Capstone_Project_Nicolas_Cardona

August 12, 2022

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 (you could download the notebook with File -> Download .ipynb, open the notebook locally, and then File -> Download as -> PDF via LaTeX), and 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.

[1]:
```
import tensorflow as tf
from scipy.io import loadmat
```

[2]:
```
# Libraries

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from tensorflow.keras.models import Sequential, load_model
```

```python
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D,␣
↪BatchNormalization, Dropout
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras import initializers, regularizers
```

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.

The train and test datasets required for this project can be downloaded from here and here. Once unzipped, you will have two files: `train_32x32.mat` and `test_32x32.mat`. You should store these files in Drive for use in this Colab notebook.

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.

[3]:
```python
# Run this cell to connect to your Drive folder

from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
```

[4]:
```python
# Load the dataset from your Drive folder
path = '/content/gdrive/MyDrive/Colab Notebooks/Data_Getting_Start_TF'

train = loadmat(path+'/train_32x32.mat')
test  = loadmat(path+'/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.

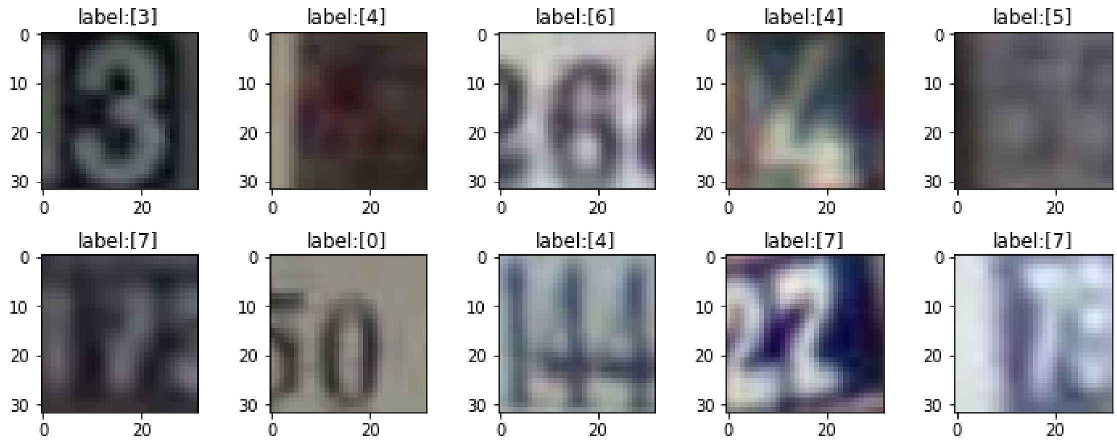
[5]:
```python
# Extract the training and testing images and labels separately from the train␣
↪and test dictionaries loaded for you.
```

```
x_train = train['X']/255.0
x_train = np.moveaxis(x_train, -1, 0)
x_test = test['X']/255.0
x_test = np.moveaxis(x_test, -1, 0)

y_train = train['y']
y_train[y_train == 10] = 0
y_test = test['y']
y_test[y_test == 10] = 0

print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

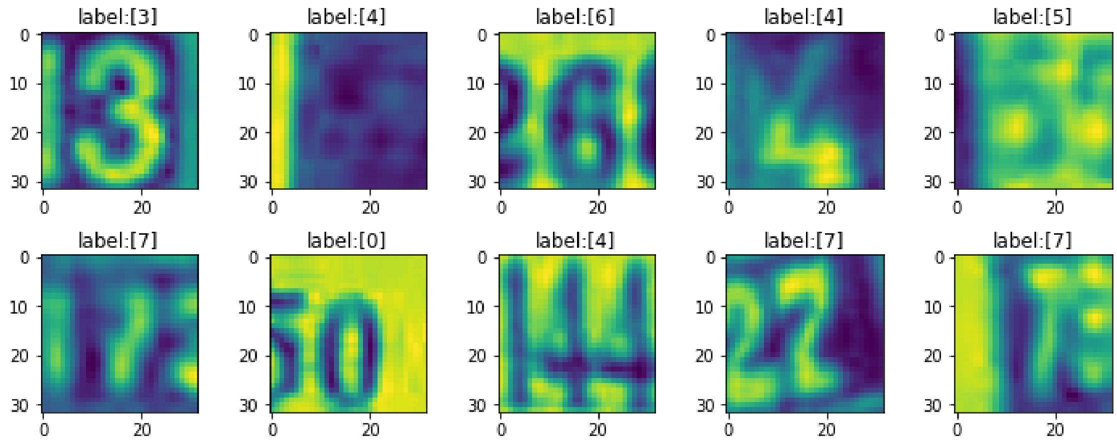
```
(73257, 32, 32, 3)
(26032, 32, 32, 3)
(73257, 1)
(26032, 1)
```

[6]:

```
# Select a random sample of images and corresponding labels from the dataset␣
↪(at least 10), and display them in a figure.
inicio = np.random.randint(x_train.shape[0], size=(1, 10))[0]

rows, cols = 2, 5
axes=[]
fig=plt.figure(figsize=(10,4))

for ind,i in enumerate(inicio):
    axes.append( fig.add_subplot(rows, cols, ind+1) )
    subplot_title=('label:'+ str(y_train[i]))
    axes[-1].set_title(subplot_title)
    img = x_train[i,:,:,:]
    plt.imshow(img)
fig.tight_layout()
plt.show()
```



[7]:
```
# 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.
x_train = np.mean(x_train, axis=3,keepdims=True)
x_test  = np.mean(x_test , axis=3,keepdims=True)

rows, cols = 2, 5
axes=[]
fig=plt.figure(figsize=(10,4))

for ind,i in enumerate(inicio):
    axes.append( fig.add_subplot(rows, cols, ind+1) )
    subplot_title=('label:'+ str(y_train[i]))
    axes[-1].set_title(subplot_title)
    img = x_train[i,:,:,0]
    #plt.imshow(img,interpolation='nearest', cmap=plt.get_cmap('gray'))
    plt.imshow(img)
fig.tight_layout()
plt.show()
```



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.

[8]:
```
def get_model():
    model = Sequential([
        Flatten(input_shape = (32,32,1)),
        Dense(64 , activation = 'softplus',
              kernel_initializer='he_normal', bias_initializer=tf.keras.
 ↪initializers.Constant(1.)), # activation = 'softplus',
        Dense(64 , activation = 'relu'),
        Dense(64 , activation = 'relu'),
        Dense(64 , activation = 'relu'),
        Dense(10 , activation = 'softmax')
    ])

    model.compile(optimizer='SGD',
```

```
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Run my function to get the model
model = get_model()
model.summary()
```

```
Model: "sequential"

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

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 64)                4160

 dense_2 (Dense)             (None, 64)                4160

 dense_3 (Dense)             (None, 64)                4160

 dense_4 (Dense)             (None, 10)                650

=================================================================
Total params: 78,730
Trainable params: 78,730
Non-trainable params: 0
_________________________________________________________________
```

[23]:
```
# Run this cell to define a function to evaluate a model's test accuracy

def get_test_accuracy(model, x_test, y_test):
    """Test model classification accuracy"""
    test_loss, test_acc = model.evaluate(x=x_test, y=y_test, verbose=0)
    print('loss: {acc:0.3f}'.format(acc=test_loss))
    print('accuracy: {acc:0.3f}'.format(acc=test_acc))

get_test_accuracy(model, x_test, y_test)
```

```
loss: 0.965
accuracy: 0.710
```

[10]:
```
# Creating Checkpoints

def get_checkpoint_best_only():
    checkpoint_best_path = 'checkpoints_best_MLP/checkpoint'
    checkpoint_best = ModelCheckpoint(filepath=checkpoint_best_path,
```

```
                                             monitor='val_accuracy',
                                             save_weights_only=True,
                                             save_freq='epoch',
                                             save_best_only = True,
                                             verbose=1)
    return checkpoint_best

def get_early_stopping():
    callback = tf.keras.callbacks.EarlyStopping(patience=5,monitor="accuracy")
    return callback


checkpoint_best_only = get_checkpoint_best_only()
early_stopping = get_early_stopping()
callbacks = [checkpoint_best_only, early_stopping]
```

[11]:
```
history = model.fit(x_train, y_train, epochs=25,
                    validation_data=(x_test, y_test),
                    batch_size = 64, callbacks=callbacks)
```

```
Epoch 1/25
1140/1145 [============================>.] - ETA: 0s - loss: 2.2382 - accuracy:
0.1884
Epoch 1: val_accuracy improved from -inf to 0.19587, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 5s 4ms/step - loss: 2.2383 -
accuracy: 0.1883 - val_loss: 2.2179 - val_accuracy: 0.1959
Epoch 2/25
1145/1145 [==============================] - ETA: 0s - loss: 2.2232 - accuracy:
0.1910
Epoch 2: val_accuracy improved from 0.19587 to 0.20828, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 3ms/step - loss: 2.2232 -
accuracy: 0.1910 - val_loss: 2.1891 - val_accuracy: 0.2083
Epoch 3/25
1137/1145 [============================>.] - ETA: 0s - loss: 2.1554 - accuracy:
0.2289
Epoch 3: val_accuracy improved from 0.20828 to 0.27347, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 5s 4ms/step - loss: 2.1548 -
accuracy: 0.2293 - val_loss: 2.0709 - val_accuracy: 0.2735
Epoch 4/25
1136/1145 [============================>.] - ETA: 0s - loss: 1.9762 - accuracy:
0.3110
Epoch 4: val_accuracy improved from 0.27347 to 0.36086, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 5s 5ms/step - loss: 1.9752 -
accuracy: 0.3112 - val_loss: 1.8408 - val_accuracy: 0.3609
```

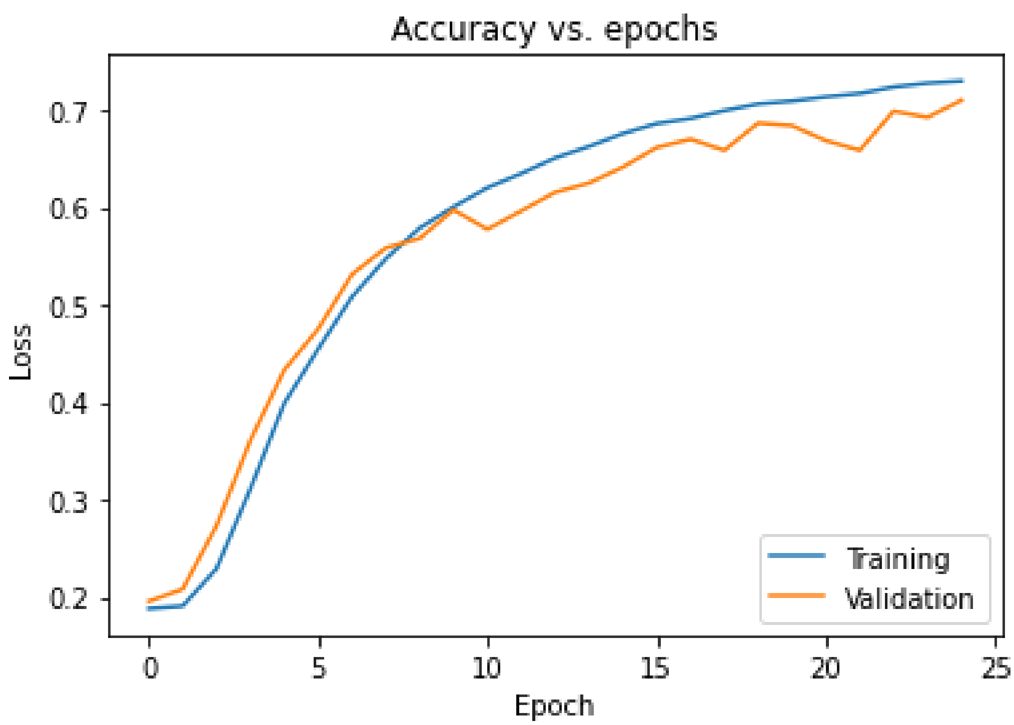
```
Epoch 5/25
1145/1145 [==============================] - ETA: 0s - loss: 1.7454 - accuracy:
0.3987
Epoch 5: val_accuracy improved from 0.36086 to 0.43301, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 4ms/step - loss: 1.7454 -
accuracy: 0.3987 - val_loss: 1.6456 - val_accuracy: 0.4330
Epoch 6/25
1138/1145 [============================>.] - ETA: 0s - loss: 1.5764 - accuracy:
0.4540
Epoch 6: val_accuracy improved from 0.43301 to 0.47499, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 5s 4ms/step - loss: 1.5756 -
accuracy: 0.4544 - val_loss: 1.5521 - val_accuracy: 0.4750
Epoch 7/25
1139/1145 [============================>.] - ETA: 0s - loss: 1.4598 - accuracy:
0.5081
Epoch 7: val_accuracy improved from 0.47499 to 0.53108, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 4ms/step - loss: 1.4601 -
accuracy: 0.5080 - val_loss: 1.4553 - val_accuracy: 0.5311
Epoch 8/25
1140/1145 [============================>.] - ETA: 0s - loss: 1.3627 - accuracy:
0.5471
Epoch 8: val_accuracy improved from 0.53108 to 0.55827, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 7s 6ms/step - loss: 1.3630 -
accuracy: 0.5471 - val_loss: 1.3752 - val_accuracy: 0.5583
Epoch 9/25
1131/1145 [============================>.] - ETA: 0s - loss: 1.2868 - accuracy:
0.5787
Epoch 9: val_accuracy improved from 0.55827 to 0.56799, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 6s 5ms/step - loss: 1.2870 -
accuracy: 0.5787 - val_loss: 1.3701 - val_accuracy: 0.5680
Epoch 10/25
1132/1145 [============================>.] - ETA: 0s - loss: 1.2356 - accuracy:
0.6003
Epoch 10: val_accuracy improved from 0.56799 to 0.59726, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 5s 4ms/step - loss: 1.2350 -
accuracy: 0.6003 - val_loss: 1.2834 - val_accuracy: 0.5973
Epoch 11/25
1131/1145 [============================>.] - ETA: 0s - loss: 1.1877 - accuracy:
0.6201
Epoch 11: val_accuracy did not improve from 0.59726
1145/1145 [==============================] - 4s 3ms/step - loss: 1.1873 -
accuracy: 0.6200 - val_loss: 1.3208 - val_accuracy: 0.5774
```

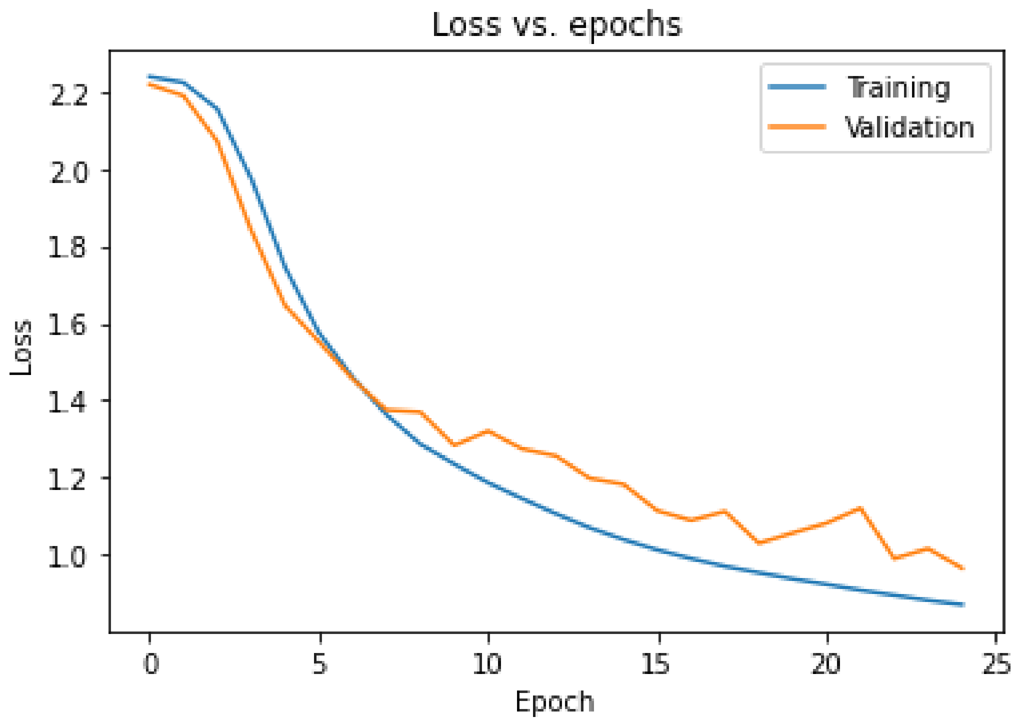
```
Epoch 12/25
1144/1145 [============================>.] - ETA: 0s - loss: 1.1462 - accuracy:
0.6345
Epoch 12: val_accuracy did not improve from 0.59726
1145/1145 [==============================] - 4s 4ms/step - loss: 1.1462 -
accuracy: 0.6346 - val_loss: 1.2747 - val_accuracy: 0.5962
Epoch 13/25
1133/1145 [============================>.] - ETA: 0s - loss: 1.1076 - accuracy:
0.6504
Epoch 13: val_accuracy improved from 0.59726 to 0.61532, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 3ms/step - loss: 1.1069 -
accuracy: 0.6506 - val_loss: 1.2565 - val_accuracy: 0.6153
Epoch 14/25
1134/1145 [============================>.] - ETA: 0s - loss: 1.0700 - accuracy:
0.6625
Epoch 14: val_accuracy improved from 0.61532 to 0.62477, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 3ms/step - loss: 1.0704 -
accuracy: 0.6624 - val_loss: 1.1988 - val_accuracy: 0.6248
Epoch 15/25
1135/1145 [============================>.] - ETA: 0s - loss: 1.0394 - accuracy:
0.6757
Epoch 15: val_accuracy improved from 0.62477 to 0.64106, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 4ms/step - loss: 1.0397 -
accuracy: 0.6755 - val_loss: 1.1830 - val_accuracy: 0.6411
Epoch 16/25
1129/1145 [============================>.] - ETA: 0s - loss: 1.0125 - accuracy:
0.6861
Epoch 16: val_accuracy improved from 0.64106 to 0.66138, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 3ms/step - loss: 1.0130 -
accuracy: 0.6859 - val_loss: 1.1145 - val_accuracy: 0.6614
Epoch 17/25
1138/1145 [============================>.] - ETA: 0s - loss: 0.9904 - accuracy:
0.6911
Epoch 17: val_accuracy improved from 0.66138 to 0.66983, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 3ms/step - loss: 0.9904 -
accuracy: 0.6911 - val_loss: 1.0896 - val_accuracy: 0.6698
Epoch 18/25
1144/1145 [============================>.] - ETA: 0s - loss: 0.9703 - accuracy:
0.6992
Epoch 18: val_accuracy did not improve from 0.66983
1145/1145 [==============================] - 4s 4ms/step - loss: 0.9703 -
accuracy: 0.6992 - val_loss: 1.1127 - val_accuracy: 0.6585
Epoch 19/25
```

```
1131/1145 [============================>.] - ETA: 0s - loss: 0.9546 - accuracy:
0.7059
Epoch 19: val_accuracy improved from 0.66983 to 0.68642, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 4ms/step - loss: 0.9539 -
accuracy: 0.7060 - val_loss: 1.0304 - val_accuracy: 0.6864
Epoch 20/25
1129/1145 [============================>.] - ETA: 0s - loss: 0.9372 - accuracy:
0.7092
Epoch 20: val_accuracy did not improve from 0.68642
1145/1145 [==============================] - 4s 3ms/step - loss: 0.9382 -
accuracy: 0.7090 - val_loss: 1.0570 - val_accuracy: 0.6837
Epoch 21/25
1130/1145 [============================>.] - ETA: 0s - loss: 0.9227 - accuracy:
0.7135
Epoch 21: val_accuracy did not improve from 0.68642
1145/1145 [==============================] - 4s 4ms/step - loss: 0.9240 -
accuracy: 0.7134 - val_loss: 1.0825 - val_accuracy: 0.6683
Epoch 22/25
1132/1145 [============================>.] - ETA: 0s - loss: 0.9086 - accuracy:
0.7171
Epoch 22: val_accuracy did not improve from 0.68642
1145/1145 [==============================] - 5s 4ms/step - loss: 0.9090 -
accuracy: 0.7168 - val_loss: 1.1212 - val_accuracy: 0.6585
Epoch 23/25
1129/1145 [============================>.] - ETA: 0s - loss: 0.8957 - accuracy:
0.7233
Epoch 23: val_accuracy improved from 0.68642 to 0.69864, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 4ms/step - loss: 0.8959 -
accuracy: 0.7233 - val_loss: 0.9905 - val_accuracy: 0.6986
Epoch 24/25
1130/1145 [============================>.] - ETA: 0s - loss: 0.8834 - accuracy:
0.7269
Epoch 24: val_accuracy did not improve from 0.69864
1145/1145 [==============================] - 4s 4ms/step - loss: 0.8827 -
accuracy: 0.7272 - val_loss: 1.0162 - val_accuracy: 0.6923
Epoch 25/25
1138/1145 [============================>.] - ETA: 0s - loss: 0.8715 - accuracy:
0.7294
Epoch 25: val_accuracy improved from 0.69864 to 0.70974, saving model to
checkpoints_best_MLP/checkpoint
1145/1145 [==============================] - 4s 4ms/step - loss: 0.8715 -
accuracy: 0.7294 - val_loss: 0.9655 - val_accuracy: 0.7097
```

[12]:
```
# Run this cell to plot the epoch vs accuracy graph
```

```
try:
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
except KeyError:
    plt.plot(history.history['acc'])
    plt.plot(history.history['val_acc'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.show()

#Run this cell to plot the epoch vs loss graph
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()
```





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.

[13]:
```
def get_new_model(dropout_rate):
    model = Sequential([
        Conv2D(filters=8, input_shape=(32,32,1), kernel_size=(3, 3),
               activation='softplus',padding='same', name='conv_1'),
        Conv2D(filters=8, kernel_size=(3, 3), activation='relu',padding='same',␣
↪name='conv_2'),
```

```
        MaxPooling2D(pool_size=(8, 8), name='pool_1'),
        Flatten(name='flatten'),
        Dense(units=64 , activation='sigmoid', name='dense_1'),␣
↪#kernel_regularizer = regularizers.l2(wd),
        Dropout(dropout_rate),
        Dense(units=128, activation='sigmoid', name='dense_2'),
        BatchNormalization(),
        Dense(units=64 , activation='relu', name='dense_3'),
        Dropout(dropout_rate),
        Dense(units=10 , activation='softmax', name='dense_4')
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

model_CNN = get_new_model(0.1)
model_CNN.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_1 (Conv2D)             (None, 32, 32, 8)         80

 conv_2 (Conv2D)             (None, 32, 32, 8)         584

 pool_1 (MaxPooling2D)       (None, 4, 4, 8)           0

 flatten (Flatten)           (None, 128)               0

 dense_1 (Dense)             (None, 64)                8256

 dropout (Dropout)           (None, 64)                0

 dense_2 (Dense)             (None, 128)               8320

 batch_normalization (BatchN  (None, 128)              512
 ormalization)

 dense_3 (Dense)             (None, 64)                8256

 dropout_1 (Dropout)         (None, 64)                0

 dense_4 (Dense)             (None, 10)                650

=================================================================
Total params: 26,658
```

```
Trainable params: 26,402
Non-trainable params: 256
_________________________________________________________________
```

[14]:
```
# Creating Checkpoints

def get_checkpoint_best_only():
    checkpoint_best_path = 'checkpoints_best_CNN/checkpoint'
    checkpoint_best = ModelCheckpoint(filepath=checkpoint_best_path,
                                      monitor='val_accuracy',
                                      save_weights_only=True,
                                      save_freq='epoch',
                                      save_best_only = True,
                                      verbose=1)
    return checkpoint_best

def get_early_stopping():
    callback = tf.keras.callbacks.EarlyStopping(patience=5,monitor="accuracy")
    return callback


checkpoint_best_only = get_checkpoint_best_only()
early_stopping = get_early_stopping()
callbacks = [checkpoint_best_only, early_stopping]
```

[15]:
```
history_CNN = model_CNN.fit(x_train, y_train, epochs=15,
                    validation_data=(x_test, y_test),
                    batch_size = 32, callbacks=callbacks)
```

```
Epoch 1/15
2290/2290 [==============================] - ETA: 0s - loss: 1.9536 - accuracy:
0.3044
Epoch 1: val_accuracy improved from -inf to 0.46954, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 15s 6ms/step - loss: 1.9536 -
accuracy: 0.3044 - val_loss: 1.5647 - val_accuracy: 0.4695
Epoch 2/15
2289/2290 [============================>.] - ETA: 0s - loss: 1.4035 - accuracy:
0.5249
Epoch 2: val_accuracy improved from 0.46954 to 0.56096, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 12s 5ms/step - loss: 1.4035 -
accuracy: 0.5249 - val_loss: 1.3494 - val_accuracy: 0.5610
Epoch 3/15
2282/2290 [============================>.] - ETA: 0s - loss: 1.1732 - accuracy:
0.6140
Epoch 3: val_accuracy improved from 0.56096 to 0.63768, saving model to
checkpoints_best_CNN/checkpoint
```

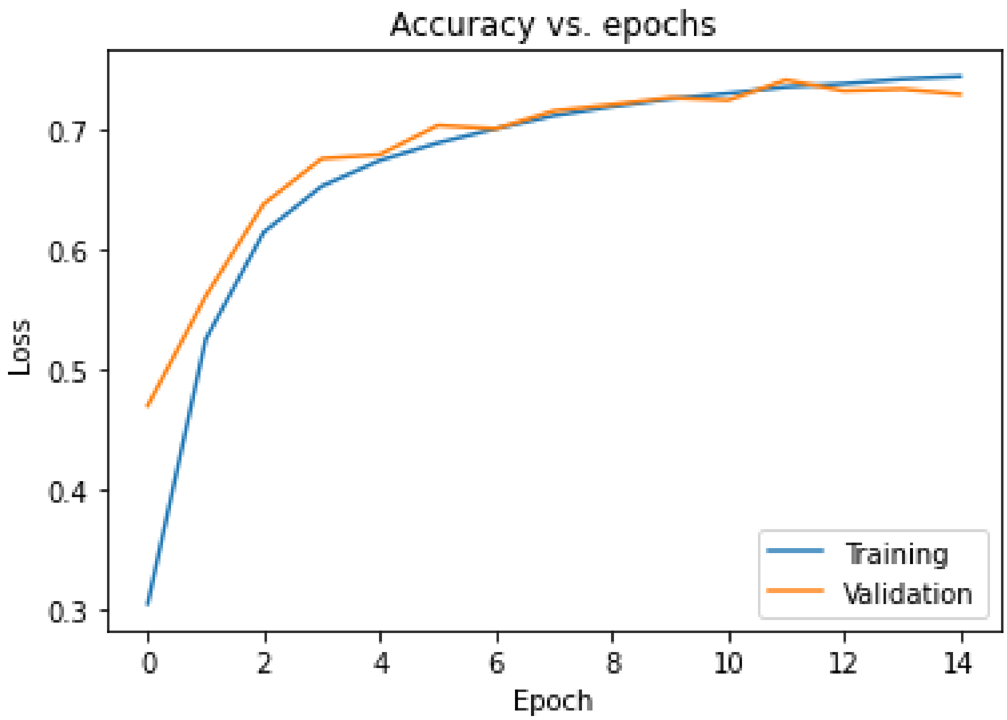
```
2290/2290 [==============================] - 12s 5ms/step - loss: 1.1731 -
accuracy: 0.6140 - val_loss: 1.1405 - val_accuracy: 0.6377
Epoch 4/15
2290/2290 [==============================] - ETA: 0s - loss: 1.0596 - accuracy:
0.6525
Epoch 4: val_accuracy improved from 0.63768 to 0.67551, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 12s 5ms/step - loss: 1.0596 -
accuracy: 0.6525 - val_loss: 1.0213 - val_accuracy: 0.6755
Epoch 5/15
2286/2290 [============================>.] - ETA: 0s - loss: 1.0004 - accuracy:
0.6739
Epoch 5: val_accuracy improved from 0.67551 to 0.67886, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 11s 5ms/step - loss: 1.0003 -
accuracy: 0.6740 - val_loss: 1.0020 - val_accuracy: 0.6789
Epoch 6/15
2280/2290 [============================>.] - ETA: 0s - loss: 0.9559 - accuracy:
0.6886
Epoch 6: val_accuracy improved from 0.67886 to 0.70325, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 12s 5ms/step - loss: 0.9561 -
accuracy: 0.6886 - val_loss: 0.9327 - val_accuracy: 0.7032
Epoch 7/15
2289/2290 [============================>.] - ETA: 0s - loss: 0.9186 - accuracy:
0.7005
Epoch 7: val_accuracy did not improve from 0.70325
2290/2290 [==============================] - 12s 5ms/step - loss: 0.9186 -
accuracy: 0.7005 - val_loss: 0.9502 - val_accuracy: 0.7004
Epoch 8/15
2290/2290 [==============================] - ETA: 0s - loss: 0.8908 - accuracy:
0.7113
Epoch 8: val_accuracy improved from 0.70325 to 0.71531, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 11s 5ms/step - loss: 0.8908 -
accuracy: 0.7113 - val_loss: 0.8894 - val_accuracy: 0.7153
Epoch 9/15
2283/2290 [============================>.] - ETA: 0s - loss: 0.8697 - accuracy:
0.7188
Epoch 9: val_accuracy improved from 0.71531 to 0.72054, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 12s 5ms/step - loss: 0.8694 -
accuracy: 0.7189 - val_loss: 0.8715 - val_accuracy: 0.7205
Epoch 10/15
2283/2290 [============================>.] - ETA: 0s - loss: 0.8520 - accuracy:
0.7252
Epoch 10: val_accuracy improved from 0.72054 to 0.72638, saving model to
checkpoints_best_CNN/checkpoint
```

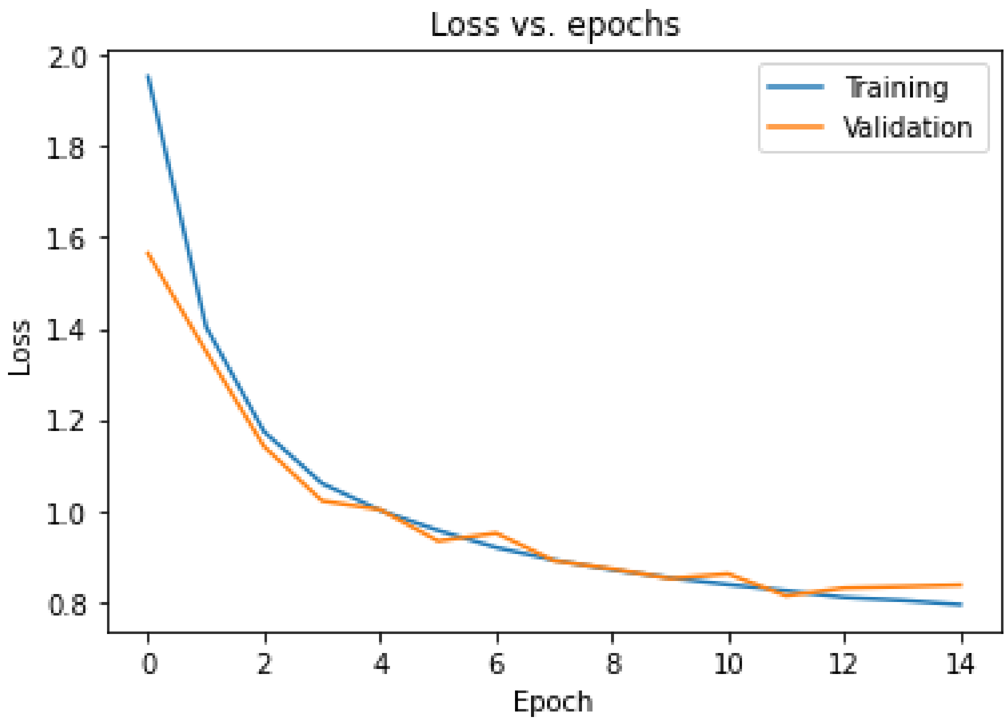
```
2290/2290 [==============================] - 12s 5ms/step - loss: 0.8519 -
accuracy: 0.7253 - val_loss: 0.8505 - val_accuracy: 0.7264
Epoch 11/15
2288/2290 [============================>.] - ETA: 0s - loss: 0.8372 - accuracy:
0.7299
Epoch 11: val_accuracy did not improve from 0.72638
2290/2290 [==============================] - 11s 5ms/step - loss: 0.8372 -
accuracy: 0.7299 - val_loss: 0.8605 - val_accuracy: 0.7242
Epoch 12/15
2288/2290 [============================>.] - ETA: 0s - loss: 0.8240 - accuracy:
0.7352
Epoch 12: val_accuracy improved from 0.72638 to 0.74093, saving model to
checkpoints_best_CNN/checkpoint
2290/2290 [==============================] - 12s 5ms/step - loss: 0.8239 -
accuracy: 0.7352 - val_loss: 0.8128 - val_accuracy: 0.7409
Epoch 13/15
2279/2290 [============================>.] - ETA: 0s - loss: 0.8091 - accuracy:
0.7379
Epoch 13: val_accuracy did not improve from 0.74093
2290/2290 [==============================] - 12s 5ms/step - loss: 0.8090 -
accuracy: 0.7378 - val_loss: 0.8297 - val_accuracy: 0.7321
Epoch 14/15
2289/2290 [============================>.] - ETA: 0s - loss: 0.8026 - accuracy:
0.7417
Epoch 14: val_accuracy did not improve from 0.74093
2290/2290 [==============================] - 11s 5ms/step - loss: 0.8026 -
accuracy: 0.7417 - val_loss: 0.8329 - val_accuracy: 0.7335
Epoch 15/15
2281/2290 [============================>.] - ETA: 0s - loss: 0.7936 - accuracy:
0.7438
Epoch 15: val_accuracy did not improve from 0.74093
2290/2290 [==============================] - 12s 5ms/step - loss: 0.7931 -
accuracy: 0.7439 - val_loss: 0.8363 - val_accuracy: 0.7292
```

[16]:

```
# Run this cell to plot the epoch vs accuracy graph

try:
    plt.plot(history_CNN.history['accuracy'])
    plt.plot(history_CNN.history['val_accuracy'])
except KeyError:
    plt.plot(history_CNN.history['acc'])
    plt.plot(history_CNN.history['val_acc'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.show()
```

```python
#Run this cell to plot the epoch vs loss graph
plt.plot(history_CNN.history['loss'])
plt.plot(history_CNN.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```





[17]:
```
# Run this cell to define a function to evaluate a model's test accuracy and␣
↪loss

def get_test_accuracy_CNN(model, x_test, y_test):
    """Test model classification accuracy"""
    test_loss, test_acc = model_CNN.evaluate(x=x_test, y=y_test, verbose=0)
    print('loss: {acc:0.3f}'.format(acc=test_loss))
    print('accuracy: {acc:0.3f}'.format(acc=test_acc))

get_test_accuracy_CNN(model_CNN, x_test, y_test)
```

```
loss: 0.836
accuracy: 0.729
```

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.

[18]:
```
print(tf.train.latest_checkpoint('checkpoints_best_MLP'))
print(tf.train.latest_checkpoint('checkpoints_best_CNN'))
```

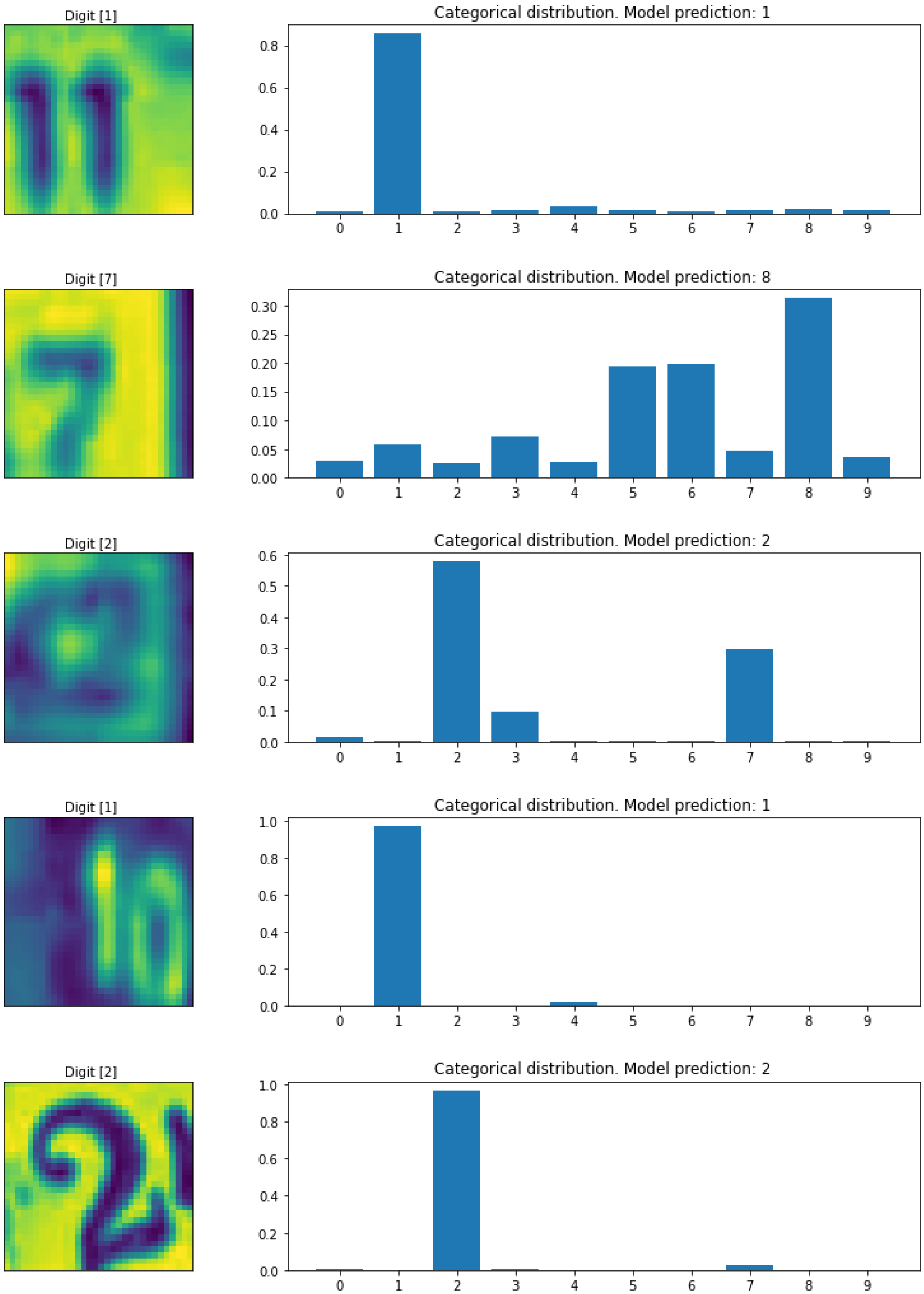
```
checkpoints_best_MLP/checkpoint
checkpoints_best_CNN/checkpoint
```

2 MLP - Predictions

[24]:
```
def get_model_best_epoch_MLP(model):
    checkpoint_path = 'checkpoints_best_MLP/checkpoint'
    model.load_weights(checkpoint_path)
    return model

model_best_epoch_MLP = get_model_best_epoch_MLP(get_model())

print('Model with best epoch weights:')
get_test_accuracy(model_best_epoch_MLP, x_test, y_test)
```

```
Model with best epoch weights:
loss: 0.965
accuracy: 0.710
```

[32]:
```
# Run this cell to get model predictions on randomly selected test images

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, 18))
fig.subplots_adjust(hspace=0.4, wspace=-0.2)

for i, (prediction, image, label) in enumerate(zip(predictions,␣
 ↪random_test_images, random_test_labels)):
    axes[i, 0].imshow(np.squeeze(image))
    axes[i, 0].get_xaxis().set_visible(False)
    axes[i, 0].get_yaxis().set_visible(False)
    axes[i, 0].text(10., -1.5, f'Digit {label}')
    axes[i, 1].bar(np.arange(len(prediction)), prediction)
    axes[i, 1].set_xticks(np.arange(len(prediction)))
    axes[i, 1].set_title(f"Categorical distribution. Model prediction: {np.
 ↪argmax(prediction)}")

plt.show()
```

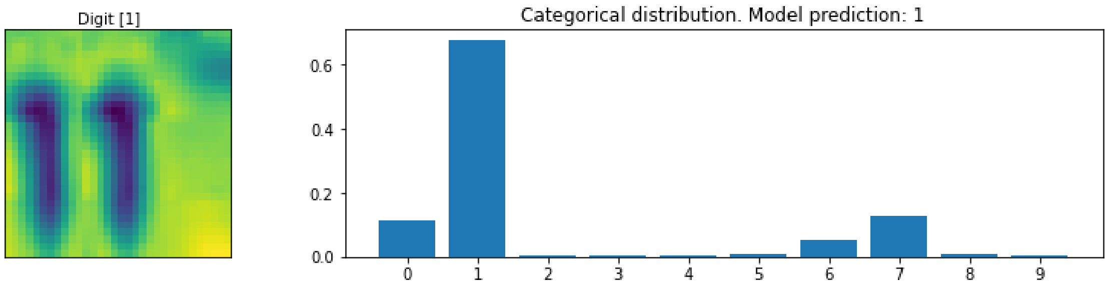

Digit [1]
Categorical distribution. Model prediction: 1
Digit [7]
Categorical distribution. Model prediction: 8
Digit [2]
Categorical distribution. Model prediction: 2
Digit [1]
Categorical distribution. Model prediction: 1
Digit [2]
Categorical distribution. Model prediction: 2

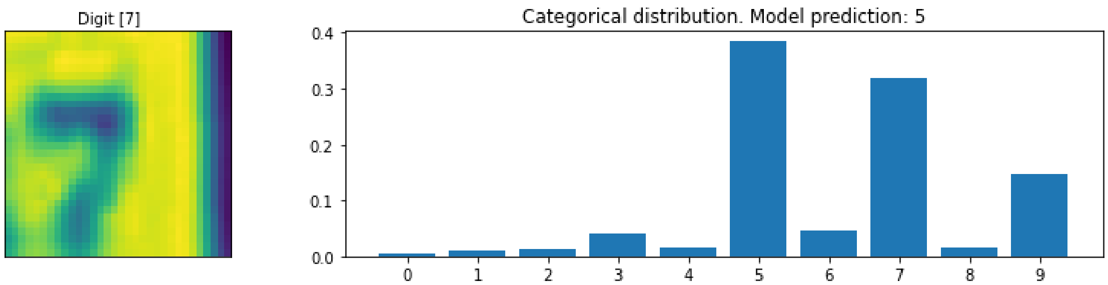
3 CNN - Predictions

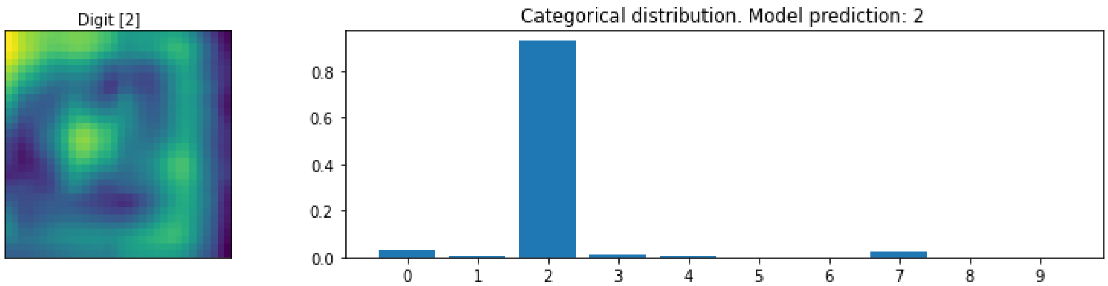
[21]:
```python
def get_model_best_epoch(model):
    checkpoint_path = 'checkpoints_best_only/checkpoint'
    model.load_weights(checkpoint_path)
    return model

model_best_epoch = get_model_best_epoch(get_new_model(0.1))

print('Model with best epoch weights:')
get_test_accuracy_CNN(model_best_epoch, x_test, y_test)
```

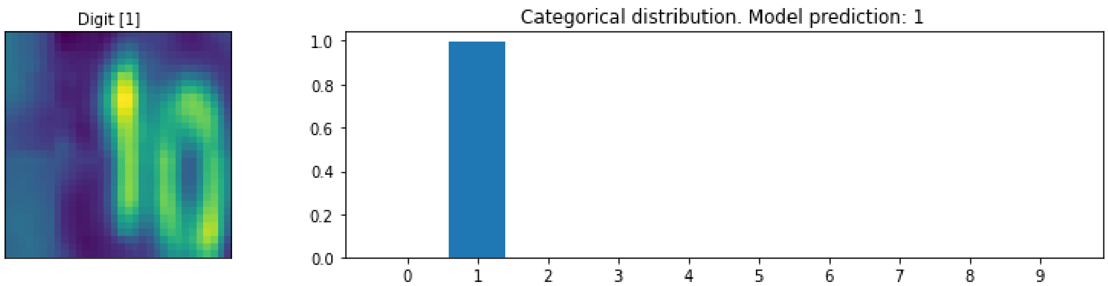
```
Model with best epoch weights:
loss: 0.836
accuracy: 0.729
```

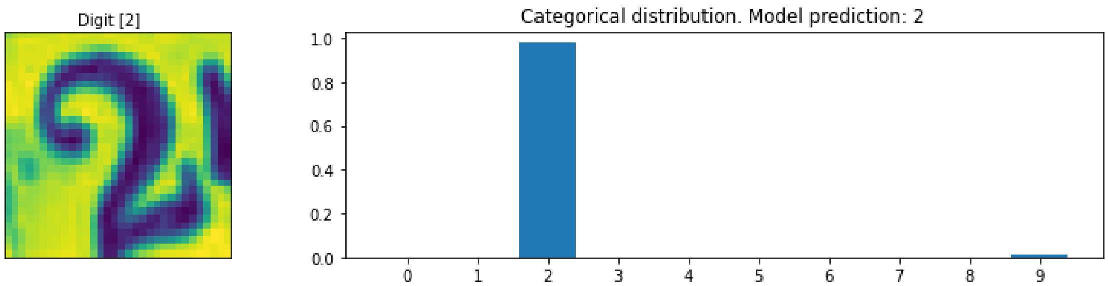
[33]:
```python
# Run this cell to get model predictions on randomly selected test images

#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_CNN.predict(random_test_images)

fig, axes = plt.subplots(5, 2, figsize=(16, 18))
fig.subplots_adjust(hspace=0.4, wspace=-0.2)

for i, (prediction, image, label) in enumerate(zip(predictions, 
 ↪random_test_images, random_test_labels)):
    axes[i, 0].imshow(np.squeeze(image))
    axes[i, 0].get_xaxis().set_visible(False)
    axes[i, 0].get_yaxis().set_visible(False)
    axes[i, 0].text(10., -1.5, f'Digit {label}')
    axes[i, 1].bar(np.arange(len(prediction)), prediction)
    axes[i, 1].set_xticks(np.arange(len(prediction)))
    axes[i, 1].set_title(f"Categorical distribution. Model prediction: {np.
 ↪argmax(prediction)}")

plt.show()
```


Digit [1]
Categorical distribution. Model prediction: 1
0.0 0.2 0.4 0.6
0 1 2 3 4 5 6 7 8 9


Digit [7]
Categorical distribution. Model prediction: 5
0.0 0.1 0.2 0.3 0.4
0 1 2 3 4 5 6 7 8 9


Digit [2]
Categorical distribution. Model prediction: 2
0.0 0.2 0.4 0.6 0.8
0 1 2 3 4 5 6 7 8 9


Digit [1]
Categorical distribution. Model prediction: 1
0.0 0.2 0.4 0.6 0.8 1.0
0 1 2 3 4 5 6 7 8 9


Digit [2]
Categorical distribution. Model prediction: 2
0.0 0.2 0.4 0.6 0.8 1.0
0 1 2 3 4 5 6 7 8 9

3.1 Conclusions:

- A lower loss was obtained in the CNN vs MLP model using fewer parameters and with fewer epochs
- Similarly, a higher accuracy was obtained in the CNN vs MLP model using fewer parameters and with fewer epochs.