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 [0]:

```
import tensorflow as tf
from scipy.io import loadmat
```

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]:

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

# import sys
# import os
# path='/content/drive/My Drive/INSAID/TensorFlow/Getting started with TensorFlow 2/Week5/notebooks'
# sys.path.append(path)
# os.chdir(path)
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qd
gf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&res
ponse_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.g
oogleapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%
3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/drive
```

In [0]:

```
# 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.

In [4]:

```
# Let's see the keys in the dictionary

print(train.keys())
print(test.keys())
```

```
dict_keys(['__header__', '__version__', '__globals__', 'X', 'y'])
dict_keys(['__header__', '__version__', '__globals__', 'X', 'y'])
```

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.

Extract the training and testing images and labels separately from the train and test dictionaries

In [5]:

```
# Extracting training and test images and corresponding labels

train_images, train_labels = train['X'], train['y']
test_images, test_labels = test['X'], test['y']

print("Train Data : ",train_images.shape, train_labels.shape)
print("Test Data : ",test_images.shape, test_labels.shape)
```

```
Train Data :  (32, 32, 3, 73257) (73257, 1)
Test Data :  (32, 32, 3, 26032) (26032, 1)
```

No. of samples is the 4th dimension, lets bring it back as the first dimesion (as usual)

In [6]:

```
train_images = train_images.transpose((3, 0, 1, 2))
test_images = test_images.transpose((3, 0, 1, 2))
print("Train Data : ",train_images.shape, train_labels.shape)
print("Test Data : ",test_images.shape, test_labels.shape)
```

```
Train Data :  (73257, 32, 32, 3) (73257, 1)
Test Data :  (26032, 32, 32, 3) (26032, 1)
```

In [0]:

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

```
import pandas as pd
```

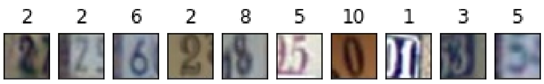
In [0]:

```
# Plot a random sample of images (alteast 10)

def plot_images(image, labels, row=1, col=10):
    '''
    Plot a random row * col images.
    '''
    fig, ax = plt.subplots(row, col)

    indexes = np.random.randint(1, image.shape[0], row*col)

    x=0;
    for i, ax in enumerate(ax.flat):
        img_index = indexes[i]

        if image[img_index].shape == (32,32,3):
            ax.imshow(image[img_index])
            ax.set_title(labels[img_index][0])
        else:
            ax.imshow(image[img_index, :, :, 0])
            ax.set_title(np.argmax(labels[img_index]))

        ax.set_xticks([]); ax.set_yticks([])

#         plt.tight_layout()
```

Select a random sample of images and corresponding labels from the dataset (at least 10)

In [9]:

```
# Plot 10 train images

plot_images(train_images, train_labels)
```



In [10]:

```
# Plot 10 test images

plot_images(test_images, test_labels)
```

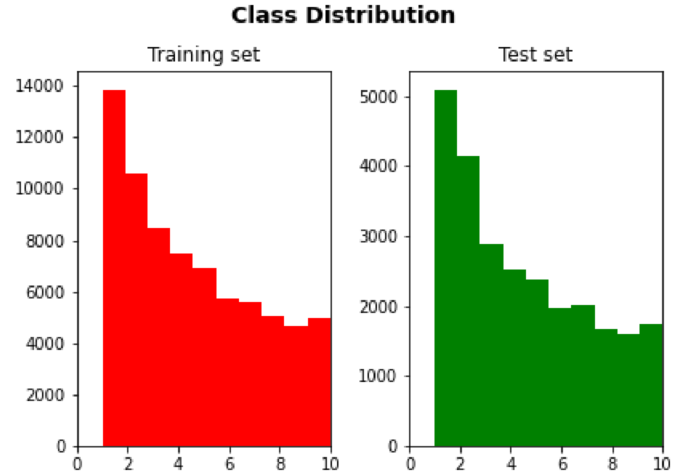
In [11]:

```
# Lets check how may unique numbers in output

print('Train :',np.unique(train_labels))
print('Test :',np.unique(test_labels))
```

```
Train : [ 1  2  3  4  5  6  7  8  9 10]
Test : [ 1  2  3  4  5  6  7  8  9 10]
```

In [12]:

```python
# Let see the distribution of the labels/classes
def plot_class_distribution(ytrain, ytest):
    fig, (ax1, ax2) = plt.subplots(1, 2, sharex=True)
    fig.suptitle('Class Distribution', fontsize=14, fontweight='bold', y=1.05)

    ax1.hist(ytrain, bins=10, color='r')
    ax1.set_title("Training set")
    ax1.set_xlim(0, 10)

    ax2.hist(ytest, bins=10, color='g')
    ax2.set_title("Test set")

    fig.tight_layout()

plot_class_distribution(train_labels, test_labels)
```



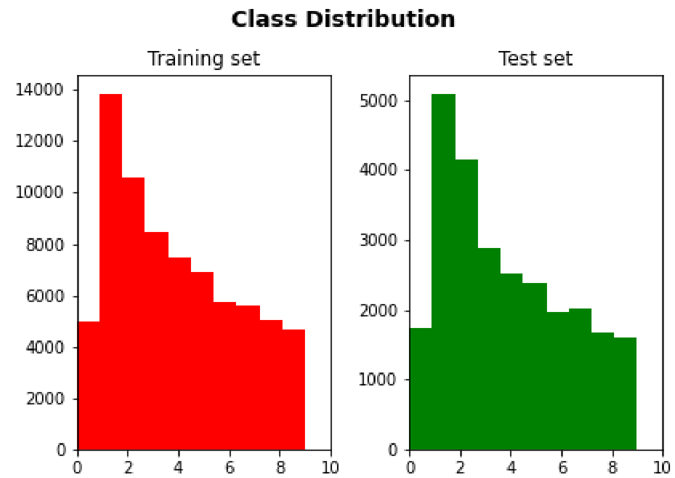
Both the plots show right skewness, which means we have high number of images for lower values when compared to high values

In [0]:

```python
# For simpicity of programming, lets convert the class 10 as class 0
test_labels[test_labels == 10] = 0
train_labels[train_labels == 10] = 0
```

In [14]:

```python
plot_class_distribution(train_labels, test_labels)
```



Convert the training and test images to grayscale by taking the average across all colour channels for each pixel

In [0]:

```
# Convert the training and test images to grayscale by taking the average across all colour channels fo
r each pixel

def rgb2grayscale(images):
    return np.expand_dims(np.dot(images, [0.2990, 0.5870, 0.1140]), axis=images.ndim-1)
```

In [16]:

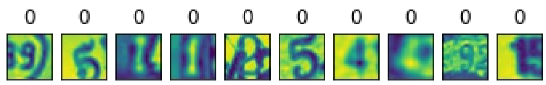
```
train_images_grayscale = rgb2grayscale(train_images)
test_images_grayscale = rgb2grayscale(test_images)

print("Train (grayscale) :", train_images_grayscale.shape)
print("Test (grayscale) :", test_images_grayscale.shape)
```

```
Train (grayscale) : (73257, 32, 32, 1)
Test (grayscale) : (26032, 32, 32, 1)
```

Select a random sample of the grayscale images and corresponding labels from the dataset (at least 10)

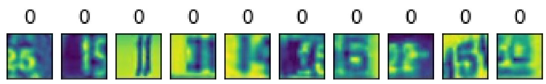
In [17]:

```
# Lets plot the gray scaled images

plot_images(train_images_grayscale, train_labels)
```



In [18]:

```
plot_images(test_images_grayscale, test_labels)
```



Data Normalization

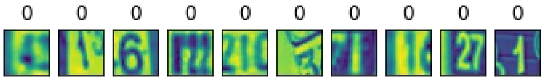
In [0]:

```
# Lets normalize the data-set

# Mean and Std.Dev values
train_mean = np.mean(train_images_grayscale, axis=0)
train_std = np.std(train_images_grayscale, axis=0)

# Subt. mean and div. by std. dev
train_images_norm = (train_images_grayscale - train_mean) / train_std
test_images_norm = (test_images_grayscale - train_mean) / train_std
```

In [20]:

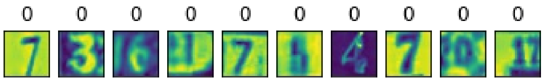
```
# Lets plot the gray scaled normalized images
```

```
plot_images(train_images_norm, train_labels)
```



In [21]:

```
plot_images(test_images_norm, test_labels)
```



Now, lets split the train images into train set and validation set

In [0]:

```
# Splitting the Training data into Train and Validation sets.
# 13% of train set gives around 9500 data having min. of 800 instances of each class
# Using random state to regenrate the whole Dataset in re-run

from sklearn.model_selection import train_test_split

xtrain, xval, ytrain, yval = train_test_split(train_images_norm, train_labels, test_size=.13, random_st
ate=42)
xtest = test_images_norm # for same naming convention
ytest = test_labels
```

One-Hot encoding target variable

In [0]:

```
# For model prediction purpose, lets one-hot encode the target varialble.
# Apply One Hot Encoding to make label suitable for CNN Classification

from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder().fit(ytrain.reshape(-1,1))

y_train = ohe.transform(ytrain.reshape(-1,1)).toarray()
y_val = ohe.transform(yval.reshape(-1,1)).toarray()
y_test = ohe.transform(ytest.reshape(-1,1)).toarray()
```

In [24]:

```
# ytrain.reshape(-1,1)
ohe.transform(ytrain[0].reshape(-1,1)).toarray()
```

Out[24]:

```
array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [25]:

```
print('y_train :', y_train.shape)
print('y_val :', y_val.shape)
print('y_test :', y_test.shape)
```

```
y_train : (63733, 10)
y_val : (9524, 10)
```

```
y_val : (9324, 10)
y_test : (26032, 10)
```

Store the processed data to disk.

In [0]:

```
# Storing only the Grayscale Data not the RGB

import h5py

# Create file
h5f = h5py.File('SVHN_grey.h5', 'w')

# Store the datasets
h5f.create_dataset('X_train', data=xtrain)
h5f.create_dataset('y_train', data=y_train)
h5f.create_dataset('X_test', data=xtest)
h5f.create_dataset('y_test', data=y_test)
h5f.create_dataset('X_val', data=xval)
h5f.create_dataset('y_val', data=y_val)

# Close the file
h5f.close()
```

Free-up RAM memory

In [0]:

```
# Lets delete all the data loaded into memory to free up some RAM mem.

del y_train, y_val, y_test, xtrain, xtest, xval, ytrain, ytest, yval, train_images_norm, test_images_no
rm, \
train_images_grayscale, test_images_grayscale, train_images, test_images, train_labels, test_labels, tr
ain, test
```

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 [28]:

```
# Open file in read mode
import h5py
h5f = h5py.File('SVHN_grey.h5', 'r')

# Read the dataset into local variables
x_train = h5f['X_train'][:]
y_train = h5f['y_train'][:]
x_test = h5f['X_test'][:]
y_test = h5f['y_test'][:]
x_val = h5f['X_val'][:]
y_val = h5f['y_val'][:]
```

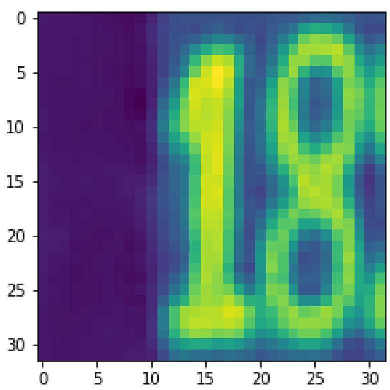
```python
# Close file
h5f.close()

print('Training set', x_train.shape, y_train.shape)
print('Validation set', x_val.shape, y_val.shape)
print('Test set', x_test.shape, y_test.shape)
```

```
Training set (63733, 32, 32, 1) (63733, 10)
Validation set (9524, 32, 32, 1) (9524, 10)
Test set (26032, 32, 32, 1) (26032, 10)
```

In [29]:

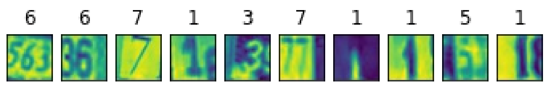
```python
# Display one of the images

i = 0
labels = np.argmax(y_train[i])
img = x_train[i,:,:,0]
plt.imshow(img)
# plt.show()
print(f"label: {labels}")
```

```
label: 1
```



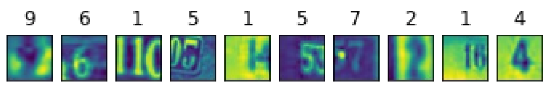
In [30]:

```python
plot_images(x_train, y_train)
```



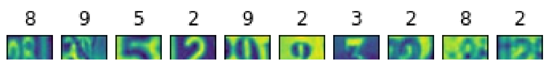
In [31]:

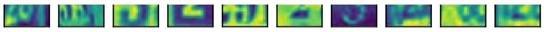
```python
plot_images(x_test, y_test)
```



In [32]:

```python
plot_images(x_val, y_val)
```





In [0]:

```
# import lib
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.callbacks import Callback, ModelCheckpoint, EarlyStopping
from tensorflow.keras.optimizers import Adam
```

Build and compile the model

In [0]:

```
# Build an MLP classifier model using the Sequential API.
# Model should use only Flatten and Dense layers, with the final layer having a 10-way softmax output

def getModel(input_shape):
    model = Sequential([
        Flatten(input_shape=input_shape, name='Flatten'),
        Dense(128, activation='relu', name='Dense_1'),
        Dense(128, activation='relu', name='Dense_2'),
        Dense(128, activation='relu', name='Dense_3'),
        Dense(128, activation='relu', name='Dense_4'),
        Dense(128, activation='relu', name='Dense_5'),
        Dense(10, activation='softmax', name='Dense_6')
    ])

    model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy', metrics=['accu
racy'])
    return model
```

Print out the model summary

In [35]:

```
print(x_train[0].shape)
model = getModel(x_train[0].shape)
model.summary()
```

```
(32, 32, 1)
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
Dense_1 (Dense)              (None, 128)               131200
_________________________________________________________________
Dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
Dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
Dense_4 (Dense)              (None, 128)               16512
_________________________________________________________________
Dense_5 (Dense)              (None, 128)               16512
_________________________________________________________________
Dense_6 (Dense)              (None, 10)                1290
=================================================================
Total params: 198,538
Trainable params: 198,538
Non-trainable params: 0
_________________________________________________________________
```

In [36]:

```python
print(f'Loss :{model.loss}')
print(f'Learning Rate :{model.optimizer.lr}, \nOptimizer: {model.optimizer}')
print(f'Mertrics : {model.metrics}')
```

```
Loss :categorical_crossentropy
Learning Rate :<tf.Variable 'learning_rate:0' shape=() dtype=float32, numpy=1e-04>,
Optimizer: <tensorflow.python.keras.optimizer_v2.adam.Adam object at 0x7fdabdf94d30>
Mertrics : []
```

Custom Callback

In [0]:

```python
class my_callback(Callback):
    def on_train_begin(self, logs=None):
        print("Starting training....")

    def on_epoch_begin(self, epoch, logs=None):
        print(f"Starting epoch {epoch}")

    def on_epoch_end(self, epoch, logs=None):
        print(f"Finishing epoch {epoch}")

    def on_train_end(self, logs=None):
        print("Finished training:")

#  track at least one appropriate metric
def checkpoint_getBestOnly():
    checkpoint_path='model_checkpoint/checkpoint'
    checkpoint = ModelCheckpoint(filepath=checkpoint_path, save_freq='epoch',
                                 save_best_only=False, verbose=1,
                                 save_weights_only=True, monitor = 'val_accuracy',
                                )
    return checkpoint
```

Train the model

In [38]:

```python
history = model.fit(x_train, y_train, epochs=30,
                    validation_data=(x_val,y_val), batch_size=64,
                    callbacks=[my_callback(), checkpoint_getBestOnly()])
```

```
Starting training....
Starting epoch 0
Epoch 1/30
979/996 [============================>.] - ETA: 0s - loss: 1.5984 - accuracy: 0.4618Finishing epoch 0

Epoch 00001: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 1.5902 - accuracy: 0.4649 - val_loss: 1.
0891 - val_accuracy: 0.6624
Starting epoch 1
Epoch 2/30
984/996 [============================>.] - ETA: 0s - loss: 0.9935 - accuracy: 0.6936Finishing epoch 1

Epoch 00002: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.9925 - accuracy: 0.6939 - val_loss: 0.
9008 - val_accuracy: 0.7241
Starting epoch 2
Epoch 3/30
984/996 [============================>.] - ETA: 0s - loss: 0.8417 - accuracy: 0.7426Finishing epoch 2

Epoch 00003: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.8414 - accuracy: 0.7428 - val_loss: 0.
7958 - val_accuracy: 0.7557
Starting epoch 3
Epoch 4/30
991/996 [============================>.] - ETA: 0s - loss: 0.7526 - accuracy: 0.7704Finishing epoch 3
```

```
Epoch 00004: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.7523 - accuracy: 0.7704 - val_loss: 0.
7435 - val_accuracy: 0.7697
Starting epoch 4
Epoch 5/30
982/996 [============================>.] - ETA: 0s - loss: 0.6916 - accuracy: 0.7890Finishing epoch 4

Epoch 00005: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.6922 - accuracy: 0.7890 - val_loss: 0.
7100 - val_accuracy: 0.7867
Starting epoch 5
Epoch 6/30
977/996 [============================>.] - ETA: 0s - loss: 0.6486 - accuracy: 0.8029Finishing epoch 5

Epoch 00006: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.6488 - accuracy: 0.8029 - val_loss: 0.
6791 - val_accuracy: 0.7940
Starting epoch 6
Epoch 7/30
993/996 [============================>.] - ETA: 0s - loss: 0.6137 - accuracy: 0.8126Finishing epoch 6

Epoch 00007: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.6139 - accuracy: 0.8125 - val_loss: 0.
6616 - val_accuracy: 0.7989
Starting epoch 7
Epoch 8/30
995/996 [============================>.] - ETA: 0s - loss: 0.5813 - accuracy: 0.8228Finishing epoch 7

Epoch 00008: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.5817 - accuracy: 0.8227 - val_loss: 0.
6468 - val_accuracy: 0.8071
Starting epoch 8
Epoch 9/30
976/996 [============================>.] - ETA: 0s - loss: 0.5594 - accuracy: 0.8299Finishing epoch 8

Epoch 00009: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.5590 - accuracy: 0.8302 - val_loss: 0.
6379 - val_accuracy: 0.8079
Starting epoch 9
Epoch 10/30
979/996 [============================>.] - ETA: 0s - loss: 0.5377 - accuracy: 0.8365Finishing epoch 9

Epoch 00010: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.5372 - accuracy: 0.8367 - val_loss: 0.
6163 - val_accuracy: 0.8137
Starting epoch 10
Epoch 11/30
993/996 [============================>.] - ETA: 0s - loss: 0.5140 - accuracy: 0.8439Finishing epoch 10

Epoch 00011: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.5142 - accuracy: 0.8439 - val_loss: 0.
6110 - val_accuracy: 0.8180
Starting epoch 11
Epoch 12/30
995/996 [============================>.] - ETA: 0s - loss: 0.4969 - accuracy: 0.8486Finishing epoch 11

Epoch 00012: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4971 - accuracy: 0.8486 - val_loss: 0.
6118 - val_accuracy: 0.8203
Starting epoch 12
Epoch 13/30
991/996 [============================>.] - ETA: 0s - loss: 0.4834 - accuracy: 0.8521Finishing epoch 12

Epoch 00013: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4832 - accuracy: 0.8521 - val_loss: 0.
5906 - val_accuracy: 0.8269
Starting epoch 13
Epoch 14/30
980/996 [============================>.] - ETA: 0s - loss: 0.4651 - accuracy: 0.8581Finishing epoch 13

Epoch 00014: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4656 - accuracy: 0.8578 - val_loss: 0.
5940 - val_accuracy: 0.8235
Starting epoch 14
Epoch 15/30
988/996 [============================>.] - ETA: 0s - loss: 0.4525 - accuracy: 0.8625Finishing epoch 14
```

```
Epoch 00015: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4531 - accuracy: 0.8623 - val_loss: 0.
5866 - val_accuracy: 0.8281
Starting epoch 15
Epoch 16/30
978/996 [============================>.] - ETA: 0s - loss: 0.4401 - accuracy: 0.8650Finishing epoch 15

Epoch 00016: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4407 - accuracy: 0.8651 - val_loss: 0.
5867 - val_accuracy: 0.8299
Starting epoch 16
Epoch 17/30
979/996 [============================>.] - ETA: 0s - loss: 0.4283 - accuracy: 0.8690Finishing epoch 16

Epoch 00017: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4277 - accuracy: 0.8691 - val_loss: 0.
5740 - val_accuracy: 0.8313
Starting epoch 17
Epoch 18/30
977/996 [============================>.] - ETA: 0s - loss: 0.4177 - accuracy: 0.8729Finishing epoch 17

Epoch 00018: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4177 - accuracy: 0.8731 - val_loss: 0.
5814 - val_accuracy: 0.8313
Starting epoch 18
Epoch 19/30
985/996 [============================>.] - ETA: 0s - loss: 0.4048 - accuracy: 0.8758Finishing epoch 18

Epoch 00019: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.4052 - accuracy: 0.8755 - val_loss: 0.
6018 - val_accuracy: 0.8269
Starting epoch 19
Epoch 20/30
995/996 [============================>.] - ETA: 0s - loss: 0.3953 - accuracy: 0.8788Finishing epoch 19

Epoch 00020: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3953 - accuracy: 0.8788 - val_loss: 0.
5808 - val_accuracy: 0.8345
Starting epoch 20
Epoch 21/30
984/996 [============================>.] - ETA: 0s - loss: 0.3862 - accuracy: 0.8824Finishing epoch 20

Epoch 00021: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3870 - accuracy: 0.8820 - val_loss: 0.
5743 - val_accuracy: 0.8352
Starting epoch 21
Epoch 22/30
986/996 [============================>.] - ETA: 0s - loss: 0.3793 - accuracy: 0.8853Finishing epoch 21

Epoch 00022: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3789 - accuracy: 0.8854 - val_loss: 0.
5703 - val_accuracy: 0.8357
Starting epoch 22
Epoch 23/30
980/996 [============================>.] - ETA: 0s - loss: 0.3690 - accuracy: 0.8868Finishing epoch 22

Epoch 00023: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3693 - accuracy: 0.8867 - val_loss: 0.
5646 - val_accuracy: 0.8401
Starting epoch 23
Epoch 24/30
988/996 [============================>.] - ETA: 0s - loss: 0.3617 - accuracy: 0.8895Finishing epoch 23

Epoch 00024: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3621 - accuracy: 0.8894 - val_loss: 0.
5736 - val_accuracy: 0.8365
Starting epoch 24
Epoch 25/30
988/996 [============================>.] - ETA: 0s - loss: 0.3533 - accuracy: 0.8923Finishing epoch 24

Epoch 00025: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3533 - accuracy: 0.8923 - val_loss: 0.
5863 - val_accuracy: 0.8342
Starting epoch 25
Epoch 26/30
988/996 [============================>.] - ETA: 0s - loss: 0.3466 - accuracy: 0.8941Finishing epoch 25
```

```
Epoch 00026: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3465 - accuracy: 0.8943 - val_loss: 0.
5895 - val_accuracy: 0.8337
Starting epoch 26
Epoch 27/30
995/996 [============================>.] - ETA: 0s - loss: 0.3359 - accuracy: 0.8970Finishing epoch 26

Epoch 00027: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3362 - accuracy: 0.8969 - val_loss: 0.
5761 - val_accuracy: 0.8388
Starting epoch 27
Epoch 28/30
984/996 [============================>.] - ETA: 0s - loss: 0.3303 - accuracy: 0.8983Finishing epoch 27

Epoch 00028: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3302 - accuracy: 0.8983 - val_loss: 0.
5652 - val_accuracy: 0.8444
Starting epoch 28
Epoch 29/30
995/996 [============================>.] - ETA: 0s - loss: 0.3247 - accuracy: 0.8998Finishing epoch 28

Epoch 00029: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3247 - accuracy: 0.8999 - val_loss: 0.
5698 - val_accuracy: 0.8408
Starting epoch 29
Epoch 30/30
992/996 [============================>.] - ETA: 0s - loss: 0.3127 - accuracy: 0.9046Finishing epoch 29

Epoch 00030: saving model to model_checkpoint/checkpoint
996/996 [==============================] - 3s 3ms/step - loss: 0.3129 - accuracy: 0.9046 - val_loss: 0.
5848 - val_accuracy: 0.8387
Finished training:
```

Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets

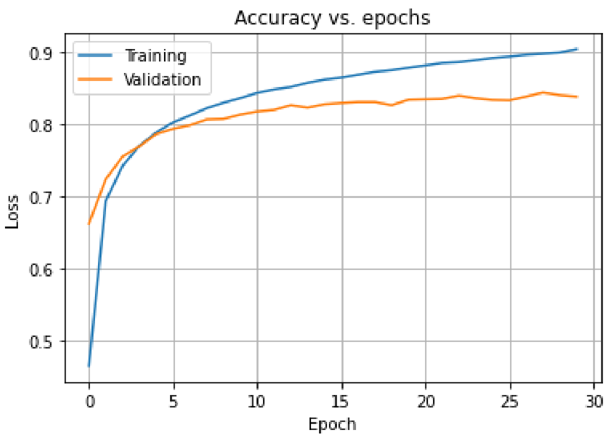
In [39]:

```python
print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

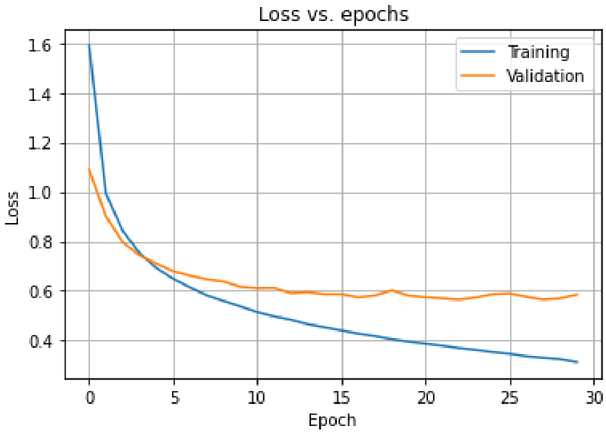
In [40]:

```python
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'])
plt.grid()
```



In [41]:

```
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'])
plt.grid()
```



In [42]:

```
import pandas as pd
df = pd.DataFrame(history.history)
df.head()
```

Out[42]:

	loss	accuracy	val_loss	val_accuracy
0	1.590182	0.464924	1.089091	0.662432
1	0.992526	0.693926	0.900844	0.724066
2	0.841377	0.742771	0.795810	0.755670
3	0.752326	0.770449	0.743468	0.769740
4	0.692217	0.788995	0.710009	0.786749

Compute and display the loss and accuracy of the trained model on the test set

In [43]:

```
loss, accuracy = model.evaluate(x_test, y_test, verbose=2)
print(f'Loss : {loss:.3f} \nAccuracy :{accuracy:.3f}')
```

```
814/814 - 1s - loss: 0.7629 - accuracy: 0.8110
Loss : 0.763
Accuracy :0.811
```

Cleanup

In [44]:

```
!ls -lh model_checkpoint
# !rm -r model_checkpoint
```

```
total 4.6M
-rw------- 1 root root   77 Jun  5 15:22 checkpoint
-rw------- 1 root root 2.3M May 20 20:15 checkpoint.data-00000-of-00001
-rw------- 1 root root 5.1K Jun  5 15:22 checkpoint.data-00000-of-00002
-rw------- 1 root root 2.3M Jun  5 15:22 checkpoint.data-00001-of-00002
-rw------- 1 root root 2.9K Jun  5 15:22 checkpoint.index
```

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 [0]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, BatchNormalization, Dropout
from tensorflow.keras.callbacks import Callback, EarlyStopping, ModelCheckpoint
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam
```

Build a CNN classifier model using the Sequential API

In [0]:

```
def getCNNModel(inputshape, decayRate, dropRate):
    model = Sequential([
        Conv2D(16, kernel_size=3, padding='SAME', activation='relu',
               kernel_initializer='he_uniform', kernel_regularizer=l2(decayRate), bias_initializer='one
s',
               name='conv2d_1', input_shape=(inputshape)),

        Dropout(dropRate, name='dropout_1'),

        Conv2D(16, kernel_size=3, padding='SAME', activation='relu',
               kernel_regularizer=l2(decayRate), name='conv2d_2'),

        Dropout(dropRate, name='dropout_2'),
        BatchNormalization(name='batch_norm_1'),
        MaxPooling2D(pool_size=(4,4), name='max_pool_1'),

        Dense(128, activation='relu', name='dense_1'),
        Flatten(name='flatten_1'),
        Dense(10, activation='softmax', name='dense_2')
    ])

    model.compile(
        optimizer=Adam(learning_rate=0.0001),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

In [47]:

```
model = getCNNModel(x_train[0].shape, 0.001, 0.3)
```

```
model = getCNNModel(x_train[0].shape, 0.001, 0.3)
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 16)        160       
_________________________________________________________________
dropout_1 (Dropout)          (None, 32, 32, 16)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 16)        2320      
_________________________________________________________________
dropout_2 (Dropout)          (None, 32, 32, 16)        0         
_________________________________________________________________
batch_norm_1 (BatchNormaliza (None, 32, 32, 16)        64        
_________________________________________________________________
max_pool_1 (MaxPooling2D)    (None, 8, 8, 16)          0         
_________________________________________________________________
dense_1 (Dense)              (None, 8, 8, 128)         2176      
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 10)                81930     
=================================================================
Total params: 86,650
Trainable params: 86,618
Non-trainable params: 32
_________________________________________________________________
```

In [48]:

```
print(model.optimizer)
print(model.loss)
print(model.metrics)
print(model.optimizer.lr)
```

```
<tensorflow.python.keras.optimizer_v2.adam.Adam object at 0x7fdaa009ee48>
categorical_crossentropy
[]
<tf.Variable 'learning_rate:0' shape=() dtype=float32, numpy=1e-04>
```

In [0]:

```
def get_early_stopping():
    return EarlyStopping(patience=4, monitor='val_accuracy')

def get_best_CNN_checkpoint():
    path = 'model_checkpoint_CNN/checkpoint'
    return ModelCheckpoint(filepath=path, save_freq='epoch',
                           save_weights_only=True, save_best_only=True,
                           monitor='val_accuracy', verbose=1)
```

In [50]:

```
history = model.fit(x_train, y_train, batch_size=256, epochs=30,
                   validation_data=(x_val,y_val),
                   callbacks=[get_early_stopping(), get_best_CNN_checkpoint()])
```

```
Epoch 1/30
249/249 [==============================] - ETA: 0s - loss: 2.2233 - accuracy: 0.2351
Epoch 00001: val_accuracy improved from -inf to 0.35395, saving model to model_checkpoint_CNN/checkpoin
t
249/249 [==============================] - 4s 16ms/step - loss: 2.2233 - accuracy: 0.2351 - val_loss: 2
.0795 - val_accuracy: 0.3539
Epoch 2/30
244/249 [============================>.] - ETA: 0s - loss: 1.7315 - accuracy: 0.4427
Epoch 00002: val_accuracy improved from 0.35395 to 0.49717, saving model to model_checkpoint_CNN/checkp
oint
```

```
oint
249/249 [==============================] - 3s 10ms/step - loss: 1.7268 - accuracy: 0.4446 - val_loss: 1
.6804 - val_accuracy: 0.4972
Epoch 3/30
247/249 [============================>.] - ETA: 0s - loss: 1.3205 - accuracy: 0.5957
Epoch 00003: val_accuracy improved from 0.49717 to 0.60038, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 1.3195 - accuracy: 0.5960 - val_loss: 1
.3471 - val_accuracy: 0.6004
Epoch 4/30
248/249 [============================>.] - ETA: 0s - loss: 1.0939 - accuracy: 0.6788
Epoch 00004: val_accuracy improved from 0.60038 to 0.66296, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 1.0936 - accuracy: 0.6789 - val_loss: 1
.1504 - val_accuracy: 0.6630
Epoch 5/30
244/249 [============================>.] - ETA: 0s - loss: 0.9659 - accuracy: 0.7208
Epoch 00005: val_accuracy improved from 0.66296 to 0.68564, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.9644 - accuracy: 0.7214 - val_loss: 1
.0441 - val_accuracy: 0.6856
Epoch 6/30
246/249 [============================>.] - ETA: 0s - loss: 0.8810 - accuracy: 0.7499
Epoch 00006: val_accuracy improved from 0.68564 to 0.71336, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 0.8810 - accuracy: 0.7497 - val_loss: 0
.9595 - val_accuracy: 0.7134
Epoch 7/30
248/249 [============================>.] - ETA: 0s - loss: 0.8227 - accuracy: 0.7700
Epoch 00007: val_accuracy improved from 0.71336 to 0.73992, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.8226 - accuracy: 0.7700 - val_loss: 0
.8992 - val_accuracy: 0.7399
Epoch 8/30
244/249 [============================>.] - ETA: 0s - loss: 0.7858 - accuracy: 0.7810
Epoch 00008: val_accuracy improved from 0.73992 to 0.75829, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 0.7860 - accuracy: 0.7809 - val_loss: 0
.8475 - val_accuracy: 0.7583
Epoch 9/30
248/249 [============================>.] - ETA: 0s - loss: 0.7514 - accuracy: 0.7929
Epoch 00009: val_accuracy improved from 0.75829 to 0.77919, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.7507 - accuracy: 0.7931 - val_loss: 0
.8019 - val_accuracy: 0.7792
Epoch 10/30
249/249 [==============================] - ETA: 0s - loss: 0.7279 - accuracy: 0.8004
Epoch 00010: val_accuracy improved from 0.77919 to 0.78990, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 0.7279 - accuracy: 0.8004 - val_loss: 0
.7672 - val_accuracy: 0.7899
Epoch 11/30
248/249 [============================>.] - ETA: 0s - loss: 0.7039 - accuracy: 0.8086
Epoch 00011: val_accuracy improved from 0.78990 to 0.79084, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.7042 - accuracy: 0.8085 - val_loss: 0
.7478 - val_accuracy: 0.7908
Epoch 12/30
245/249 [============================>.] - ETA: 0s - loss: 0.6831 - accuracy: 0.8133
Epoch 00012: val_accuracy improved from 0.79084 to 0.80460, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 0.6842 - accuracy: 0.8131 - val_loss: 0
.7197 - val_accuracy: 0.8046
Epoch 13/30
245/249 [============================>.] - ETA: 0s - loss: 0.6691 - accuracy: 0.8187
Epoch 00013: val_accuracy improved from 0.80460 to 0.81121, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.6693 - accuracy: 0.8187 - val_loss: 0
.7020 - val_accuracy: 0.8112
Epoch 14/30
244/249 [============================>.] - ETA: 0s - loss: 0.6556 - accuracy: 0.8228
Epoch 00014: val_accuracy improved from 0.81121 to 0.81709, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.6549 - accuracy: 0.8230 - val_loss: 0
.6857 - val_accuracy: 0.8171
Epoch 15/30
249/249 [==============================] - ETA: 0s - loss: 0.6441 - accuracy: 0.8271
Epoch 00015: val_accuracy improved from 0.81709 to 0.81877, saving model to model_checkpoint_CNN/checkp
```

```
Epoch 00015: val_accuracy improved from 0.81709 to 0.81877, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.6441 - accuracy: 0.8271 - val_loss: 0
.6770 - val_accuracy: 0.8188
Epoch 16/30
248/249 [============================>.] - ETA: 0s - loss: 0.6293 - accuracy: 0.8298
Epoch 00016: val_accuracy improved from 0.81877 to 0.82098, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.6294 - accuracy: 0.8297 - val_loss: 0
.6648 - val_accuracy: 0.8210
Epoch 17/30
246/249 [============================>.] - ETA: 0s - loss: 0.6166 - accuracy: 0.8346
Epoch 00017: val_accuracy improved from 0.82098 to 0.82675, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.6170 - accuracy: 0.8344 - val_loss: 0
.6471 - val_accuracy: 0.8268
Epoch 18/30
248/249 [============================>.] - ETA: 0s - loss: 0.6067 - accuracy: 0.8375
Epoch 00018: val_accuracy did not improve from 0.82675
249/249 [==============================] - 2s 9ms/step - loss: 0.6066 - accuracy: 0.8375 - val_loss: 0.
6448 - val_accuracy: 0.8259
Epoch 19/30
244/249 [============================>.] - ETA: 0s - loss: 0.5966 - accuracy: 0.8395
Epoch 00019: val_accuracy improved from 0.82675 to 0.83253, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.5971 - accuracy: 0.8393 - val_loss: 0
.6233 - val_accuracy: 0.8325
Epoch 20/30
249/249 [==============================] - ETA: 0s - loss: 0.5877 - accuracy: 0.8432
Epoch 00020: val_accuracy improved from 0.83253 to 0.83358, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.5877 - accuracy: 0.8432 - val_loss: 0
.6235 - val_accuracy: 0.8336
Epoch 21/30
244/249 [============================>.] - ETA: 0s - loss: 0.5768 - accuracy: 0.8458
Epoch 00021: val_accuracy improved from 0.83358 to 0.83631, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 0.5773 - accuracy: 0.8455 - val_loss: 0
.6134 - val_accuracy: 0.8363
Epoch 22/30
244/249 [============================>.] - ETA: 0s - loss: 0.5720 - accuracy: 0.8469
Epoch 00022: val_accuracy did not improve from 0.83631
249/249 [==============================] - 2s 9ms/step - loss: 0.5719 - accuracy: 0.8472 - val_loss: 0.
6080 - val_accuracy: 0.8360
Epoch 23/30
245/249 [============================>.] - ETA: 0s - loss: 0.5607 - accuracy: 0.8495
Epoch 00023: val_accuracy improved from 0.83631 to 0.84345, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.5602 - accuracy: 0.8497 - val_loss: 0
.5938 - val_accuracy: 0.8434
Epoch 24/30
249/249 [==============================] - ETA: 0s - loss: 0.5519 - accuracy: 0.8532
Epoch 00024: val_accuracy did not improve from 0.84345
249/249 [==============================] - 2s 9ms/step - loss: 0.5519 - accuracy: 0.8532 - val_loss: 0.
5985 - val_accuracy: 0.8382
Epoch 25/30
249/249 [==============================] - ETA: 0s - loss: 0.5464 - accuracy: 0.8549
Epoch 00025: val_accuracy improved from 0.84345 to 0.85080, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 0.5464 - accuracy: 0.8549 - val_loss: 0
.5742 - val_accuracy: 0.8508
Epoch 26/30
249/249 [==============================] - ETA: 0s - loss: 0.5367 - accuracy: 0.8570
Epoch 00026: val_accuracy did not improve from 0.85080
249/249 [==============================] - 2s 9ms/step - loss: 0.5367 - accuracy: 0.8570 - val_loss: 0.
5743 - val_accuracy: 0.8497
Epoch 27/30
246/249 [============================>.] - ETA: 0s - loss: 0.5306 - accuracy: 0.8595
Epoch 00027: val_accuracy improved from 0.85080 to 0.85090, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 2s 10ms/step - loss: 0.5307 - accuracy: 0.8593 - val_loss: 0
.5673 - val_accuracy: 0.8509
Epoch 28/30
245/249 [============================>.] - ETA: 0s - loss: 0.5237 - accuracy: 0.8608
Epoch 00028: val_accuracy did not improve from 0.85090
249/249 [==============================] - 2s 9ms/step - loss: 0.5233 - accuracy: 0.8609 - val_loss: 0.
5635 - val_accuracy: 0.8497
Epoch 29/30
```

```
Epoch 29/30
247/249 [============================>.] - ETA: 0s - loss: 0.5174 - accuracy: 0.8633
Epoch 00029: val_accuracy improved from 0.85090 to 0.85363, saving model to model_checkpoint_CNN/checkp
oint
249/249 [==============================] - 3s 10ms/step - loss: 0.5183 - accuracy: 0.8631 - val_loss: 0
.5580 - val_accuracy: 0.8536
Epoch 30/30
249/249 [==============================] - ETA: 0s - loss: 0.5122 - accuracy: 0.8634
Epoch 00030: val_accuracy did not improve from 0.85363
249/249 [==============================] - 2s 9ms/step - loss: 0.5122 - accuracy: 0.8634 - val_loss: 0.
5605 - val_accuracy: 0.8486
```

Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets

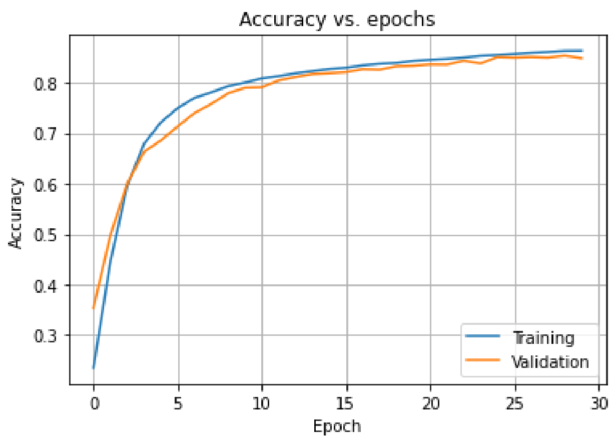
In [51]:

```python
print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

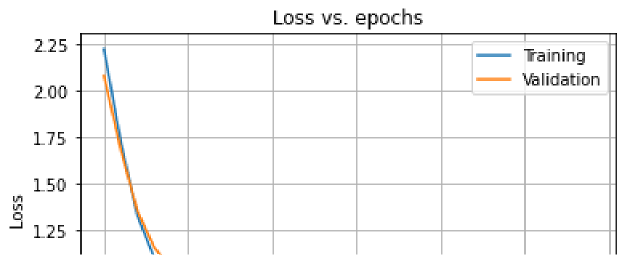
In [52]:

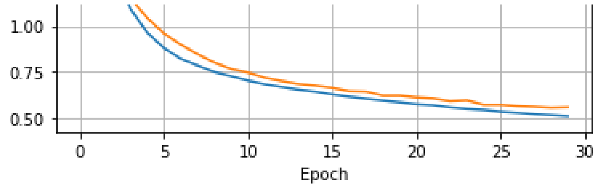
```python
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'])
plt.grid()
```



In [53]:

```python
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'])
plt.grid()
```





In [54]:

```
import pandas as pd
df = pd.DataFrame(history.history)
df.head()
```

Out[54]:

	loss	accuracy	val_loss	val_accuracy
0	2.223334	0.235059	2.079538	0.353948
1	1.726768	0.444558	1.680401	0.497165
2	1.319486	0.596049	1.347089	0.600378
3	1.093594	0.678895	1.150359	0.662957
4	0.964386	0.721416	1.044122	0.685636

Compute and display the loss and accuracy of the trained model on the test set

In [55]:

```
loss, accuracy = model.evaluate(x_test, y_test, verbose=2)
print(f'Loss : {loss:.3f} \nAccuracy :{accuracy:.3f}')
```

```
814/814 - 2s - loss: 0.6014 - accuracy: 0.8394
Loss : 0.601
Accuracy :0.839
```

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.

Load the best weights for the MLP and CNN models

In [56]:

```
model_mlp = getModel(x_train[0].shape)
model_mlp.load_weights(filepath='model_checkpoint/checkpoint')
```

Out[56]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7fdabb65b860>
```

In [57]:

```
model_cnn = getCNNModel(x_train[0].shape, 0.001, 0.3)
model_cnn.load_weights(filepath='model_checkpoint_CNN/checkpoint')
```

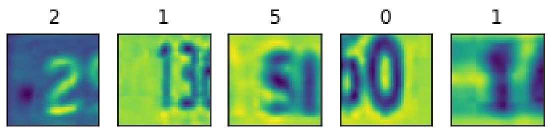
Out[57]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7fdaa035f898>
```

Randomly select 5 images and corresponding labels from the test set and display the images with their labels.

In [58]:

```python
plot_images(x_test, y_test, 1, 5)
```

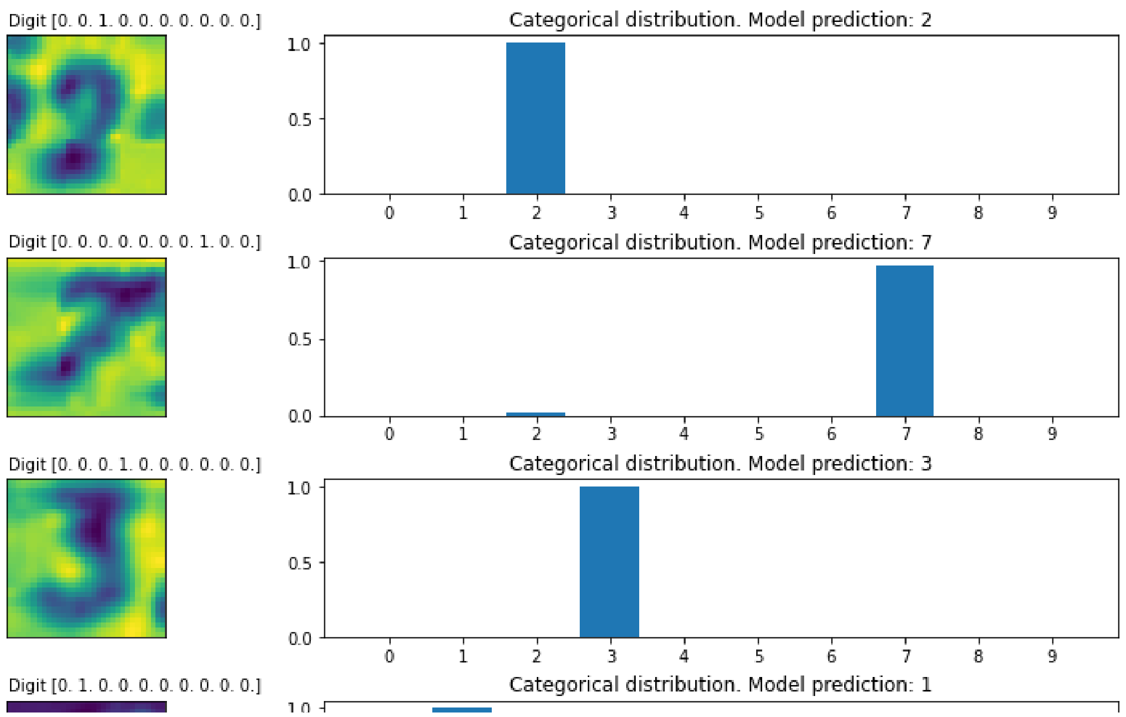


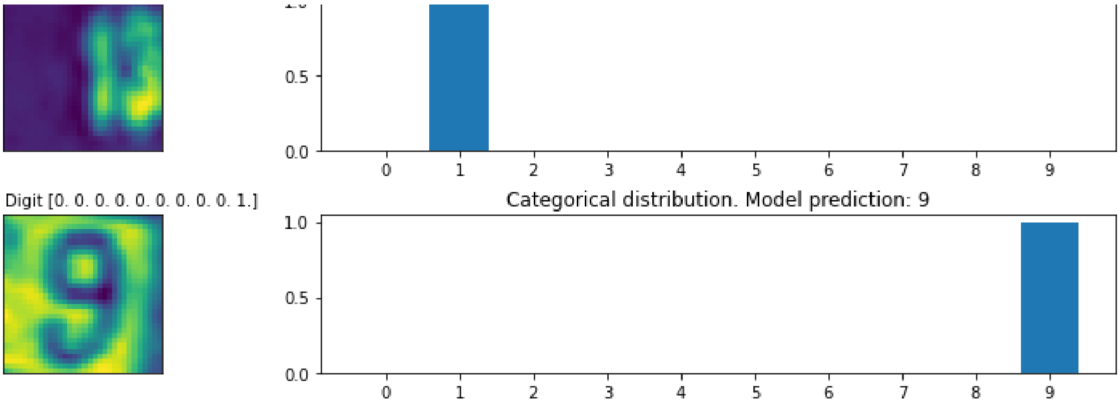
In [0]:

```python
def plot_prediction_bars(predictions, images, labels):
    fig, axes = plt.subplots(5, 2, figsize=(16, 12))
    fig.subplots_adjust(hspace=0.4, wspace=-0.2)

    for i, (prediction, image, label) in enumerate(zip(predictions, images, 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(0., -2.5, 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)}")
```

In [60]:

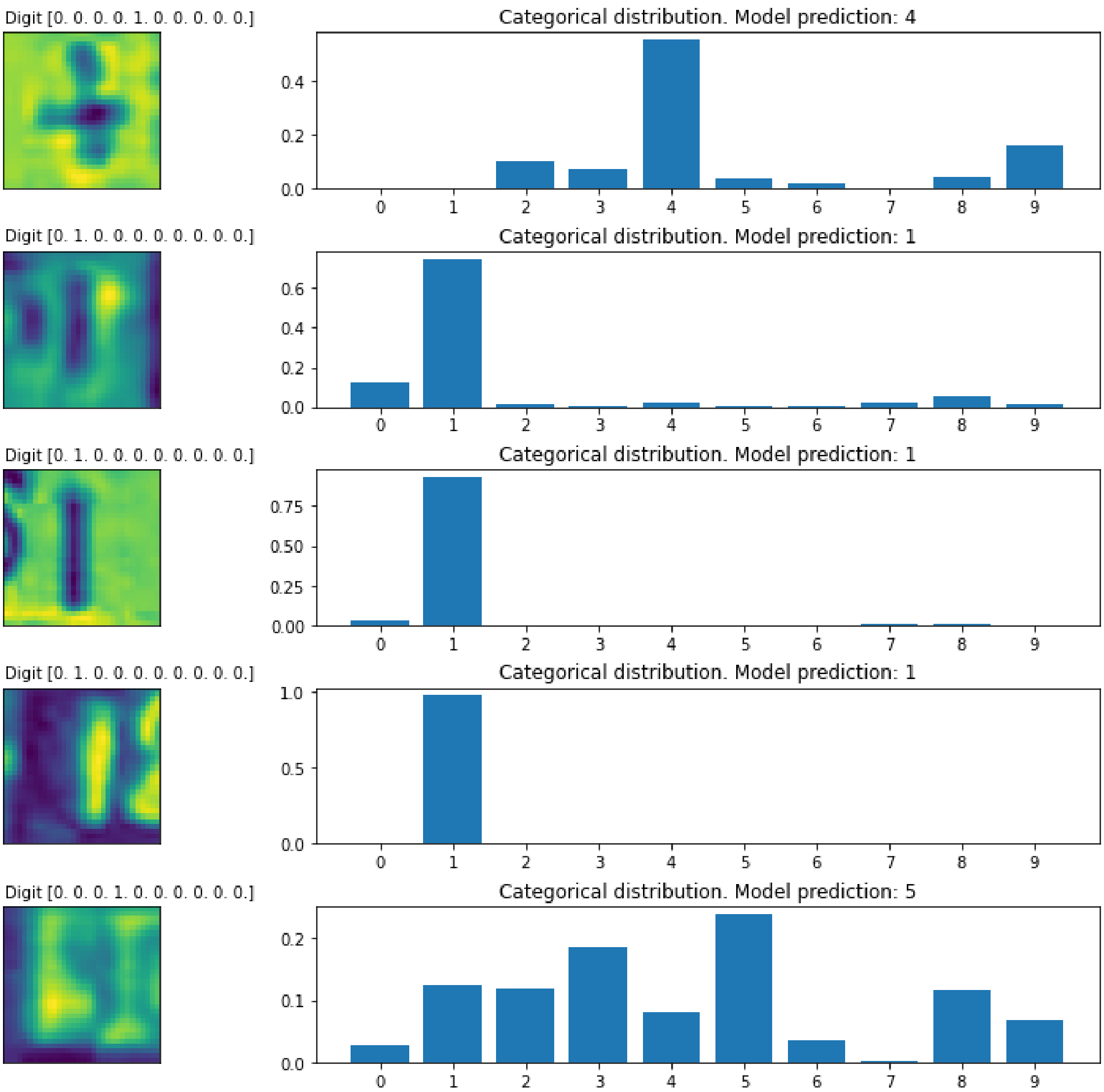
```python
num_test_images = x_test.shape[0]     # Get the total no. of images
indexes = np.random.choice(num_test_images, 5) # Choose 5 random indexes
images = x_test[indexes, ...]   # Get the random 5 images
labels = y_test[indexes, ...]   # Get correspodig labels

predictions = model_mlp.predict(images)
plot_prediction_bars(predictions, images, labels)
```





In [61]:

```
num_test_images = x_test.shape[0]     # Get the total no. of images
indexes = np.random.choice(num_test_images, 5) # Choose 5 random indexes
images = x_test[indexes, ...]   # Get the random 5 images
labels = y_test[indexes, ...]   # Get correspodig labels

predictions = model_cnn.predict(images)
plot_prediction_bars(predictions, images, labels)
```



In [0]:

```
# THANK YOU
```