Capstone Project

December 14, 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 [3]: import tensorflow as tf
        from scipy.io import loadmat
        import matplotlib.pyplot as plt
        import numpy as np
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Flatten,Dense,Conv2D, MaxPool2D
        from tensorflow.keras.layers import BatchNormalization,Dropout
        from tensorflow.keras.activations import relu,softmax
        from sklearn.model_selection import train_test_split
        import pandas as pd
        from tensorflow.keras.models import load_model
        from tensorflow.keras import activations
```

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 [4]: # 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 [5]: x_train=train["X"]
        y_train=train["y"]
        x_test=test["X"]
```

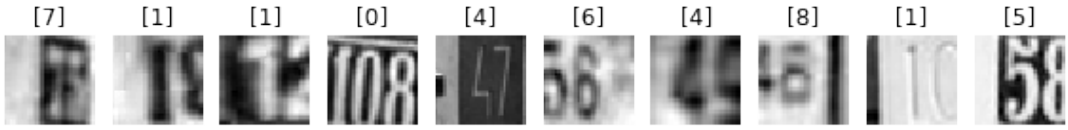
```
y_test=test["y"]
y_train=np.where(y_train!=10,y_train,y_train*0)
y_test=np.where(y_test!=10,y_test,y_test*0)
##--------------------------##
x_train=np.moveaxis(x_train,-1,0)
x_test=np.moveaxis(x_test,-1,0)
```

In [6]:
```
random_samples_idx=np.random.randint(0,x_train.shape[0],10)
```

In [7]:
```
fig, ax = plt.subplots(1, 10, figsize=(10, 1))
for i,j in enumerate(random_samples_idx):
    ax[i].set_axis_off()
    ax[i].set_title(y_train[j])
    ax[i].imshow(x_train[j])
```

In [8]:
```
x_train=x_train.mean(axis=3)
x_test=x_test.mean(axis=3)
```

In [9]:
```
random_samples_idx=np.random.randint(0,x_train.shape[0],10)
fig, ax = plt.subplots(1, 10, figsize=(10, 1))
for i,j in enumerate(random_samples_idx):
    ax[i].set_axis_off()
    ax[i].set_title(y_train[j])
    ax[i].imshow(x_train[j],cmap="gray")
```



In [10]:
```
x_train=x_train[...,np.newaxis]
x_test=x_test[...,np.newaxis]
```

1.3 2. MLP neural network classifier

- Build an MLP classifier model using the Sequential API. Your model should use only Flatten and Dense layers, with the final layer having a 10-way softmax output.
- You should design and build the model yourself. Feel free to experiment with different MLP architectures. *Hint: to achieve a reasonable accuracy you won't need to use more than 4 or 5 layers.*
- Print out the model summary (using the summary() method)
- Compile and train the model (we recommend a maximum of 30 epochs), making use of both training and validation sets during the training run.
- Your model should track at least one appropriate metric, and use at least two callbacks during training, one of which should be a ModelCheckpoint callback.
- As a guide, you should aim to achieve a final categorical cross entropy training loss of less than 1.0 (the validation loss might be higher).
- Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets.
- Compute and display the loss and accuracy of the trained model on the test set.

```
In [11]: x_test, x_validation, y_test, y_validation=train_test_split(x_test,y_test,
                                                                    test_size=0.3)
```

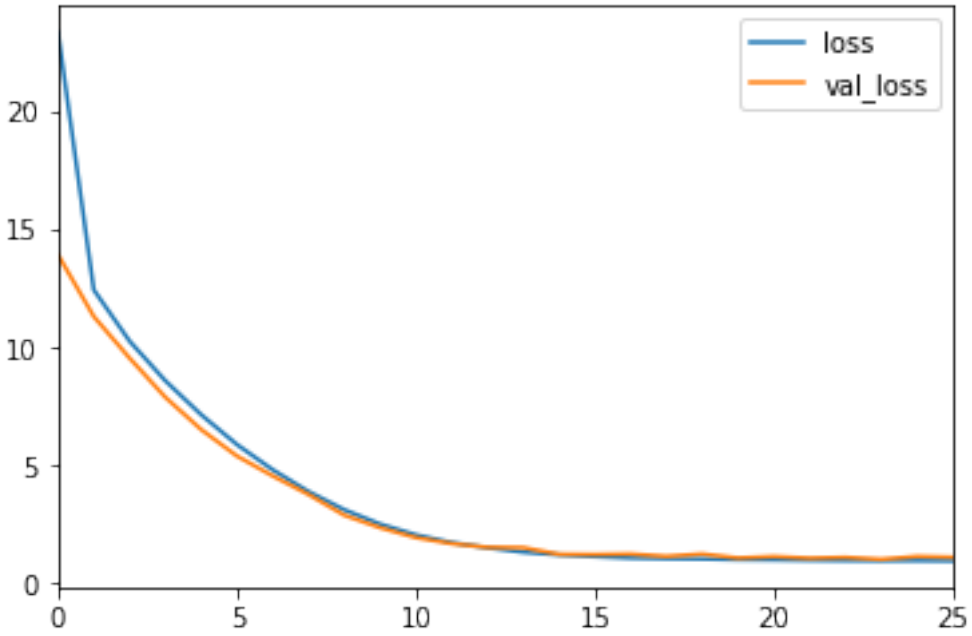
```
In [124]: model_1=Sequential()
          model_1.add(Flatten(input_shape=x_train.shape[1:]))
          model_1.add(Dense(units=1024,activation="relu",
                    kernel_regularizer=tf.keras.regularizers.l2()))
          model_1.add(Dense(units=512,activation="relu",
                    kernel_regularizer=tf.keras.regularizers.l2()))
          model_1.add(Dense(units=512,activation="relu",
                    kernel_regularizer=tf.keras.regularizers.l2()))
          model_1.add(Dense(units=10,activation="softmax",
                    kernel_regularizer=tf.keras.regularizers.l2()))
          model_1.summary()
```

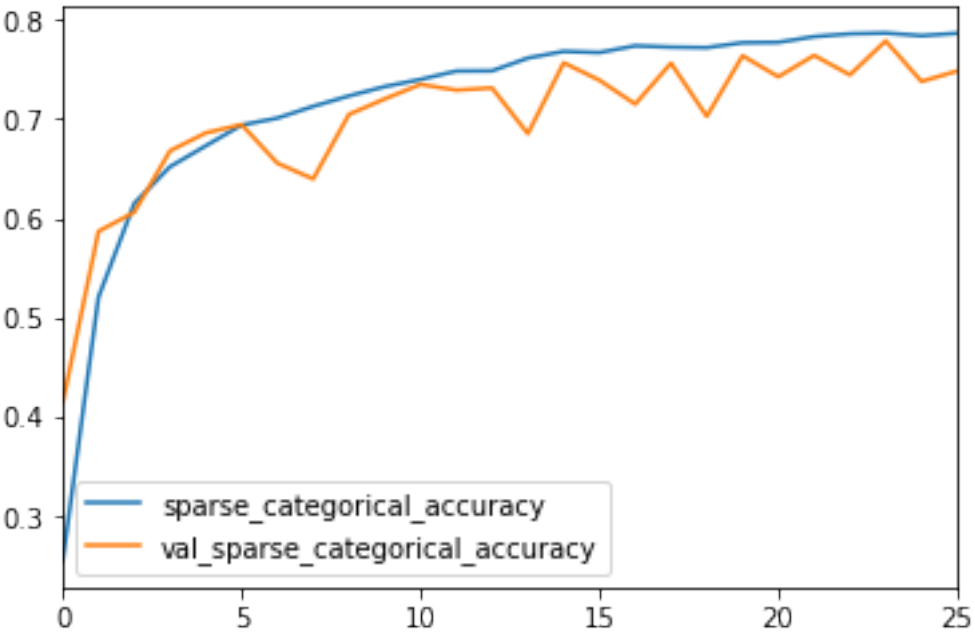
```
Model: "sequential_51"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_39 (Flatten)         (None, 1024)              0
_________________________________________________________________
dense_88 (Dense)             (None, 1024)              1049600
_________________________________________________________________
dense_89 (Dense)             (None, 512)               524800
_________________________________________________________________
dense_90 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_91 (Dense)             (None, 10)                5130
=================================================================
Total params: 1,842,186
Trainable params: 1,842,186
Non-trainable params: 0
```

```
In [125]: model_1.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                          loss="sparse_categorical_crossentropy",
                          metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])

In [126]: callback_1=tf.keras.callbacks.ModelCheckpoint(filepath="check/checkpoints",
              save_best_only=True,
              save_weights_only=False,monitor='val_sparse_categorical_accuracy',
              mode="max")

          callback_2=tf.keras.callbacks.EarlyStopping(
              monitor='val_sparse_categorical_accuracy',
              min_delta=0.1,
              patience=20)
          callbacks=[callback_1,callback_2]

In [127]: history=model_1.fit(x_train,y_train,validation_data=(x_validation,y_validation),
                              epochs=30,
                              batch_size=128,
                              callbacks=callbacks)

Train on 73257 samples, validate on 7810 samples
Epoch 1/30
73216/73257 [============================>.] - ETA: 0s - loss: 23.3800 - sparse_categorical_acc
73257/73257 [==============================] - 109s 1ms/sample - loss: 23.3748 - sparse_categor
Epoch 2/30
73216/73257 [============================>.] - ETA: 0s - loss: 12.4511 - sparse_categorical_acc
73257/73257 [==============================] - 102s 1ms/sample - loss: 12.4504 - sparse_categor
Epoch 3/30
73216/73257 [============================>.] - ETA: 0s - loss: 10.2785 - sparse_categorical_acc
73257/73257 [==============================] - 128s 2ms/sample - loss: 10.2780 - sparse_categor
Epoch 4/30
73216/73257 [============================>.] - ETA: 0s - loss: 8.5965 - sparse_categorical_accu
73257/73257 [==============================] - 103s 1ms/sample - loss: 8.5962 - sparse_categori
Epoch 5/30
73216/73257 [============================>.] - ETA: 0s - loss: 7.1584 - sparse_categorical_accu
73257/73257 [==============================] - 102s 1ms/sample - loss: 7.1578 - sparse_categori
Epoch 6/30
73216/73257 [============================>.] - ETA: 0s - loss: 5.8928 - sparse_categorical_accu
73257/73257 [==============================] - 103s 1ms/sample - loss: 5.8924 - sparse_categori
Epoch 7/30
73257/73257 [==============================] - 101s 1ms/sample - loss: 4.8236 - sparse_categori
Epoch 8/30
73257/73257 [==============================] - 102s 1ms/sample - loss: 3.8977 - sparse_categori
Epoch 9/30
73216/73257 [============================>.] - ETA: 0s - loss: 3.1272 - sparse_categorical_accu
73257/73257 [==============================] - 103s 1ms/sample - loss: 3.1269 - sparse_categori
```

```
Epoch 10/30
73216/73257 [============================>.] - ETA: 0s - loss: 2.5184 - sparse_categorical_accu
73257/73257 [==============================] - 102s 1ms/sample - loss: 2.5183 - sparse_categori
Epoch 11/30
73216/73257 [============================>.] - ETA: 0s - loss: 2.0607 - sparse_categorical_accu
73257/73257 [==============================] - 101s 1ms/sample - loss: 2.0604 - sparse_categori
Epoch 12/30
73257/73257 [==============================] - 99s 1ms/sample - loss: 1.7349 - sparse_categorio
Epoch 13/30
73257/73257 [==============================] - 99s 1ms/sample - loss: 1.5140 - sparse_categorio
Epoch 14/30
73257/73257 [==============================] - 103s 1ms/sample - loss: 1.3319 - sparse_categori
Epoch 15/30
73216/73257 [============================>.] - ETA: 0s - loss: 1.2159 - sparse_categorical_accu
73257/73257 [==============================] - 105s 1ms/sample - loss: 1.2159 - sparse_categori
Epoch 16/30
73257/73257 [==============================] - 103s 1ms/sample - loss: 1.1454 - sparse_categori
Epoch 17/30
73257/73257 [==============================] - 103s 1ms/sample - loss: 1.0825 - sparse_categori
Epoch 18/30
73257/73257 [==============================] - 103s 1ms/sample - loss: 1.0588 - sparse_categori
Epoch 19/30
73257/73257 [==============================] - 105s 1ms/sample - loss: 1.0436 - sparse_categori
Epoch 20/30
73216/73257 [============================>.] - ETA: 0s - loss: 1.0144 - sparse_categorical_accu
73257/73257 [==============================] - 109s 1ms/sample - loss: 1.0144 - sparse_categori
Epoch 21/30
73257/73257 [==============================] - 106s 1ms/sample - loss: 1.0069 - sparse_categori
Epoch 22/30
73216/73257 [============================>.] - ETA: 0s - loss: 0.9883 - sparse_categorical_accu
73257/73257 [==============================] - 106s 1ms/sample - loss: 0.9884 - sparse_categori
Epoch 23/30
73257/73257 [==============================] - 104s 1ms/sample - loss: 0.9759 - sparse_categori
Epoch 24/30
73216/73257 [============================>.] - ETA: 0s - loss: 0.9693 - sparse_categorical_accu
73257/73257 [==============================] - 106s 1ms/sample - loss: 0.9693 - sparse_categori
Epoch 25/30
73257/73257 [==============================] - 105s 1ms/sample - loss: 0.9708 - sparse_categori
Epoch 26/30
73257/73257 [==============================] - 105s 1ms/sample - loss: 0.9616 - sparse_categori
```

```
In [128]: df = pd.DataFrame(history.history)
          df.plot(y=['loss', 'val_loss'])
          df.plot(y=['sparse_categorical_accuracy', 'val_sparse_categorical_accuracy'])

Out[128]: <matplotlib.axes._subplots.AxesSubplot at 0x7f9b5f1fb4e0>
```






```
In [129]: model_1.evaluate(x_test,y_test,batch_size=32,verbose=2)

18222/1 - 15s - loss: 1.1588 - sparse_categorical_accuracy: 0.7374
```

```
Out[129]: [1.137349641515426, 0.73735046]
```

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 [141]: model_2=Sequential()
          model_2.add(Conv2D(filters=32,kernel_size=(3,3),padding='same',
                             activation="relu",input_shape=(x_train.shape[1:])))
          model_2.add(MaxPool2D(pool_size=(3,3),padding='valid'))
          model_2.add(Conv2D(filters=128,kernel_size=(3,3),padding='valid',
                             activation="relu",input_shape=(x_train.shape[1:])))
          model_2.add(MaxPool2D(pool_size=(3, 3),padding='same'))
          model_2.add(Conv2D(filters=256,kernel_size=(3,3),padding='valid',
                             activation="relu",input_shape=(x_train.shape[1:])))
          model_2.add(MaxPool2D(pool_size=(3, 3),padding='same'))
          model_2.add(BatchNormalization())
          model_2.add(Flatten())
          model_2.add(Dense(units=64,activation="relu",
                    kernel_regularizer=tf.keras.regularizers.l2()))
          model_2.add(Dropout(0.2))
          model_2.add(Dense(units=10,activation="softmax",
                    kernel_regularizer=tf.keras.regularizers.l2()))
          model_2.summary()
```

```
Model: "sequential_53"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_110 (Conv2D)          (None, 32, 32, 32)        320
_________________________________________________________________
max_pooling2d_73 (MaxPooling (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_111 (Conv2D)          (None, 8, 8, 128)         36992
```

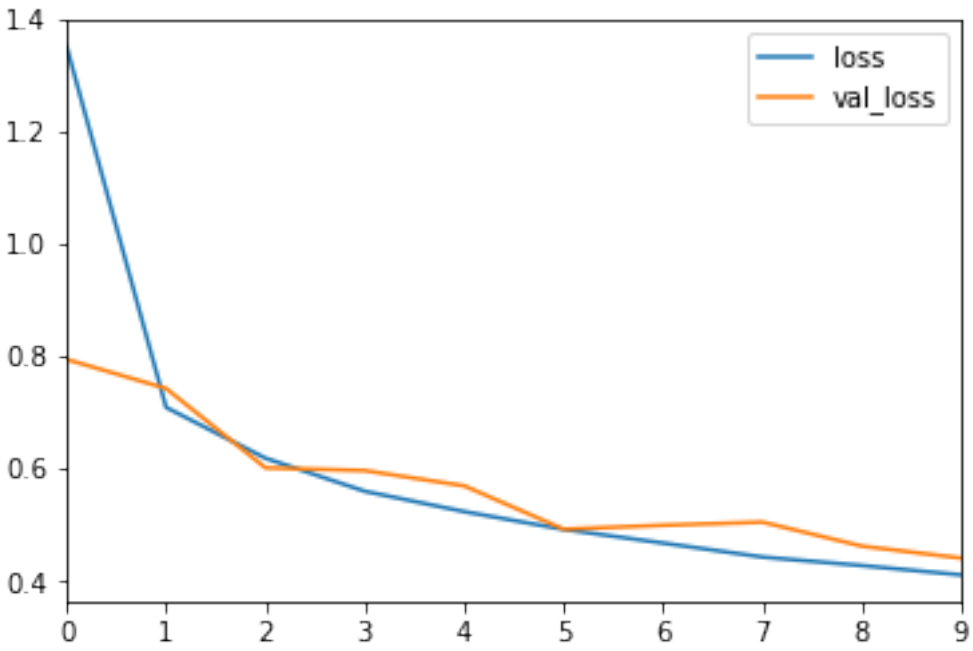
```
_________________________________________________________________
max_pooling2d_74 (MaxPooling (None, 3, 3, 128)         0
_________________________________________________________________
conv2d_112 (Conv2D)          (None, 1, 1, 256)         295168
_________________________________________________________________
max_pooling2d_75 (MaxPooling (None, 1, 1, 256)         0
_________________________________________________________________
batch_normalization_68 (Batc (None, 1, 1, 256)         1024
_________________________________________________________________
flatten_41 (Flatten)         (None, 256)               0
_________________________________________________________________
dense_94 (Dense)             (None, 64)                16448
_________________________________________________________________
dropout_67 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_95 (Dense)             (None, 10)                650
=================================================================
Total params: 350,602
Trainable params: 350,090
Non-trainable params: 512
_________________________________________________________________
```

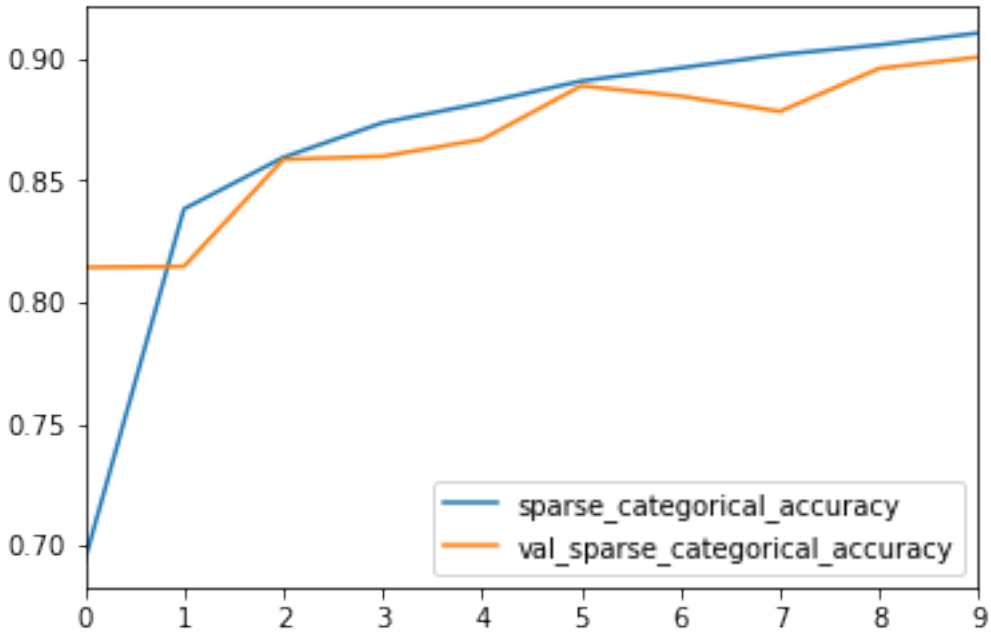
```
In [142]: model_2.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0007), #or 0.001
                          loss="sparse_categorical_crossentropy",
                          metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])

In [143]: callback_1=tf.keras.callbacks.ModelCheckpoint(filepath="check_2/checkpoints",
              save_best_only=True,
              save_weights_only=False,monitor='val_sparse_categorical_accuracy',mode="max")

          callback_2=tf.keras.callbacks.EarlyStopping(
              monitor='val_sparse_categorical_accuracy',
              min_delta=0.1,
              patience=20)
          callbacks=[callback_1,callback_2]

In [144]: history_2=model_2.fit(x_train,y_train,validation_data=(x_validation,y_validation),
                                epochs=10,batch_size=64, #epoch=10
                                callbacks=callbacks)

Train on 73257 samples, validate on 7810 samples
Epoch 1/10
73216/73257 [============================>.] - ETA: 0s - loss: 1.3516 - sparse_categorical_accu
73257/73257 [==============================] - 316s 4ms/sample - loss: 1.3515 - sparse_categori
Epoch 2/10
73216/73257 [============================>.] - ETA: 0s - loss: 0.7097 - sparse_categorical_accu
73257/73257 [==============================] - 309s 4ms/sample - loss: 0.7098 - sparse_categori
Epoch 3/10
```

```
73216/73257 [============================>.] - ETA: 0s - loss: 0.6192 - sparse_categorical_accu
73257/73257 [==============================] - 304s 4ms/sample - loss: 0.6192 - sparse_categori
Epoch 4/10
73216/73257 [============================>.] - ETA: 0s - loss: 0.5605 - sparse_categorical_accu
73257/73257 [==============================] - 336s 5ms/sample - loss: 0.5604 - sparse_categori
Epoch 5/10
73216/73257 [============================>.] - ETA: 0s - loss: 0.5241 - sparse_categorical_accu
73257/73257 [==============================] - 311s 4ms/sample - loss: 0.5242 - sparse_categori
Epoch 6/10
73216/73257 [============================>.] - ETA: 0s - loss: 0.4927 - sparse_categorical_accu
73257/73257 [==============================] - 308s 4ms/sample - loss: 0.4926 - sparse_categori
Epoch 7/10
73257/73257 [==============================] - 302s 4ms/sample - loss: 0.4682 - sparse_categori
Epoch 8/10
73257/73257 [==============================] - 303s 4ms/sample - loss: 0.4434 - sparse_categori
Epoch 9/10
73216/73257 [============================>.] - ETA: 0s - loss: 0.4282 - sparse_categorical_accu
73257/73257 [==============================] - 304s 4ms/sample - loss: 0.4282 - sparse_categori
Epoch 10/10
73216/73257 [============================>.] - ETA: 0s - loss: 0.4116 - sparse_categorical_accu
73257/73257 [==============================] - 303s 4ms/sample - loss: 0.4116 - sparse_categori
```

```
In [145]: df = pd.DataFrame(history_2.history)
          df.plot(y=['loss', 'val_loss'])
          df.plot(y=['sparse_categorical_accuracy', 'val_sparse_categorical_accuracy'])
```

```
Out[145]: <matplotlib.axes._subplots.AxesSubplot at 0x7f9b4519ad68>
```





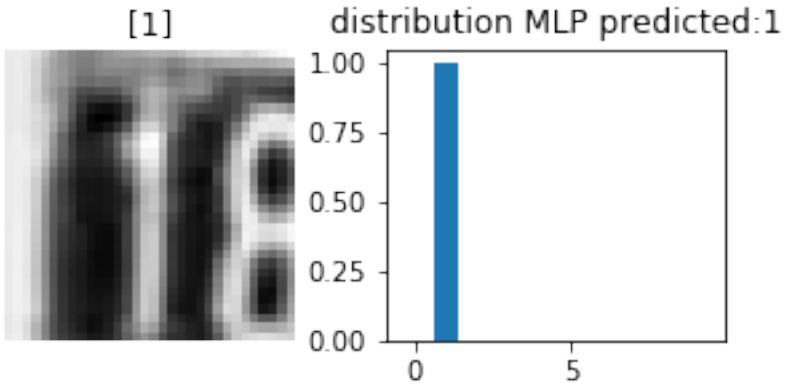

```
In [146]: model_2.evaluate(x_test,y_test,batch_size=128,verbose=2)

18222/1 - 20s - loss: 0.3960 - sparse_categorical_accuracy: 0.8953


Out[146]: [0.45171643710989495, 0.8953463]
```

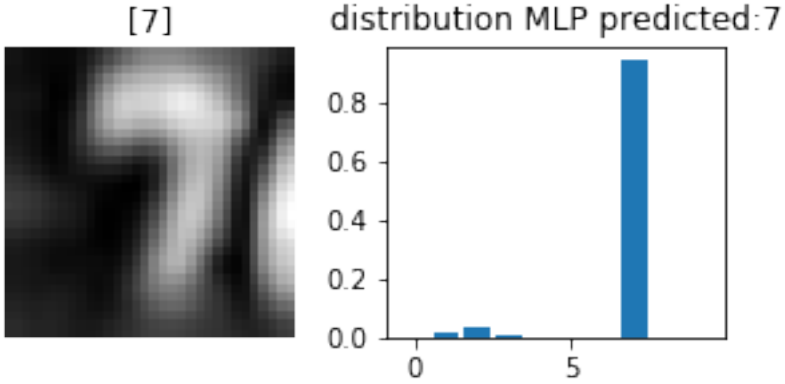
1.5 4. Get model predictions

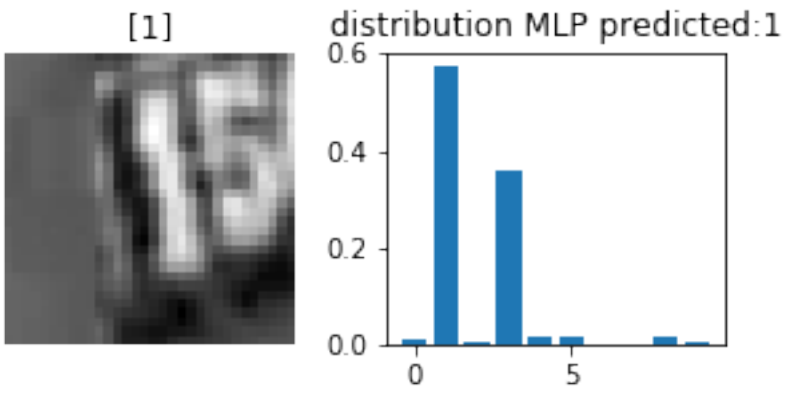
- Load the best weights for the MLP and CNN models that you saved during the training run.
- Randomly select 5 images and corresponding labels from the test set and display the images with their labels.
- Alongside the image and label, show each model's predictive distribution as a bar chart, and the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

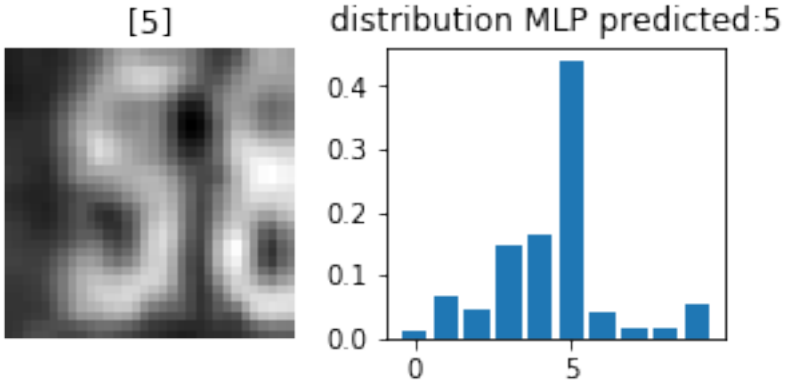
```
In [12]: model_1=load_model("check/checkpoints")
         model_2=load_model("check_2/checkpoints")

In [13]: random_samples_idx=np.random.randint(0,x_test.shape[0],5)

In [14]: ##prediction model mlp
         %matplotlib inline
         for i in range(5):
                 fig, ax = plt.subplots(1,2,figsize=(5,2))
```

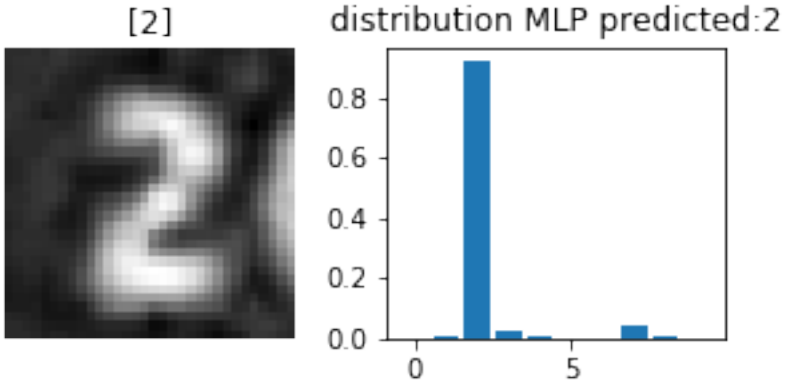
```
s=random_samples_idx[i]
passive=x_test[s]
passive_1=passive.reshape(passive.shape[0],passive.shape[1])
passive=passive[np.newaxis,...]
passive_2=model_1.predict(passive)
res=np.argmax(passive_2)
ax[0].set_axis_off()
ax[0].set_title(y_test[s])
ax[0].imshow(passive_1,cmap="gray")
ax[1].bar(x=list(range(passive_2.shape[1])),
          height=np.squeeze(passive_2).tolist())
ax[1].set_title("distribution MLP predicted:"+str(res))
plt.show()
```





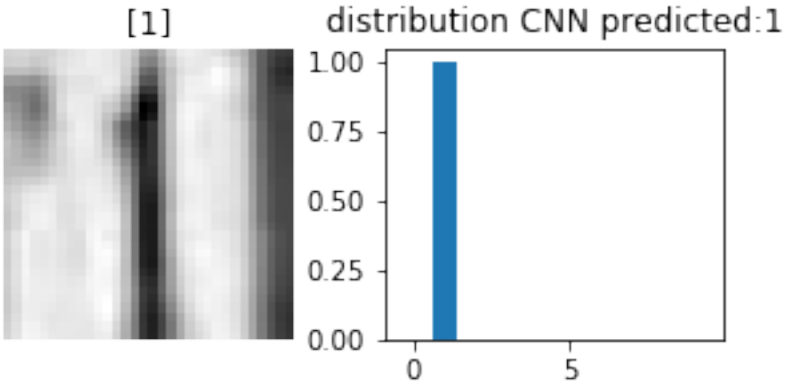


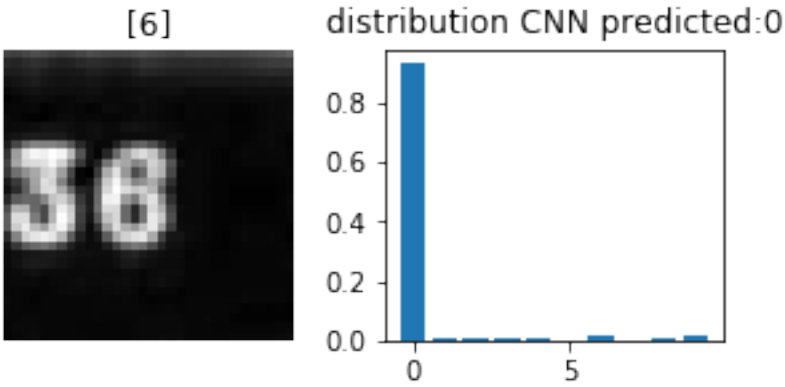


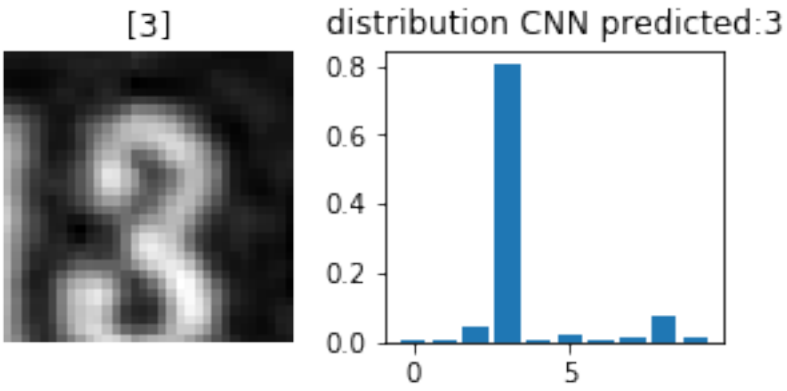


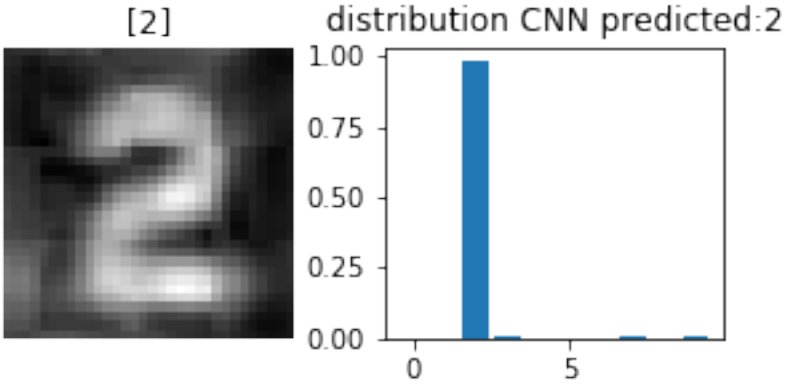

```
In [15]: random_samples_idx=np.random.randint(0,x_test.shape[0],5)
```

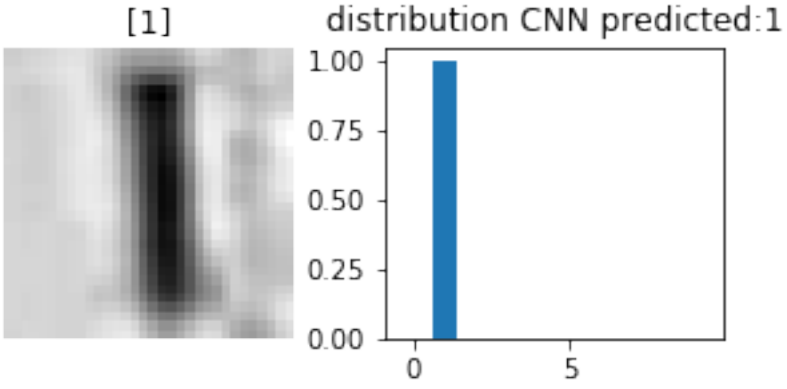
In [16]:
```
##prediction for CNN
for i in range(5):
        fig, ax = plt.subplots(1,2,figsize=(5,2))
        s=random_samples_idx[i]
        passive=x_test[s]
        passive_1=passive.reshape(passive.shape[0],passive.shape[1])
        passive=passive[np.newaxis,...]
        passive_2=model_2.predict(passive)
        res=np.argmax(passive_2)
        ax[0].set_axis_off()
        ax[0].set_title(y_test[s])
        ax[0].imshow(passive_1,cmap="gray")
        ax[1].bar(x=list(range(passive_2.shape[1])),
                  height=np.squeeze(passive_2).tolist())
        ax[1].set_title("distribution CNN predicted:"+str(res))
```






[3]
distribution CNN predicted:3
0.8
0.6
0.4
0.2
0.0
0
5


[2]
distribution CNN predicted:2
1.00
0.75
0.50
0.25
0.00
0
5


[1]
distribution CNN predicted:1
1.00
0.75
0.50
0.25
0.00
0
5