Capstone Project

April 16, 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 [1]: import tensorflow as tf
        from scipy.io import loadmat
        import numpy as np
        import matplotlib.pyplot as plt
```

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

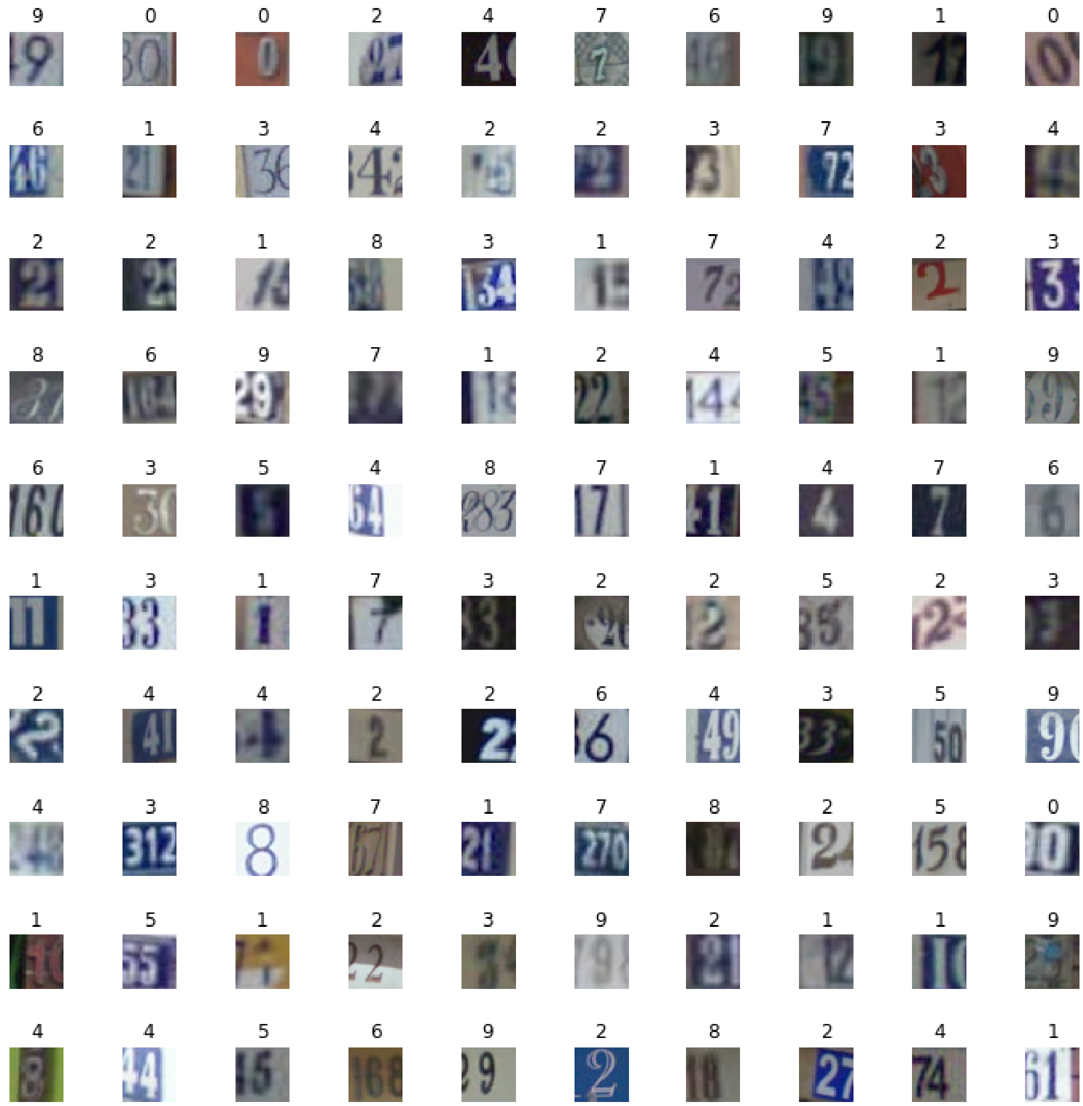
```
x_train = train['X']
y_train = train['y']
x_test = test['X']
y_test = test['y']
```

In [4]:
```
# The loaded format seems to be:
# - X shape is (H, W, C, num_examples). 32x32 RGB images. uint8 in range [0..255].
# - y shape is (num_examples, 1). uint8 in range[1..10].
#   Numbers 1..9 label the digits 1..9; number 10 labels the digit 0.

# Here we extract those shapes and assert our expectations about the dataset.

assert x_train.dtype == 'uint8'
assert y_train.dtype == 'uint8'
assert x_test.dtype == 'uint8'
assert y_test.dtype == 'uint8'

assert len(x_train.shape) == 4
assert len(y_train.shape) == 2
assert len(x_test.shape) == 4
assert len(y_test.shape) == 2

num_train_examples = x_train.shape[3]
num_test_examples = x_test.shape[3]

assert y_train.shape[0] == num_train_examples
assert y_test.shape[0] == num_test_examples

assert x_train.shape[0:3] == (32, 32, 3)
assert x_test.shape[0:3] == (32, 32, 3)

assert np.min(x_train) == 0
assert np.max(x_train) == 255
assert np.min(x_test) == 0
assert np.max(x_test) == 255

assert np.min(y_train) == 1
assert np.max(y_train) == 10
assert np.min(y_test) == 1
assert np.max(y_test) == 10
```

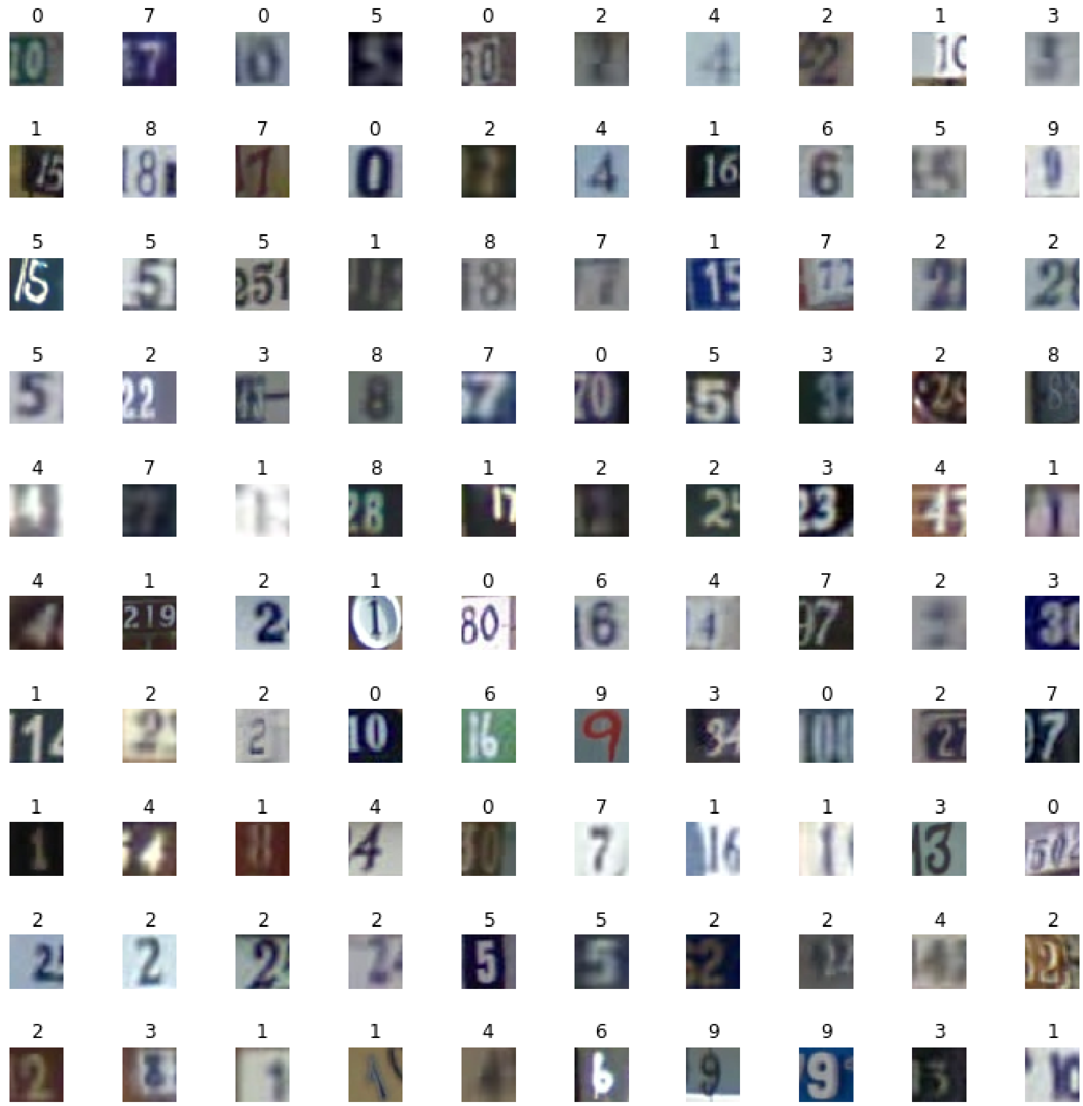
In [5]:
```
# The given formats are not ideal. Convert to formats that work nicely with TensorFlow

# Convert to expected (num_examples, H, W, C) format.
x_train = np.transpose(x_train, axes=[3, 0, 1, 2])
x_test = np.transpose(x_test, axes=[3, 0, 1, 2])

assert x_train.shape == (num_train_examples, 32, 32, 3)
```

```python
assert x_test.shape == (num_test_examples, 32, 32, 3)

# Convert to floats in range 0..1.
x_train = x_train / 255.
x_test = x_test / 255.

# Labels have unnecessary extra dimension; remove it.
y_train = np.reshape(y_train, (num_train_examples,))
y_test = np.reshape(y_test, (num_test_examples,))

assert y_train.shape == (num_train_examples,)
assert y_test.shape == (num_test_examples,)

# The label range [1..10] doesn't play nicely with sparse_categorical_crossentropy,
# which expects 10 labels in the range [0..9].
# Fix by converting label '10' to label '0' (which makes more sense anyway).
y_train[y_train == 10] = 0
y_test[y_test == 10] = 0
```

In [6]:
```python
# Select a random sample of images and corresponding labels from the dataset (at least
# and display them in a figure.

# (Here I show 100 images, to make sure there are several examples of each label.)

def show_sample_rgb(x, y):
    side_length = 10
    random_example_indexes = np.random.choice(x.shape[0], size=side_length**2)
    random_x_subset = x[random_example_indexes]
    random_y_subset = y[random_example_indexes]
    fig, ax = plt.subplots(side_length, side_length, figsize=(side_length, side_length)
    fig.tight_layout()
    for i in range(side_length):
        for j in range(side_length):
            example_index = i*10+j
            ax[i,j].set_axis_off()
            ax[i,j].imshow(random_x_subset[example_index])
            ax[i,j].set_title(random_y_subset[example_index])

# First show some examples from the training data
show_sample_rgb(x_train, y_train)
```



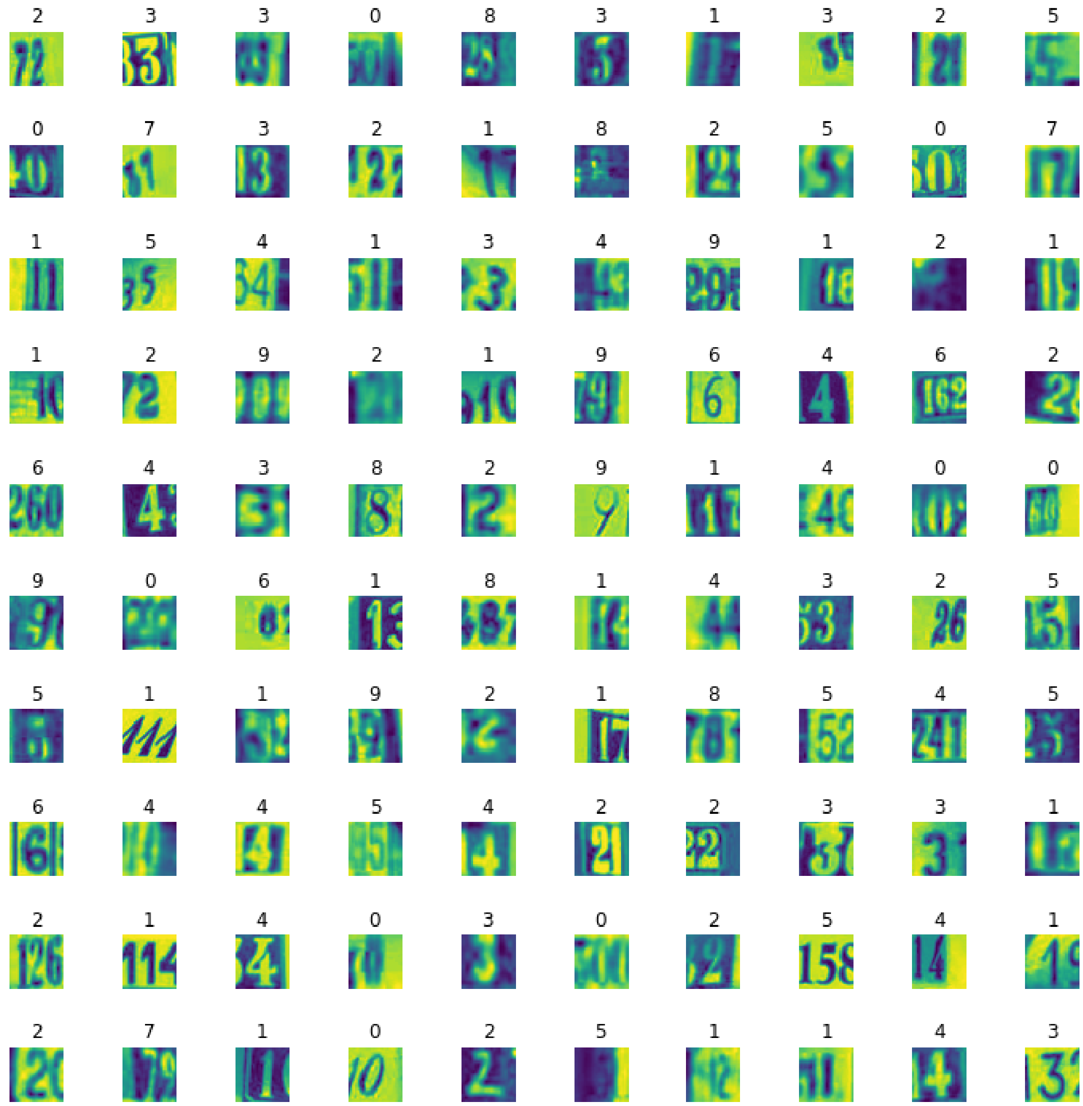
In [7]:
```
# Now some examples from the test data, to check we've transformed it in the same way
show_sample_rgb(x_test, y_test)
```



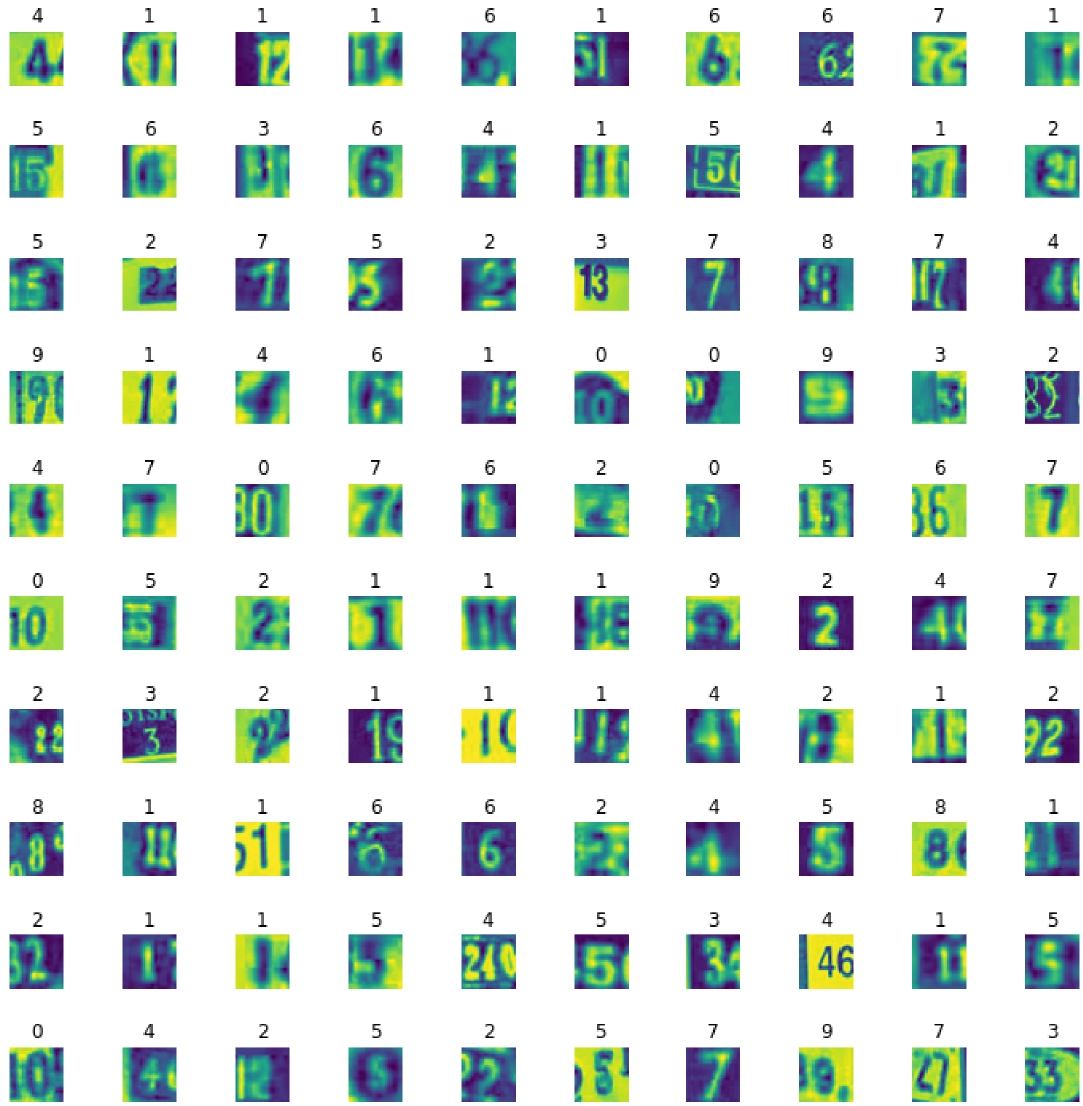
In [8]:
```
# Convert the training and test images to grayscale
# by taking the average across all colour channels for each pixel.
# Hint: retain the channel dimension, which will now have size 1.

# Note we could use `tf.image.rgb_to_grayscale` here, but instead just use numpy.
x_train = np.mean(x_train, axis=3, keepdims=True)
x_test = np.mean(x_test, axis=3, keepdims=True)

assert x_train.shape == (num_train_examples, 32, 32, 1)
assert x_test.shape == (num_test_examples, 32, 32, 1)
```

In [9]:
```
# Select a random sample of the grayscale images and corresponding labels from the dat
# and display them in a figure.
```

```
# Note: imshow() doesn't like the [H, W, 1] format, so we must remove the channel axis
# Also note: imshow() displays it with a colormap despite being greyscale :-)

def show_sample_greyscale(x, y):
    side_length = 10
    random_example_indexes = np.random.choice(x.shape[0], size=side_length**2)
    random_x_subset = x[random_example_indexes]
    random_y_subset = y[random_example_indexes]
    fig, ax = plt.subplots(side_length, side_length, figsize=(side_length, side_length
    fig.tight_layout()
    for i in range(side_length):
        for j in range(side_length):
            example_index = i*10+j
            ax[i,j].set_axis_off()
            ax[i,j].imshow(random_x_subset[example_index][:, :, 0])
            ax[i,j].set_title(random_y_subset[example_index])

# First show some training examples
show_sample_greyscale(x_train, y_train)
```



```
In [10]: # Also show the test set, to check we've prepared it in the same way
         show_sample_greyscale(x_test, y_test)
```



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]:
```
# 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 layer

mlp_model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(32, 32, 1)),  # Immediately flatten. Dense l
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),  # Number of units per dense layer
    tf.keras.layers.Dense(128, activation='relu'),  # These numbers give reasonable r
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax'),
])
```

In [12]:
```
# Print out the model summary (using the summary() method)
mlp_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 256)               262400
_________________________________________________________________
dense_1 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
Total params: 329,610
Trainable params: 329,610
Non-trainable params: 0
_________________________________________________________________
```

In [13]:
```
mlp_model.compile(
    optimizer='adam',
```

```
        loss='sparse_categorical_crossentropy',
        # Your model should track at least one appropriate metric
        metrics=['accuracy'],
    )
```

In [14]:
```
# Compile and train the model (we recommend a maximum of 30 epochs),
# making use of both training and validation sets during the training run.

mlp_model_path = 'mlp_checkpoints/checkpoint'

mlp_history = mlp_model.fit(
    x_train, y_train,
    epochs=30,
    batch_size=256,
    # making use of both training and validation sets during the training run
    validation_split=0.2,
    # use at least two callbacks during training, one of which should be a ModelCheck
    callbacks=[
        tf.keras.callbacks.ModelCheckpoint(
            mlp_model_path, save_freq='epoch',
            save_best_only=True, monitor='val_accuracy'  # Final task asks to load th
        ),
        # Stop if accuracy stops improving. (Note this doesn't seem to happen within
        tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=3),
    ],
)
```

```
Train on 58605 samples, validate on 14652 samples
Epoch 1/30
58368/58605 [============================>.] - ETA: 0s - loss: 2.0959 - accuracy: 0.2458WARNIN
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
INFO:tensorflow:Assets written to: mlp_checkpoints/checkpoint/assets
58605/58605 [==============================] - 23s 386us/sample - loss: 2.0943 - accuracy: 0.2
Epoch 2/30
58368/58605 [============================>.] - ETA: 0s - loss: 1.5388 - accuracy: 0.4710INFO:t
58605/58605 [==============================] - 20s 343us/sample - loss: 1.5384 - accuracy: 0.4
Epoch 3/30
58368/58605 [============================>.] - ETA: 0s - loss: 1.2923 - accuracy: 0.5768INFO:t
58605/58605 [==============================] - 20s 347us/sample - loss: 1.2919 - accuracy: 0.5
Epoch 4/30
58368/58605 [============================>.] - ETA: 0s - loss: 1.1386 - accuracy: 0.6381INFO:t
58605/58605 [==============================] - 20s 342us/sample - loss: 1.1381 - accuracy: 0.6
Epoch 5/30
58112/58605 [============================>.] - ETA: 0s - loss: 1.0544 - accuracy: 0.6664INFO:t
58605/58605 [==============================] - 21s 363us/sample - loss: 1.0547 - accuracy: 0.6
Epoch 6/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.9881 - accuracy: 0.6921INFO:t
```

```
58605/58605 [==============================] - 21s 359us/sample - loss: 0.9876 - accuracy: 0.69
Epoch 7/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.9265 - accuracy: 0.7126INFO:te
58605/58605 [==============================] - 21s 355us/sample - loss: 0.9269 - accuracy: 0.71
Epoch 8/30
58112/58605 [============================>.] - ETA: 0s - loss: 0.8792 - accuracy: 0.7266INFO:te
58605/58605 [==============================] - 21s 351us/sample - loss: 0.8796 - accuracy: 0.72
Epoch 9/30
58605/58605 [==============================] - 19s 330us/sample - loss: 0.8589 - accuracy: 0.73
Epoch 10/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.8147 - accuracy: 0.7450INFO:te
58605/58605 [==============================] - 21s 360us/sample - loss: 0.8149 - accuracy: 0.74
Epoch 11/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.7905 - accuracy: 0.7522INFO:te
58605/58605 [==============================] - 20s 350us/sample - loss: 0.7903 - accuracy: 0.75
Epoch 12/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.7629 - accuracy: 0.7614INFO:te
58605/58605 [==============================] - 21s 354us/sample - loss: 0.7630 - accuracy: 0.76
Epoch 13/30
58605/58605 [==============================] - 19s 327us/sample - loss: 0.7441 - accuracy: 0.76
Epoch 14/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.7194 - accuracy: 0.7741INFO:te
58605/58605 [==============================] - 21s 351us/sample - loss: 0.7191 - accuracy: 0.77
Epoch 15/30
58605/58605 [==============================] - 19s 326us/sample - loss: 0.7057 - accuracy: 0.77
Epoch 16/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.7021 - accuracy: 0.7804INFO:te
58605/58605 [==============================] - 21s 352us/sample - loss: 0.7020 - accuracy: 0.78
Epoch 17/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.6732 - accuracy: 0.7883INFO:te
58605/58605 [==============================] - 21s 355us/sample - loss: 0.6736 - accuracy: 0.78
Epoch 18/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.6655 - accuracy: 0.7898INFO:te
58605/58605 [==============================] - 21s 354us/sample - loss: 0.6653 - accuracy: 0.78
Epoch 19/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.6436 - accuracy: 0.7982INFO:te
58605/58605 [==============================] - 21s 355us/sample - loss: 0.6431 - accuracy: 0.79
Epoch 20/30
58605/58605 [==============================] - 19s 320us/sample - loss: 0.6329 - accuracy: 0.80
Epoch 21/30
58605/58605 [==============================] - 19s 319us/sample - loss: 0.6253 - accuracy: 0.80
Epoch 22/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.6217 - accuracy: 0.8064INFO:te
58605/58605 [==============================] - 20s 348us/sample - loss: 0.6215 - accuracy: 0.80
Epoch 23/30
58605/58605 [==============================] - 19s 322us/sample - loss: 0.5903 - accuracy: 0.81
Epoch 24/30
58368/58605 [============================>.] - ETA: 0s - loss: 0.5960 - accuracy: 0.8124INFO:te
```

```
58605/58605 [==============================] - 20s 344us/sample - loss: 0.5962 - accuracy: 0.8
Epoch 25/30
58605/58605 [==============================] - 19s 319us/sample - loss: 0.5795 - accuracy: 0.8
Epoch 26/30
58605/58605 [==============================] - 19s 323us/sample - loss: 0.5734 - accuracy: 0.8
Epoch 27/30
58605/58605 [==============================] - 19s 321us/sample - loss: 0.5678 - accuracy: 0.8
```

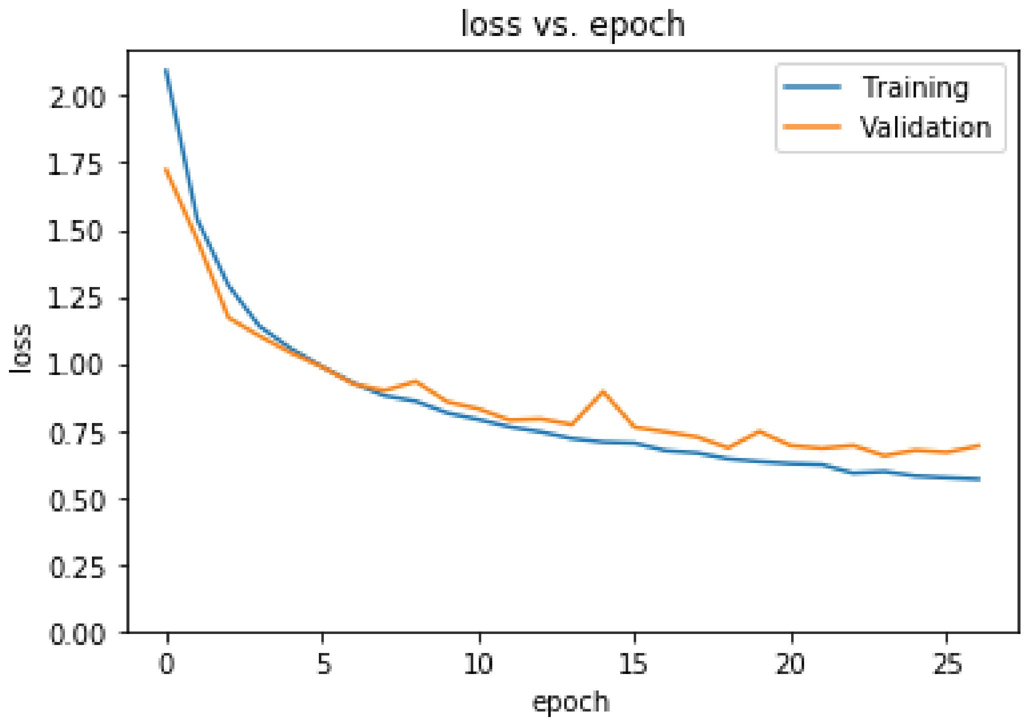
In [15]:
```python
# Confirm that we've saved a checkpoint during training
!ls mlp_checkpoints/checkpoint
```

```
assets         saved_model.pb         variables
```

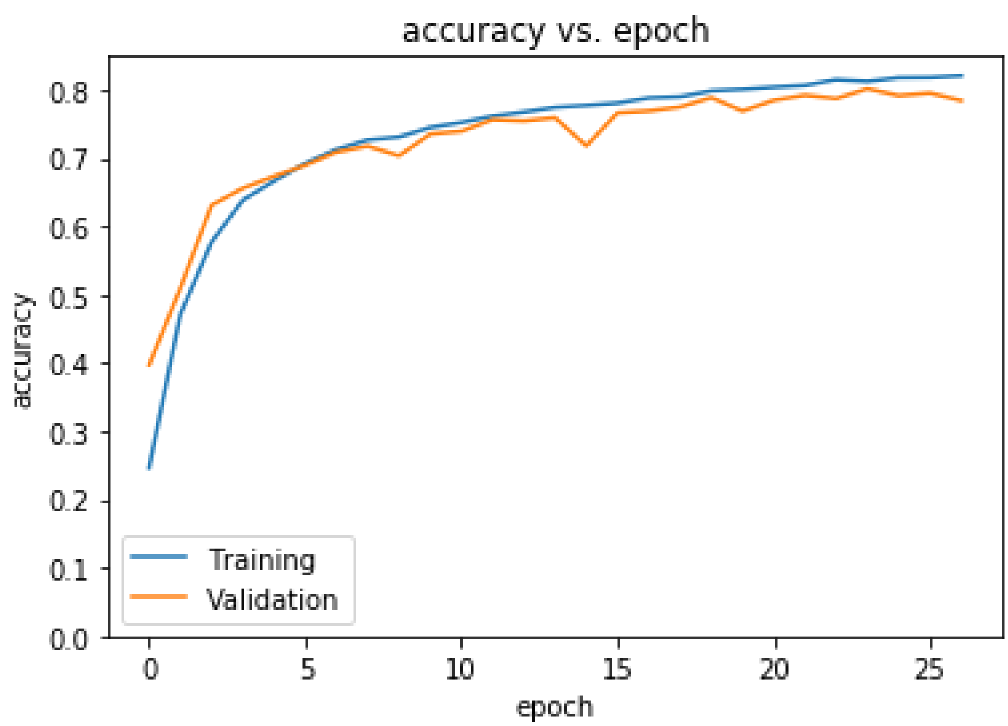
In [16]:
```python
# As a guide, you should aim to achieve
# a final categorical cross entropy training loss of less than 1.0
# (the validation loss might be higher).

mlp_training_loss = mlp_history.history['loss'][-1]
print(mlp_training_loss)
assert mlp_training_loss < 1.0
```

```
0.567750245413574
```

In [30]:
```python
# Plot the learning curves for loss vs epoch and accuracy vs epoch for both training 
def plot_curves(history, metric_name):
    plt.plot(history.history[metric_name])
    plt.plot(history.history[f"val_{metric_name}"])
    plt.title(f"{metric_name} vs. epoch")
    plt.ylabel(metric_name)
    plt.ylim(ymin=0)
    plt.xlabel('epoch')
    plt.legend(['Training', 'Validation'])
    plt.show()

plot_curves(mlp_history, 'loss')
```




```
In [31]: plot_curves(mlp_history, 'accuracy')
```



```
In [19]: # Compute and display the loss and accuracy of the trained model on the test set.

         mlp_test_loss, mlp_test_accuracy = mlp_model.evaluate(x_test, y_test, verbose=2)
         print(mlp_test_loss)
         print(mlp_test_accuracy)
```

```
26032/1 - 5s - loss: 0.7230 - accuracy: 0.7704
0.7545684045564198
0.77039796
```

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 [20]: cnn_model = tf.keras.Sequential([
             tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu', input_shape=
             tf.keras.layers.BatchNormalization(),
             tf.keras.layers.MaxPool2D(),

             tf.keras.layers.Dropout(0.2),
             tf.keras.layers.Conv2D(filters=64, kernel_size=3, activation='relu'),
             tf.keras.layers.BatchNormalization(),
             tf.keras.layers.MaxPool2D(),

             tf.keras.layers.Dropout(0.2),
             tf.keras.layers.Conv2D(filters=64, kernel_size=3, activation='relu'),
             tf.keras.layers.BatchNormalization(),
             tf.keras.layers.MaxPool2D(),

             tf.keras.layers.Flatten(),  # Flatten just before Dense layers
```

```
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(128, activation='relu'),

        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax'),
    ])
```

In [21]:
```
cnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        320
_________________________________________________________________
batch_normalization (BatchNo (None, 30, 30, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 13, 13, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
batch_normalization_2 (Batch (None, 4, 4, 64)          256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_5 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                1290
=================================================================
Total params: 90,570
Trainable params: 90,250
Non-trainable params: 320
```

```
In [22]: # The CNN model should use fewer trainable parameters than your MLP model.
         print(mlp_model.count_params())
         print(cnn_model.count_params())
         assert cnn_model.count_params() < mlp_model.count_params()
```

```
329610
90570
```

```
In [23]: # Use the same compilation settings as for MLP.
         cnn_model.compile(
             optimizer='adam',
             loss='sparse_categorical_crossentropy',
             # Your model should track at least one appropriate metric
             metrics=['accuracy'],
         )
```

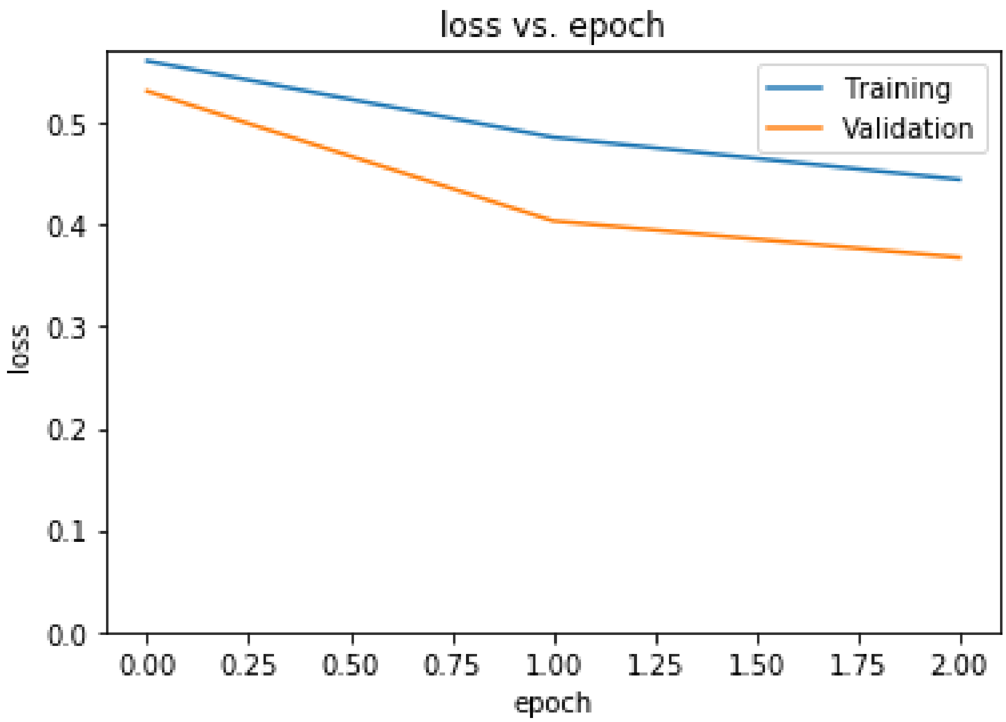
```
In [25]: # Use the same fit() settings as for MLP.
         cnn_model_path = 'cnn_checkpoints/checkpoint'

         cnn_history = cnn_model.fit(
             x_train, y_train,
             epochs=3,  # CNN model seems to be much slower to train, but fewer epochs seem su
             batch_size=256,
             # making use of both training and validation sets during the training run
             validation_split=0.2,
             # use at least two callbacks during training, one of which should be a ModelCheck
             callbacks=[
                 tf.keras.callbacks.ModelCheckpoint(
                     cnn_model_path, save_freq='epoch',
                     save_best_only=True, monitor='val_accuracy'  # Final task asks to load th
                 ),
                 tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=3),
             ],
         )
```

```
Train on 58605 samples, validate on 14652 samples
Epoch 1/3
58368/58605 [============================>.] - ETA: 1s - loss: 0.5603 - accuracy: 0.8272INFO:te
58605/58605 [==============================] - 520s 9ms/sample - loss: 0.5604 - accuracy: 0.827
Epoch 2/3
58368/58605 [============================>.] - ETA: 1s - loss: 0.4854 - accuracy: 0.8501INFO:te
58605/58605 [==============================] - 479s 8ms/sample - loss: 0.4856 - accuracy: 0.850
Epoch 3/3
58368/58605 [============================>.] - ETA: 1s - loss: 0.4437 - accuracy: 0.8641INFO:te
58605/58605 [==============================] - 488s 8ms/sample - loss: 0.4441 - accuracy: 0.864
```

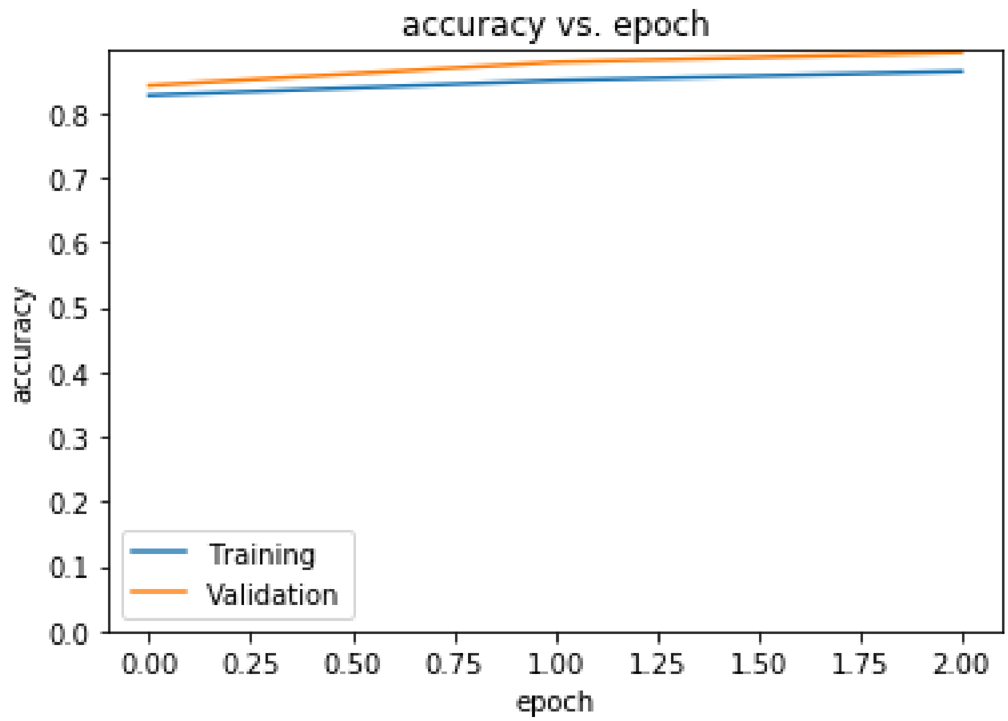
```
In [26]: # Confirm that we've saved a checkpoint during training
         !ls cnn_checkpoints/checkpoint
```

```
assets          saved_model.pb         variables
```

```
In [32]: # Plot the learning curves for loss vs epoch and accuracy vs epoch for both training a
         # Use function defined earlier.
         plot_curves(cnn_history, 'loss')
```




```
In [33]: # (Note, weirdly, how the model performs better on validation set ... just a fluke?)
         plot_curves(cnn_history, 'accuracy')
```



In [34]:
```
# Compute and display the loss and accuracy of the trained model on the test set.

cnn_test_loss, cnn_test_accuracy = cnn_model.evaluate(x_test, y_test, verbose=2)
print(cnn_test_loss)
print(cnn_test_accuracy)
```

```
26032/1 - 67s - loss: 0.3061 - accuracy: 0.8904
0.37037192306583766
0.89044255
```

In [35]:
```
# "You should aim to beat the MLP model performance ..."
# Note: I'm interpreting "beat" to mean test accuracy
print(mlp_test_accuracy)
print(cnn_test_accuracy)
assert cnn_test_accuracy > mlp_test_accuracy
```

```
0.77039796
0.89044255
```

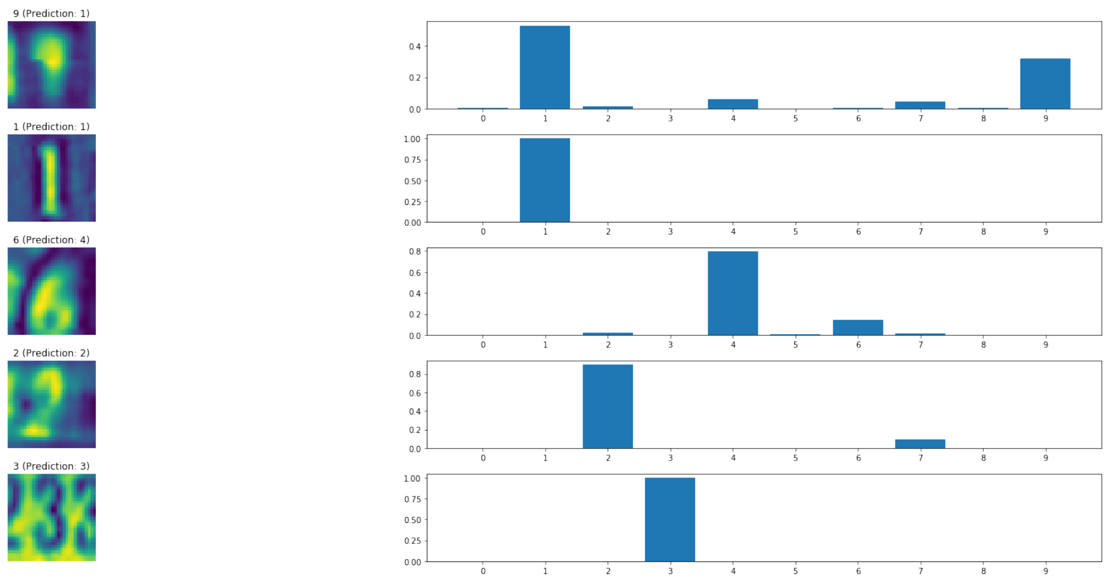
1.5 4. Get model predictions

- 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 [36]:
```
# Note these are the best models because we used save_best_only
loaded_mlp_model = tf.keras.models.load_model(mlp_model_path)
loaded_cnn_model = tf.keras.models.load_model(cnn_model_path)
```

In [52]:
```
num_random_test_images = 5
def show_model_predictions(model):
    random_example_indexes = np.random.choice(num_test_examples, size=num_random_test_
    random_x_subset = x_test[random_example_indexes]
    random_y_subset = y_test[random_example_indexes]
    predictions = model.predict(random_x_subset)
    fig, ax = plt.subplots(num_random_test_images, 2, figsize=(num_random_test_images
    fig.tight_layout()
    for i in range(num_random_test_images):
        image = random_x_subset[i]
        true_label = random_y_subset[i]
        prediction = predictions[i]
        ax[i][0].set_axis_off()
        ax[i][0].imshow(image[:, :, 0])
        ax[i][0].set_title(f"{true_label} (Prediction: {np.argmax(prediction)})")
        numbers = range(10)
        ax[i][1].bar(numbers, prediction)
        ax[i][1].set_xticks(numbers)

# First show for MLP
show_model_predictions(loaded_mlp_model)
```



In [53]: # Now for CNN
```
show_model_predictions(loaded_cnn_model)
```