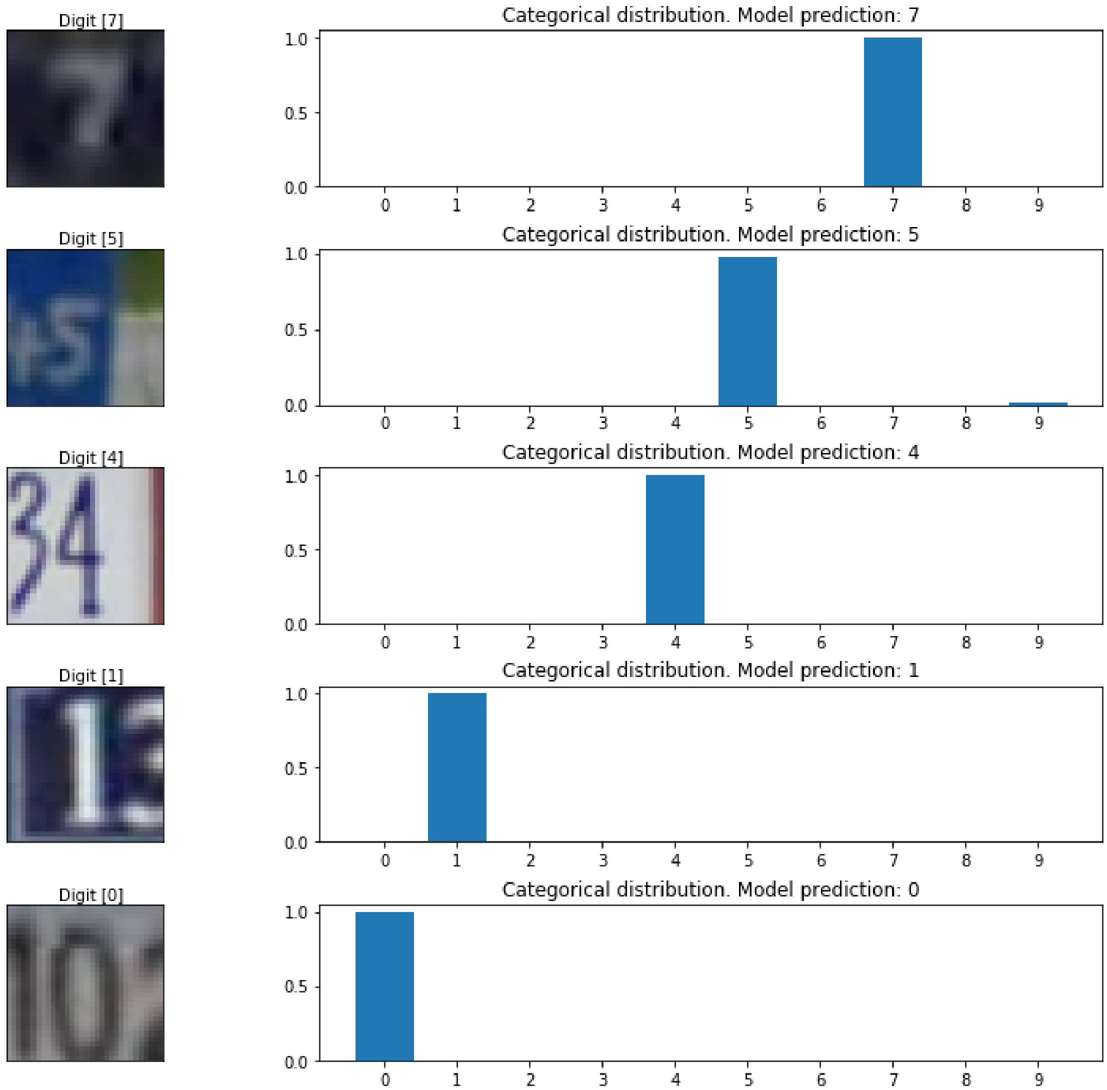
```
In [35]: def show_predictive_distribution(model):

             num_test_images = X_test.shape[0]
```

```
random_inx = np.random.choice(num_test_images, 5)
random_test_images = X_test[random_inx, ...]
random_test_labels = y_test[random_inx, ...]

predictions = model.predict(random_test_images)

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

for i, (prediction, image, label) in enumerate(zip(predictions, random_test_image
    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()
```

```
In [36]: show_predictive_distribution(Best_MLP)
```




```
In [37]: show_predictive_distribution(Best_CNN)
```



Digit [7]
Categorical distribution. Model prediction: 7
Digit [5]
Categorical distribution. Model prediction: 5
Digit [4]
Categorical distribution. Model prediction: 4
Digit [1]
Categorical distribution. Model prediction: 1
Digit [0]
Categorical distribution. Model prediction: 0


```
In [ ]:
```