Capstone Project

Image classifier for the SVHN dataset

Instructions

In this notebook, you will create a neural network that classifies real-world images digits. You will use concepts from throughout this course in building, training, testing, validating and saving your Tensorflow classifier model.

This project is peer-assessed. Within this notebook you will find instructions in each section for how to complete the project. Pay close attention to the instructions as the peer review will be carried out according to a grading rubric that checks key parts of the project instructions. Feel free to add extra cells into the notebook as required.

How to submit

When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able to view the outputs of your notebook. Save the notebook as a pdf (File -> Download as -> PDF via LaTeX). You should then submit this pdf for review.

Let's get started!

We'll start by running some imports, and loading the dataset. For this project you are free to make further imports throughout the notebook as you wish.

In [1]:

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

For the capstone project, you will use the SVHN dataset (http://ufldl.stanford.edu/housenumbers/). This is an image dataset of over 600,000 digit images in all, and is a harder dataset than MNIST as the numbers appear in the context of natural scene images. SVHN is obtained from house numbers in Google Street View images.

- Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng. "Reading Digits in Natural Images with Unsupervised Feature Learning". NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.

Your goal is to develop an end-to-end workflow for building, training, validating, evaluating and saving a neural network that classifies a real-world image into one of ten classes.

In [2]:

```
# Run this cell to load the dataset

train = loadmat('data/train_32x32.mat')
test = loadmat('data/test_32x32.mat')
```

Both `train` and `test` are dictionaries with keys `X` and `y` for the input images and labels respectively.

1. Inspect and preprocess the dataset

- Extract the training and testing images and labels separately from the train and test dictionaries loaded for you.
- Select a random sample of images and corresponding labels from the dataset (at least 10), and display them in a figure.
- Convert the training and test images to grayscale by taking the average across all colour channels for each pixel. *Hint: retain the channel dimension, which will now have size 1.*
- Select a random sample of the grayscale images and corresponding labels from the dataset (at least 10), and display them in a figure.

In [3]:

```
#Extracting data
X_train = train['X']
y_train = train['y']

X_test = test['X']
y_test = test['y']
```

In [4]:

```
#Understanding some data details
import numpy as np

print(np.unique(y_train))
print(X_train.shape)
print(y_train.shape)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
(32, 32, 3, 73257)
(73257, 1)
```

In [5]:

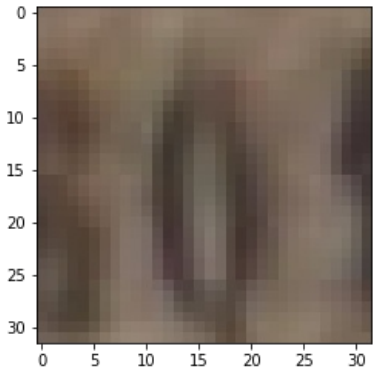
```
#Showing some random data and their classes
import matplotlib.pyplot as plt
%matplotlib inline

random_images = np.random.randint(y_train.shape[0], size=10)

fig, ax = plt.subplots(2, 5, figsize=(9, 6))
for i, j in zip(range(10), random_images):
    ax[i//5, i%5].imshow(X_train[:, :, :, j])
    ax[i//5, i%5].axis('off')
    ax[i//5, i%5].set_title('Class: {}'.format(y_train[j, 0]))
```

In [6]:

```
#Confirming that class 10 is actually the digit '0'
plt.imshow(X_train[:, :, :, np.where(y_train[:, 0]==10)[0][0]])
```

Out[6]:

```
<matplotlib.image.AxesImage at 0x7f90b06bf588>
```



In [7]:

```
#Encoding class 10 to class 0
lista_y_train = np.where(y_train[:, 0]==10)
y_train[lista_y_train[0], 0] = 0

lista_y_test = np.where(y_test[:, 0]==10)
y_test[lista_y_test[0], 0] = 0

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

```
(73257, 1)
(26032, 1)
```

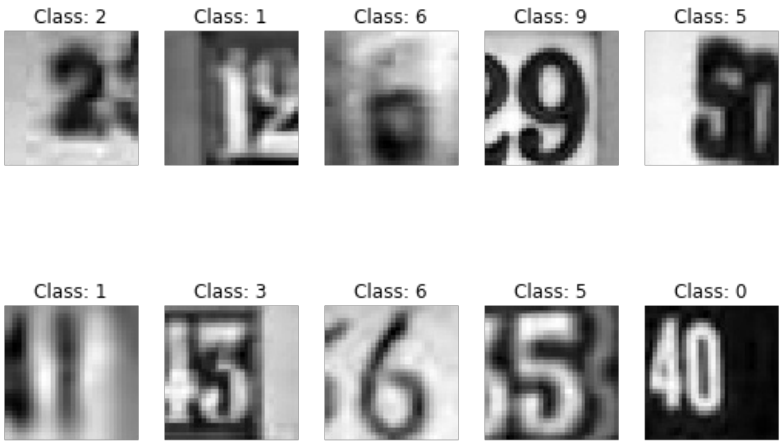
In [8]:

```
#Generating grey data - with and without additional dimension
X_train_grey = np.mean(X_train[:, :, :, :], axis=2, keepdims=True)
X_test_grey = np.mean(X_test[:, :, :, :], axis=2, keepdims=True)
print(X_train_grey.shape)
print(X_test_grey.shape)
print()

X_train_grey_without_dim = np.mean(X_train[:, :, :, :], axis=2)
X_test_grey_without_dim = np.mean(X_test[:, :, :, :], axis=2)
print(X_train_grey_without_dim.shape)
print(X_test_grey_without_dim.shape)
```

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

(32, 32, 73257)
(32, 32, 26032)
```

In [9]:

```
#Showing the grey version of the images alereadу showed
fig, ax = plt.subplots(2, 5, figsize=(9, 6))
for i, j in zip(range(10), random_images):
    ax[i//5, i%5].imshow(X_train_grey_without_dim[:, :, j], cmap='gray')
    ax[i//5, i%5].axis('off')
    ax[i//5, i%5].set_title('Class: {}'.format(y_train[j, 0]))
```



In [10]:

```
#Encoding labels to array format. This way we can use categorical crossentropy
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(sparse=False)

ohe.fit(y_train)
y_train = ohe.transform(y_train)
y_test = ohe.transform(y_test)

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

```
(73257, 10)
(26032, 10)
```

In [11]:

```
#Fixing the dimensions to be able to use the arrays on the model.fit method later
X_train = np.moveaxis(X_train, -1, 0)
X_test = np.moveaxis(X_test, -1, 0)

print(X_train.shape)
print(X_test.shape)
```

```
(73257, 32, 32, 3)
(26032, 32, 32, 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 [12]:

```
#Creating, compiling and analysing the model
from tensorflow.keras import Sequential, optimizers
from tensorflow.keras.layers import Flatten, Dense

model = Sequential([
    Flatten(input_shape=X_train.shape[1:]),
    Dense(512, activation='relu'),
    Dense(256, activation='relu'),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

opt = optimizers.Adam(learning_rate=1e-3)

model.compile(optimizer=opt,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1573376
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_2 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_3 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_4 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_5 (Dense)              (None, 10)                650
=================================================================
Total params: 1,812,298
Trainable params: 1,812,298
Non-trainable params: 0
_________________________________________________________________
```

In [13]:

```
#Creating callback functions and fitting the data with the model
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

early_stopping = EarlyStopping(monitor='val_loss', min_delta=1e-5,
                               patience=3, verbose=2)

best_check = ModelCheckpoint('best_checkpoint', monitor='val_loss', verbose=2,
                               save_best_only=True, save_weights_only=True)


history = model.fit(x=X_train, y=y_train, epochs=20, batch_size=128,
          verbose=2, validation_split=0.15, callbacks=[early_stopping, best_check])
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/20

Epoch 00001: val_loss improved from inf to 2.50603, saving model to best_checkpoint
62268/62268 - 66s - loss: 16.2051 - accuracy: 0.1435 - val_loss: 2.5060 - val_accuracy: 0.1926
Epoch 2/20

Epoch 00002: val_loss improved from 2.50603 to 2.06870, saving model to best_checkpoint
62268/62268 - 62s - loss: 2.1772 - accuracy: 0.2596 - val_loss: 2.0687 - val_accuracy: 0.3097
Epoch 3/20

Epoch 00003: val_loss improved from 2.06870 to 1.53449, saving model to best_checkpoint
62268/62268 - 62s - loss: 1.7221 - accuracy: 0.4291 - val_loss: 1.5345 - val_accuracy: 0.5056
Epoch 4/20

Epoch 00004: val_loss improved from 1.53449 to 1.26713, saving model to best_checkpoint
62268/62268 - 62s - loss: 1.4035 - accuracy: 0.5538 - val_loss: 1.2671 - val_accuracy: 0.6021
Epoch 5/20

Epoch 00005: val_loss improved from 1.26713 to 1.18863, saving model to best_checkpoint
62268/62268 - 62s - loss: 1.3219 - accuracy: 0.5895 - val_loss: 1.1886 - val_accuracy: 0.6364
Epoch 6/20

Epoch 00006: val_loss improved from 1.18863 to 1.18807, saving model to best_checkpoint
62268/62268 - 62s - loss: 1.2084 - accuracy: 0.6276 - val_loss: 1.1881 - val_accuracy: 0.6377
Epoch 7/20

Epoch 00007: val_loss improved from 1.18807 to 1.16592, saving model to best_checkpoint
62268/62268 - 62s - loss: 1.1460 - accuracy: 0.6464 - val_loss: 1.1659 - val_accuracy: 0.6550
Epoch 8/20

Epoch 00008: val_loss improved from 1.16592 to 1.04891, saving model to best_checkpoint
62268/62268 - 62s - loss: 1.1189 - accuracy: 0.6582 - val_loss: 1.0489 - val_accuracy: 0.6740
Epoch 9/20

Epoch 00009: val_loss did not improve from 1.04891
62268/62268 - 62s - loss: 1.0997 - accuracy: 0.6618 - val_loss: 1.1346 - val_accuracy: 0.6451
Epoch 10/20

Epoch 00010: val_loss improved from 1.04891 to 1.03209, saving model to best_checkpoint
62268/62268 - 60s - loss: 1.0446 - accuracy: 0.6782 - val_loss: 1.0321 - val_accuracy: 0.6779
Epoch 11/20

Epoch 00011: val_loss improved from 1.03209 to 1.00250, saving model to best_checkpoint
62268/62268 - 62s - loss: 1.0095 - accuracy: 0.6883 - val_loss: 1.0025 - val_accuracy: 0.6906
Epoch 12/20

Epoch 00012: val_loss improved from 1.00250 to 0.94481, saving model to best_checkpoint
62268/62268 - 62s - loss: 0.9679 - accuracy: 0.7040 - val_loss: 0.9448 - val_accuracy: 0.7175
Epoch 13/20

Epoch 00013: val_loss did not improve from 0.94481
62268/62268 - 60s - loss: 0.9568 - accuracy: 0.7070 - val_loss: 1.0008 - val_accuracy: 0.6899
Epoch 14/20

Epoch 00014: val_loss improved from 0.94481 to 0.92803, saving model to best_checkpoint
62268/62268 - 59s - loss: 0.9336 - accuracy: 0.7122 - val_loss: 0.9280 - val_accuracy: 0.7141
Epoch 15/20

Epoch 00015: val_loss improved from 0.92803 to 0.91250, saving model to best_checkpoint
62268/62268 - 61s - loss: 0.8981 - accuracy: 0.7265 - val_loss: 0.9125 - val_accuracy: 0.7214
Epoch 16/20

Epoch 00016: val_loss did not improve from 0.91250
62268/62268 - 63s - loss: 0.8942 - accuracy: 0.7257 - val_loss: 0.9963 - val_accuracy: 0.6983
Epoch 17/20

Epoch 00017: val_loss did not improve from 0.91250
62268/62268 - 61s - loss: 0.8691 - accuracy: 0.7336 - val_loss: 0.9179 - val_accuracy: 0.7235
Epoch 18/20

Epoch 00018: val_loss did not improve from 0.91250
62268/62268 - 61s - loss: 0.8707 - accuracy: 0.7340 - val_loss: 0.9197 - val_accuracy: 0.7162
Epoch 00018: early stopping
```

In [14]:

```
#Checking the creation of the best_checkpoint file
! ls
```

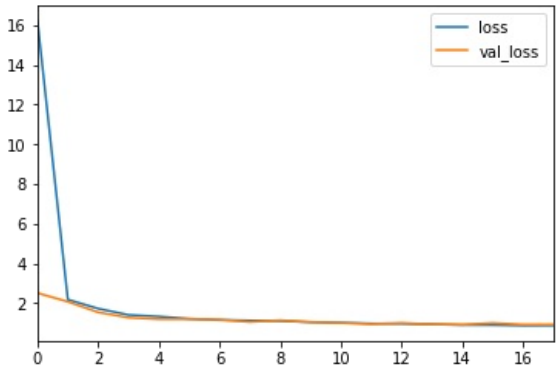
```
 best_checkpoint                         best_checkpoint.index
 best_checkpoint_2                      'Capstone Project.ipynb'
 best_checkpoint_2.data-00000-of-00001   checkpoint
 best_checkpoint_2.index                 data
 best_checkpoint.data-00000-of-00001
```

In [15]:

```
#Plotting the loss of the training and validating sets to look for some overfitting pattern
import pandas as pd

df = pd.DataFrame(history.history)
df.plot(y=['loss', 'val_loss'])
```

Out[15]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f916fc9d2e8>
```

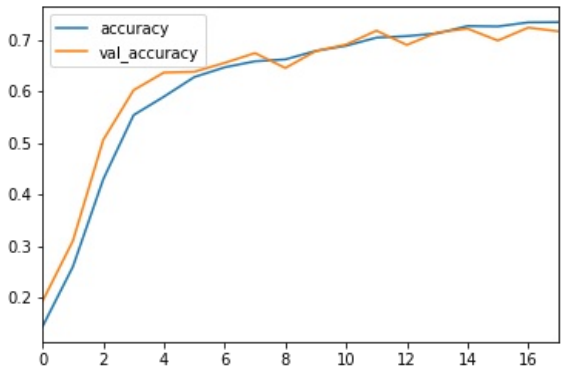


In [16]:

```
#Plotting the accuracy of the training and validating sets to look for some overfitting pattern
df.plot(y=['accuracy', 'val_accuracy'])
```

Out[16]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f90b47d7940>
```

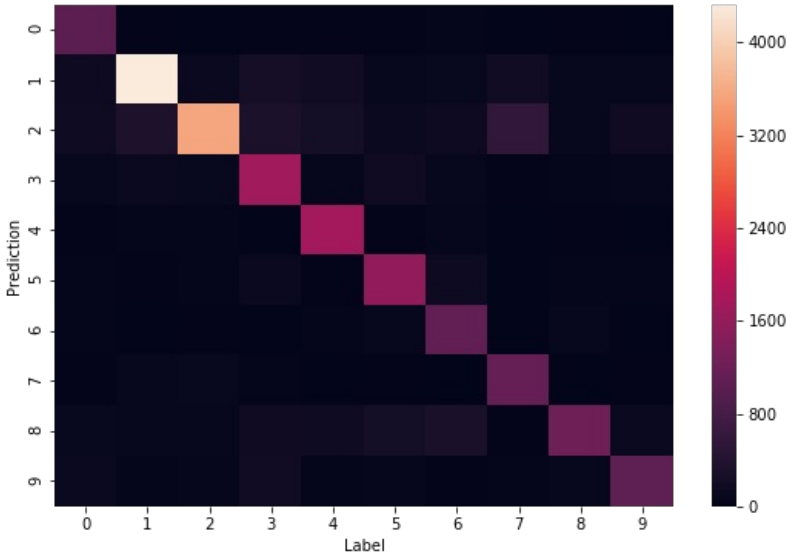


In [17]:

```
#Plotting the confusion matrix to visualize the model quality
import seaborn as sns

plt.figure(figsize=(9, 6))
sns.heatmap(tf.math.confusion_matrix(np.argmax(model.predict(X_test), axis=1), np.argmax(y_test, axis=1)))
plt.xlabel('Label')
plt.ylabel('Prediction')
```

Out[17]:

```
Text(60.0, 0.5, 'Prediction')
```



In [18]:

```
#Measuring test loss and accuracy of the model
loss, accuracy = model.evaluate(x=X_test, y=y_test, batch_size=128, verbose=2)
print('Test Loss and Test Accuracy for the model: {}, {}'.format(loss, accuracy))
```

```
26032/1 - 10s - loss: 1.0228 - accuracy: 0.7096
Test Loss and Test Accuracy for the model: 1.001450748112426, 0.7095881700515747
```

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 [19]:

```
#Creating, compiling and analysing the model
from tensorflow.keras.layers import Conv2D, MaxPool2D, BatchNormalization, Dropout

model_2 = Sequential([
    Conv2D(16, kernel_size=(3, 3), input_shape=X_train.shape[1:], activation='relu'),
    BatchNormalization(),
    MaxPool2D(pool_size=(2, 2)),
    Conv2D(16, kernel_size=(3, 3), activation='relu'),
    BatchNormalization(),
    MaxPool2D(pool_size=(2, 2)),
    Conv2D(16, kernel_size=(3, 3), activation='relu'),
    Dropout(0.2),
    Flatten(),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

model_2.compile(optimizer=opt,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model_2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 16)        448
_________________________________________________________________
batch_normalization (BatchNo (None, 30, 30, 16)        64
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 16)        2320
_________________________________________________________________
batch_normalization_1 (Batch (None, 13, 13, 16)        64
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 16)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 16)          2320
_________________________________________________________________
dropout (Dropout)            (None, 4, 4, 16)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_7 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_8 (Dense)              (None, 10)                1290
=================================================================
Total params: 105,194
Trainable params: 105,130
Non-trainable params: 64
_________________________________________________________________
```

In [20]:

```
#Creating callback functions and fitting the data with the model
best_check_2 = ModelCheckpoint('best_checkpoint_2', monitor='val_loss', verbose=2,
                               save_best_only=True, save_weights_only=True)

history_2 = model_2.fit(x=X_train, y=y_train, epochs=10, batch_size=128,
          verbose=2, validation_split=0.15, callbacks=[early_stopping, best_check_2])
```

```
Train on 62268 samples, validate on 10989 samples
Epoch 1/10

Epoch 00001: val_loss improved from inf to 0.47640, saving model to best_checkpoint_2
62268/62268 - 233s - loss: 0.8309 - accuracy: 0.7322 - val_loss: 0.4764 - val_accuracy: 0.8607
Epoch 2/10

Epoch 00002: val_loss improved from 0.47640 to 0.40418, saving model to best_checkpoint_2
62268/62268 - 229s - loss: 0.4654 - accuracy: 0.8575 - val_loss: 0.4042 - val_accuracy: 0.8799
Epoch 3/10

Epoch 00003: val_loss improved from 0.40418 to 0.39189, saving model to best_checkpoint_2
62268/62268 - 230s - loss: 0.3984 - accuracy: 0.8774 - val_loss: 0.3919 - val_accuracy: 0.8815
Epoch 4/10

Epoch 00004: val_loss improved from 0.39189 to 0.35151, saving model to best_checkpoint_2
62268/62268 - 226s - loss: 0.3561 - accuracy: 0.8902 - val_loss: 0.3515 - val_accuracy: 0.8964
Epoch 5/10

Epoch 00005: val_loss improved from 0.35151 to 0.34783, saving model to best_checkpoint_2
62268/62268 - 226s - loss: 0.3342 - accuracy: 0.8968 - val_loss: 0.3478 - val_accuracy: 0.8964
Epoch 6/10

Epoch 00006: val_loss did not improve from 0.34783
62268/62268 - 226s - loss: 0.3145 - accuracy: 0.9032 - val_loss: 0.3584 - val_accuracy: 0.8942
Epoch 7/10

Epoch 00007: val_loss improved from 0.34783 to 0.32151, saving model to best_checkpoint_2
62268/62268 - 228s - loss: 0.2967 - accuracy: 0.9087 - val_loss: 0.3215 - val_accuracy: 0.9082
Epoch 8/10

Epoch 00008: val_loss did not improve from 0.32151
62268/62268 - 228s - loss: 0.2795 - accuracy: 0.9137 - val_loss: 0.3411 - val_accuracy: 0.8992
Epoch 9/10

Epoch 00009: val_loss did not improve from 0.32151
62268/62268 - 228s - loss: 0.2683 - accuracy: 0.9160 - val_loss: 0.3417 - val_accuracy: 0.9014
Epoch 10/10

Epoch 00010: val_loss did not improve from 0.32151
62268/62268 - 227s - loss: 0.2534 - accuracy: 0.9213 - val_loss: 0.3234 - val_accuracy: 0.9091
Epoch 00010: early stopping
```

In [21]:

```
#Checking the creation of the best_checkpoint_2 file
! ls
```

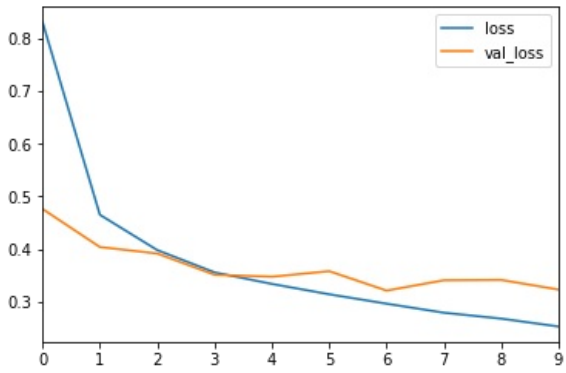
```
 best_checkpoint                        best_checkpoint.index
 best_checkpoint_2                     'Capstone Project.ipynb'
 best_checkpoint_2.data-00000-of-00001  checkpoint
 best_checkpoint_2.index                data
 best_checkpoint.data-00000-of-00001
```

In [22]:

```
#Plotting the loss of the training and validating sets to look for some overfitting pattern
df_2 = pd.DataFrame(history_2.history)
df_2.plot(y=['loss', 'val_loss'])
```

Out[22]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8f0c636278>
```

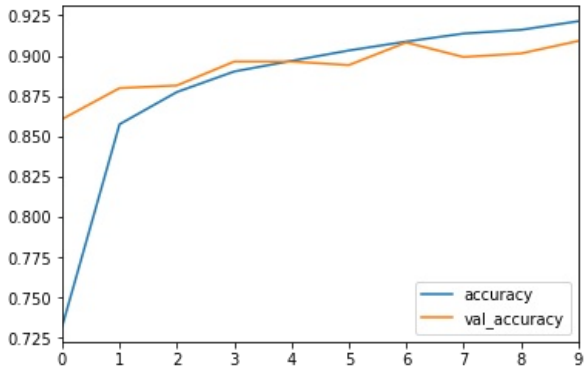


In [23]:

```
#Plotting the accuracy of the training and validating sets to look for some overfitting pattern
df_2.plot(y=['accuracy', 'val_accuracy'])
```

Out[23]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f90d4d4dd30>
```

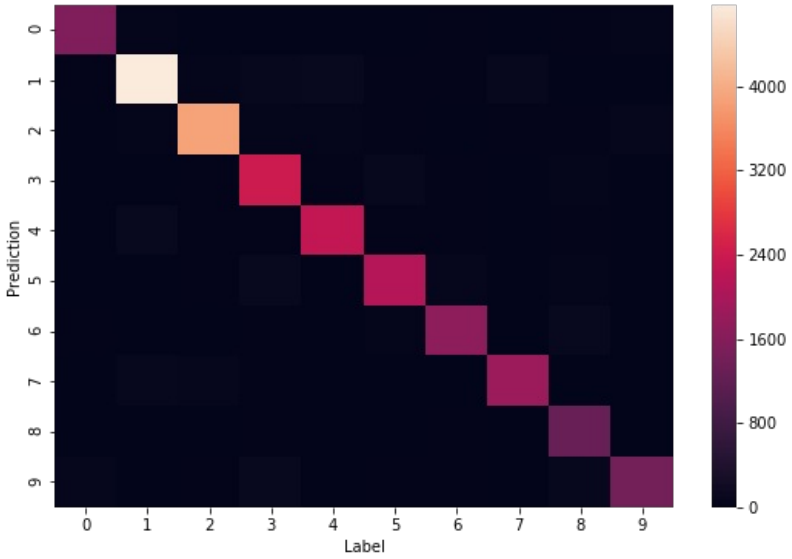


In [24]:

```
#Plotting the confusion matrix to visualize the model quality
plt.figure(figsize=(9, 6))
sns.heatmap(tf.math.confusion_matrix(np.argmax(model_2.predict(X_test.astype('float32')), axis=1), np.argmax(y_te
st, axis=1)))
plt.xlabel('Label')
plt.ylabel('Prediction')
```

Out[24]:

```
Text(60.0, 0.5, 'Prediction')
```



In [25]:

```
#Measuring test loss and accuracy of the model
loss, accuracy = model_2.evaluate(x=X_test, y=y_test, batch_size=128, verbose=2)
print('Test Loss and Test Accuracy for the model: {}, {}'.format(loss, accuracy))
```

```
26032/1 - 28s - loss: 0.3561 - accuracy: 0.8955
Test Loss and Test Accuracy for the model: 0.372380012684097, 0.8954747915267944
```

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 [26]:

```
#Reloading weights to the models
model.load_weights('./best_checkpoint')
model_2.load_weights('./best_checkpoint_2')
```

Out[26]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f8f0c2e3908>
```

In [27]:

```
#Selecting random examples from the test set, displaying the images, printing the true label, the label predicted
#by the first model, the label predicted by the second model and finally plotting the models distribution of labe
ls
#for each example
random_images = np.random.randint(y_test.shape[0], size=5)

labels = np.argmax(y_test[random_images, :], axis=1)

model_predictions = model.predict(X_test[random_images, :, :, :])
model_2_predictions = model_2.predict(X_test[random_images, :, :, :].astype('float32'))

model_labels = np.argmax(model_predictions, axis=1)
model_2_labels = np.argmax(model_2_predictions, axis=1)

fig, ax = plt.subplots(3, 5, figsize=(20, 16))
for i, j in zip(range(5), random_images):
    ax[0, i%5].imshow(X_test[j, :, :, :])
    ax[0, i%5].axis('off')
    ax[0, i%5].set_title('Class: {}; MLP: {}; CNN: {}'.format(labels[i%5], model_labels[i%5],
                                                              model_2_labels[i%5]))

    ax[1, i%5].bar(range(10), model_predictions[i%5])
    ax[1, i%5].axis('on')
    ax[1, i%5].set_title('MLP distribution')

    ax[2, i%5].bar(range(10), model_2_predictions[i%5])
    ax[2, i%5].axis('on')
    ax[2, i%5].set_title('CNN distribution')
```