Capstone Project

Image classifier for the SVHN dataset

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.

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.

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
```

```
from scipy.io import loadmat
import random
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Flatten,BatchNormalization,So
ftmax,Conv2D,Dropout,MaxPooling2D
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.optimizers import Adam
```

 SVHN overview 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.

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. Inspect and preprocess the dataset

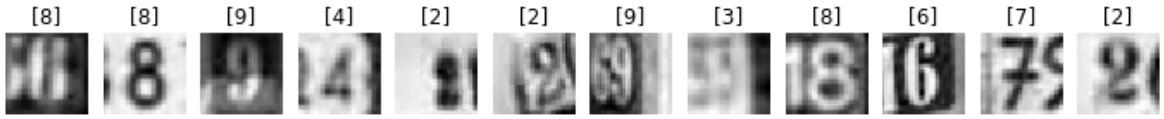
- Extract the training and testing images and labels separately from the train and test dictionaries loaded for you.

- 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']
#print(y_test.shape[0])
for i in range(y_train.shape[0]):
    y_train[i]=y_train[i]%10
for i in range(y_test.shape[0]):
    y_test[i]=y_test[i]%10
```

In [4]:
```
X_train=np.moveaxis(X_train,-1,0)
X_test=np.moveaxis(X_test,-1,0)
#print(X_train.shape) (73257,32,32,3)
random.seed(7836)
ilst=[]
for i in range(73257):
    ilst.append(i)
slst=random.sample(ilst,12)

fig,ax=plt.subplots(1,12,figsize=(12,1))
for i in range(12):
    ax[i].set_axis_off()
    ax[i].imshow(X_train[slst[i]])
    ax[i].set_title(y_train[slst[i]])
```

In [5]:
```
X_train_gray=np.mean(X_train,-1,keepdims=True)
X_test_gray=np.mean(X_test,-1,keepdims=True)

fig,ax=plt.subplots(1,12,figsize=(12,1))
for i in range(12):
    ax[i].set_axis_off()
    ax[i].imshow(X_train_gray[slst[i],:,:,0],cmap='gray')
    ax[i].set_title(y_train[slst[i]])
```



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 [27]:
```
def MLP_model():
    model=Sequential()
```

```
    model.add(Flatten(input_shape=X_train[0].shape))
    model.add(Dense(512,activation="relu"))
    model.add(Dense(256,activation="relu"))
    model.add(Dense(128,activation="relu"))
    model.add(Dense(128,activation="relu"))
    model.add(Dense(128,activation="relu"))
    model.add(Dense(10,activation="softmax"))

    return model

model=MLP_model()
print(model.summary())
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_5 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_21 (Dense)             (None, 512)               1573376
_________________________________________________________________
dense_22 (Dense)             (None, 256)               131328
_________________________________________________________________
dense_23 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_24 (Dense)             (None, 128)               16512
_________________________________________________________________
dense_25 (Dense)             (None, 128)               16512
_________________________________________________________________
dense_26 (Dense)             (None, 10)                1290
=================================================================
Total params: 1,771,914
Trainable params: 1,771,914
Non-trainable params: 0
_________________________________________________________________
None
```

In [28]:
```
cp1_path='MLP_Checkpoint/best_model'
```

```
cp1=ModelCheckpoint(cp1_path,save_best_only=True,
                    save_weights_only=True,
                    verbose=False,save_freq='epoch',
                    monitor='val_loss',mode='min')
early_stop=EarlyStopping(monitor='val_loss',mode='min',patience=3)
```

In [29]:
```
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',m
etrics=['accuracy'])
```

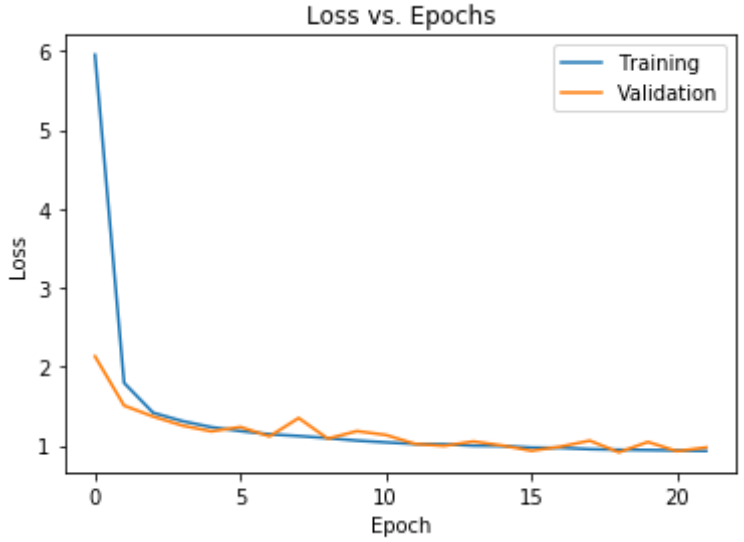
In [30]:
```
history=model.fit(X_train,y_train,epochs=25,batch_size=32,
                  verbose=1,validation_split=0.15,
                  callbacks=[cp1,early_stop])
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/25
62268/62268 [==============================] - 125s 2ms/sample - loss:
5.9511 - accuracy: 0.1591 - val_loss: 2.1303 - val_accuracy: 0.2474
Epoch 2/25
62268/62268 [==============================] - 123s 2ms/sample - loss:
1.7941 - accuracy: 0.3790 - val_loss: 1.5064 - val_accuracy: 0.5035
Epoch 3/25
62268/62268 [==============================] - 125s 2ms/sample - loss:
1.4152 - accuracy: 0.5362 - val_loss: 1.3678 - val_accuracy: 0.5716
Epoch 4/25
62268/62268 [==============================] - 124s 2ms/sample - loss:
1.3099 - accuracy: 0.5802 - val_loss: 1.2536 - val_accuracy: 0.6064
Epoch 5/25
62268/62268 [==============================] - 123s 2ms/sample - loss:
1.2326 - accuracy: 0.6104 - val_loss: 1.1797 - val_accuracy: 0.6325
Epoch 6/25
62268/62268 [==============================] - 123s 2ms/sample - loss:
1.1834 - accuracy: 0.6276 - val_loss: 1.2346 - val_accuracy: 0.6180
Epoch 7/25
62268/62268 [==============================] - 124s 2ms/sample - loss:
1.1429 - accuracy: 0.6419 - val_loss: 1.1159 - val_accuracy: 0.6479
Epoch 8/25
62268/62268 [==============================] - 123s 2ms/sample - loss:
1.1207 - accuracy: 0.6507 - val_loss: 1.3488 - val_accuracy: 0.5771

Epoch 9/25
```

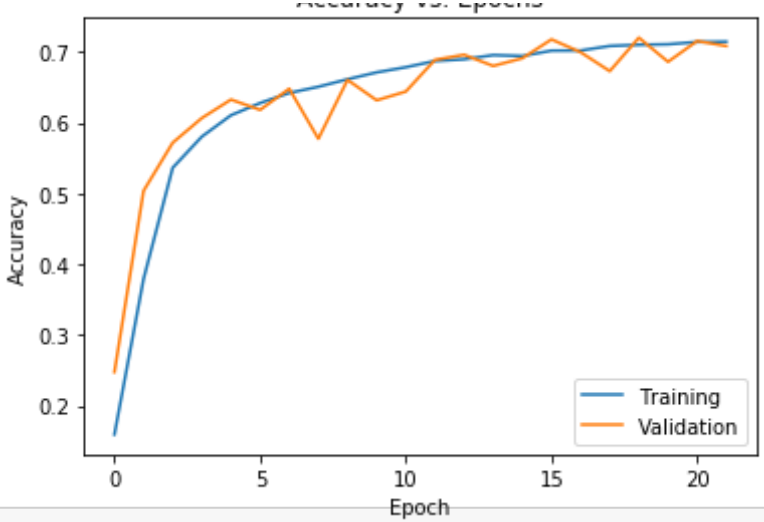
```
62268/62268 [==============================] - 123s 2ms/sample - loss:
1.0942 - accuracy: 0.6613 - val_loss: 1.0871 - val_accuracy: 0.6604
Epoch 10/25
62268/62268 [==============================] - 124s 2ms/sample - loss:
1.0629 - accuracy: 0.6710 - val_loss: 1.1827 - val_accuracy: 0.6315
Epoch 11/25
62268/62268 [==============================] - 115s 2ms/sample - loss:
1.0402 - accuracy: 0.6783 - val_loss: 1.1336 - val_accuracy: 0.6439
Epoch 12/25
62268/62268 [==============================] - 115s 2ms/sample - loss:
1.0202 - accuracy: 0.6871 - val_loss: 1.0205 - val_accuracy: 0.6887
Epoch 13/25
62268/62268 [==============================] - 122s 2ms/sample - loss:
1.0155 - accuracy: 0.6895 - val_loss: 0.9956 - val_accuracy: 0.6959
Epoch 14/25
62268/62268 [==============================] - 115s 2ms/sample - loss:
0.9981 - accuracy: 0.6954 - val_loss: 1.0515 - val_accuracy: 0.6801
Epoch 15/25
62268/62268 [==============================] - 117s 2ms/sample - loss:
0.9941 - accuracy: 0.6942 - val_loss: 1.0002 - val_accuracy: 0.6910
Epoch 16/25
62268/62268 [==============================] - 115s 2ms/sample - loss:
0.9737 - accuracy: 0.7017 - val_loss: 0.9335 - val_accuracy: 0.7176
Epoch 17/25
62268/62268 [==============================] - 121s 2ms/sample - loss:
0.9707 - accuracy: 0.7019 - val_loss: 0.9861 - val_accuracy: 0.6995
Epoch 18/25
62268/62268 [==============================] - 118s 2ms/sample - loss:
0.9542 - accuracy: 0.7083 - val_loss: 1.0619 - val_accuracy: 0.6728
Epoch 19/25
62268/62268 [==============================] - 123s 2ms/sample - loss:
0.9469 - accuracy: 0.7100 - val_loss: 0.9141 - val_accuracy: 0.7200
Epoch 20/25
62268/62268 [==============================] - 123s 2ms/sample - loss:
0.9426 - accuracy: 0.7107 - val_loss: 1.0453 - val_accuracy: 0.6860
Epoch 21/25
62268/62268 [==============================] - 126s 2ms/sample - loss:
0.9374 - accuracy: 0.7140 - val_loss: 0.9282 - val_accuracy: 0.7154
Epoch 22/25
62268/62268 [==============================] - 128s 2ms/sample - loss:
```

```
0.9324 - accuracy: 0.7144 - val_loss: 0.9757 - val_accuracy: 0.7083
```

In [31]:
```
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")
plt.show()

fig.add_subplot(122)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title("Accuracy vs. Epochs")
plt.ylabel("Accuracy")
plt.xlabel("Epoch")
plt.legend(["Training","Validation"],loc="lower right")
plt.show()
```



Accuracy vs. Epochs



In [32]:
```
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 1.124320526319409
Test Accuracy is 0.682275652885437
```

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 [17]:
```
def CNN_model():
    model=Sequential()
    model.add(Conv2D(16,(3,3),activation="relu", input_shape=X_train[0]
.shape))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2,2)))
    model.add(Conv2D(16,(3,3),activation="relu"))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2,2)))
    model.add(Flatten())
    model.add(Dense(128,activation="relu"))
    model.add(Dropout(0.3))
    model.add(Dense(128,activation="relu"))
    model.add(Dropout(0.3))
    model.add(Dense(10,activation="softmax"))

    return model

model=CNN_model()
print(model.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 30, 30, 16)        448
_________________________________________________________________
batch_normalization_4 (Batch (None, 30, 30, 16)        64
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 15, 15, 16)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 13, 13, 16)        2320
_________________________________________________________________
batch_normalization_5 (Batch (None, 13, 13, 16)        64
```

```
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 6, 6, 16)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_12 (Dense)             (None, 128)               73856
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_13 (Dense)             (None, 128)               16512
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_14 (Dense)             (None, 10)                1290
=================================================================
Total params: 94,554
Trainable params: 94,490
Non-trainable params: 64
_________________________________________________________________
None
```

In [18]:
```
cp2_path='CNN_Checkpoint/best_model'
cp2=ModelCheckpoint(cp2_path,save_best_only=True,
                    save_weights_only=True,
                    verbose=False,
                    save_freq='epoch',
                    monitor='val_loss',
                    mode='min')
early_stop=EarlyStopping(monitor='val_loss',
                         mode='min',
                         patience=3)
```

In [19]:
```
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',m
etrics=['accuracy'])
```

In [20]:
```
history=model.fit(X_train,y_train,epochs=25,
                  batch_size=32,verbose=1,
```

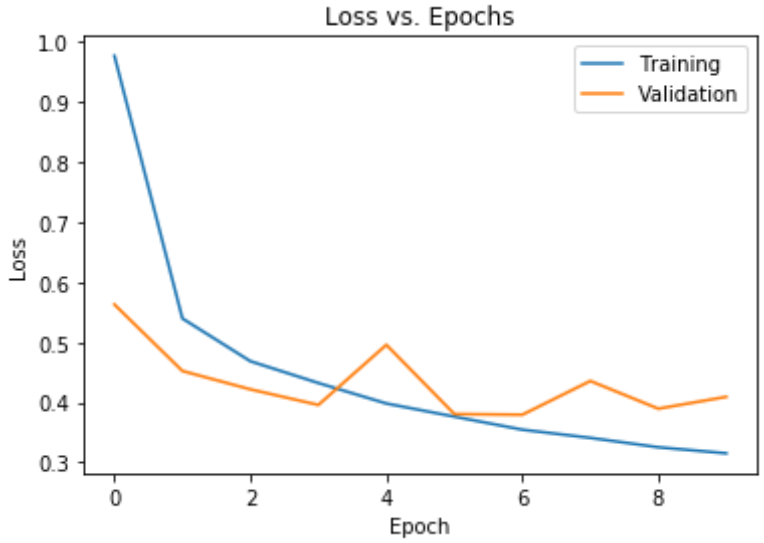
```
                    validation_split=0.15,
                    callbacks=[cp2,early_stop])
```

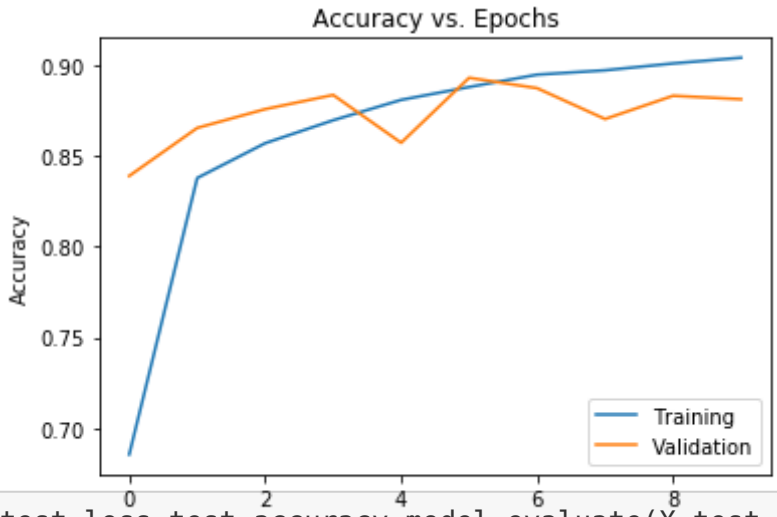
```
Train on 62268 samples, validate on 10989 samples
Epoch 1/25
62268/62268 [==============================] - 318s 5ms/sample - loss:
0.9774 - accuracy: 0.6853 - val_loss: 0.5632 - val_accuracy: 0.8390
Epoch 2/25
62268/62268 [==============================] - 314s 5ms/sample - loss:
0.5399 - accuracy: 0.8379 - val_loss: 0.4524 - val_accuracy: 0.8655
Epoch 3/25
62268/62268 [==============================] - 313s 5ms/sample - loss:
0.4687 - accuracy: 0.8571 - val_loss: 0.4219 - val_accuracy: 0.8759
Epoch 4/25
62268/62268 [==============================] - 313s 5ms/sample - loss:
0.4324 - accuracy: 0.8697 - val_loss: 0.3959 - val_accuracy: 0.8837
Epoch 5/25
62268/62268 [==============================] - 316s 5ms/sample - loss:
0.3983 - accuracy: 0.8810 - val_loss: 0.4960 - val_accuracy: 0.8573
Epoch 6/25
62268/62268 [==============================] - 313s 5ms/sample - loss:
0.3764 - accuracy: 0.8881 - val_loss: 0.3807 - val_accuracy: 0.8932
Epoch 7/25
62268/62268 [==============================] - 309s 5ms/sample - loss:
0.3547 - accuracy: 0.8949 - val_loss: 0.3796 - val_accuracy: 0.8874
Epoch 8/25
62268/62268 [==============================] - 307s 5ms/sample - loss:
0.3410 - accuracy: 0.8973 - val_loss: 0.4358 - val_accuracy: 0.8704
Epoch 9/25
62268/62268 [==============================] - 307s 5ms/sample - loss:
0.3254 - accuracy: 0.9011 - val_loss: 0.3896 - val_accuracy: 0.8832
Epoch 10/25
62268/62268 [==============================] - 310s 5ms/sample - loss:
0.3153 - accuracy: 0.9042 - val_loss: 0.4094 - val_accuracy: 0.8813
```

In [21]:

```
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="super right")
plt.show()

fig.add_subplot(122)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title("Accuracy vs. Epochs")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training','Validation'],loc="lower right")
plt.show()
```





In [22]:
```
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 0.44843153155007537
Test accuracy is 0.8705439567565918
```

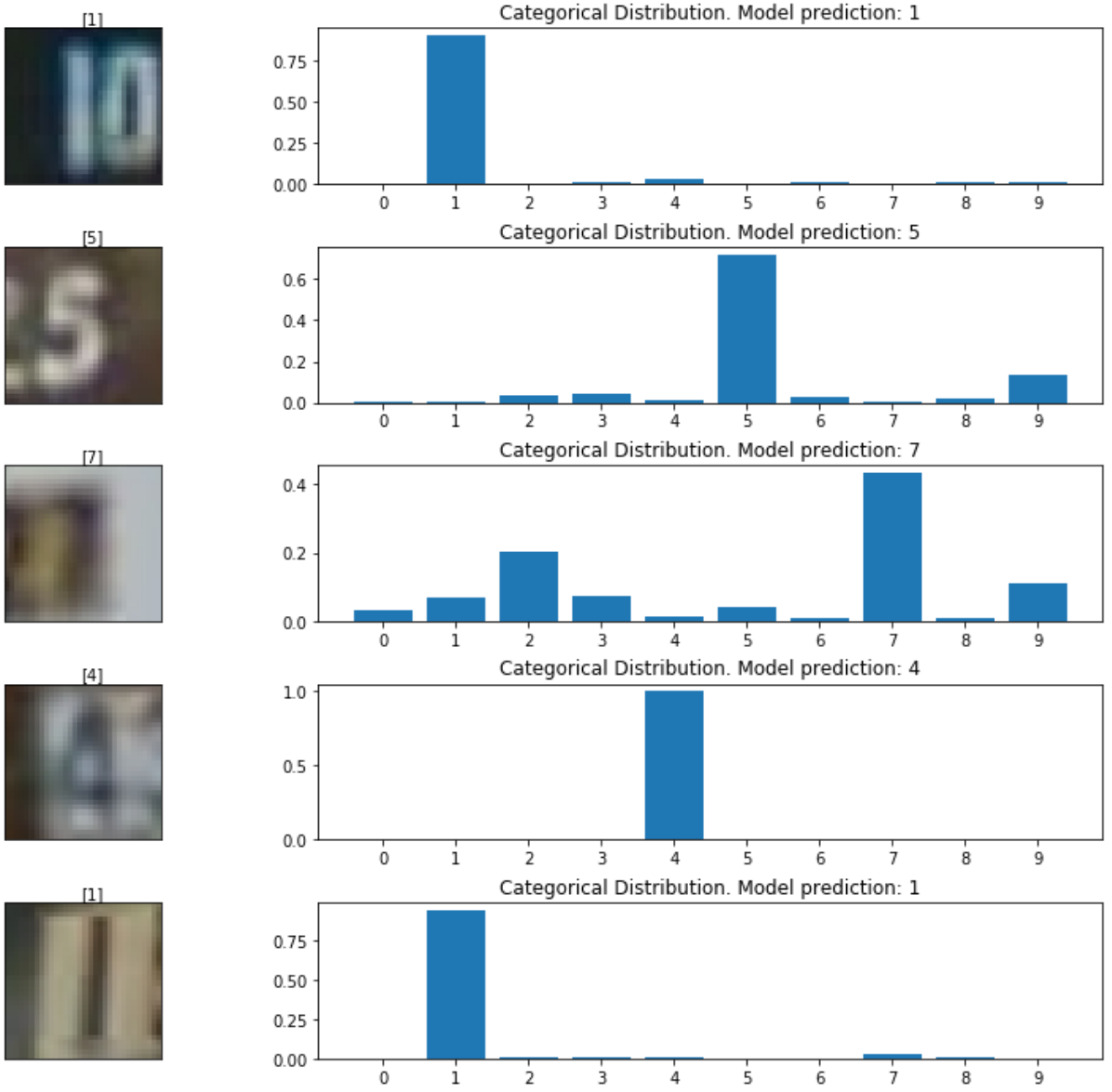
4. Get model predictions

- 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 [49]:
```
MLP=MLP_model()
MLP.load_weights('MLP_Checkpoint/best_model')
CNN=CNN_model()
CNN.load_weights('CNN_Checkpoint/best_model')
```

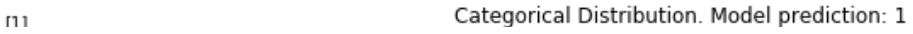
In [109]:
```
random.seed(1769)
```

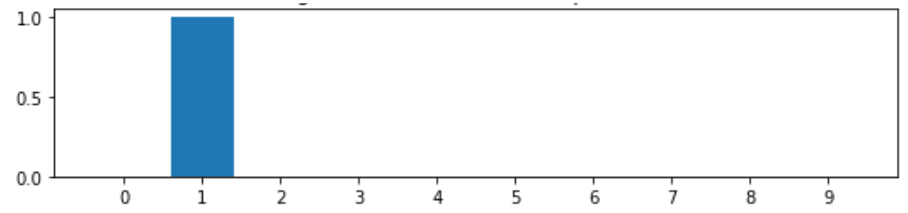
```
random_inx=np.random.choice(X_test.shape[0],5)

random_test_images=X_test[random_inx,...]
random_test_labels=y_test[random_inx,...]

# Multilayer Perceptron
mlp_predictions=MLP.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(mlp_predictions,random_
test_images,random_test_labels)):
    axes[i,0].imshow(image)
    axes[i,0].get_xaxis().set_visible(False)
    axes[i,0].get_yaxis().set_visible(False)
    axes[i,0].text(15,-1.5,f'{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(prediction)}")
```



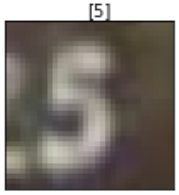
In [110]:
```
# Convolutional Neural Network
cnn_predictions=CNN.predict(random_test_images)
fig,axes=plt.subplots(5,2,figsize=(16,12))
```

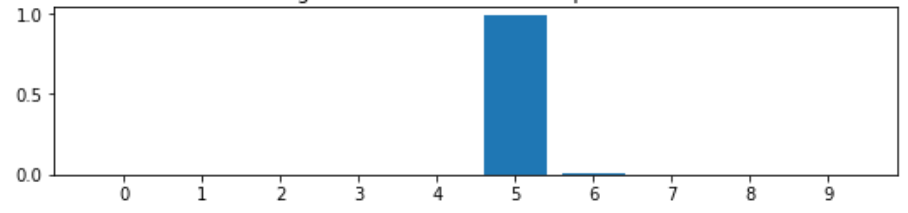
```
fig.subplots_adjust(hspace=0.4,wspace=-0.2)
for i,(prediction,image,label) in enumerate(zip(cnn_predictions,random_
test_images,random_test_labels)):
    axes[i,0].imshow(image)
    axes[i,0].get_xaxis().set_visible(False)
    axes[i,0].get_yaxis().set_visible(False)
    axes[i,0].text(15,-1.5,f'{label}')
    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(prediction)}")
```



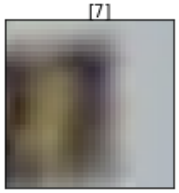


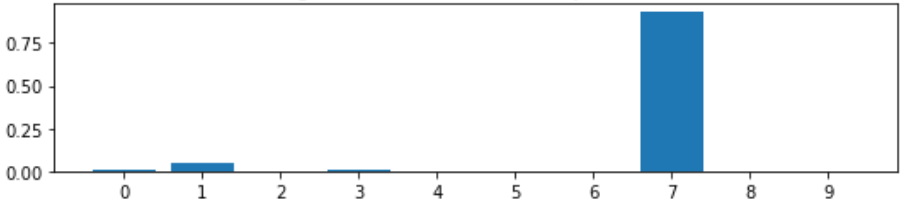
[5]



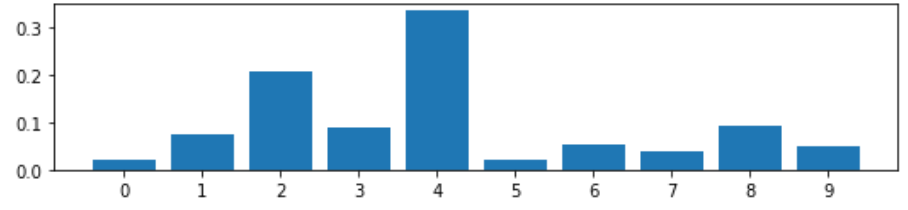


[7]

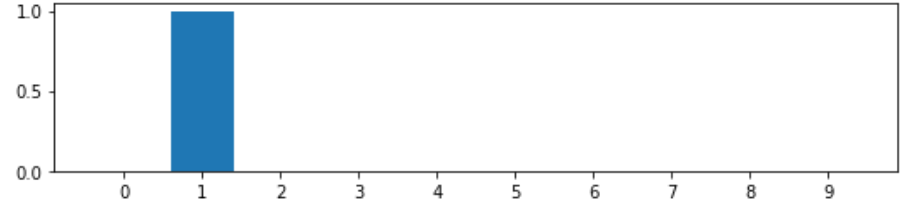




[4]



[1]



In [ ]: