Capstone Project

April 8, 2025

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
```

 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.

First, let's extract training images and labels. We note that $train["X"]$ as shape $(32, 32, 3, 73257)$ which is not convinient. So we will use the numpy *transpose* method to swap axes so that the number of samples come the first at the index

```
In [3]: X_train = train["X"].transpose(-1, 0,1,2)
        y_train = train["y"]
        X_train.shape
```

```
Out[3]: (73257, 32, 32, 3)
```

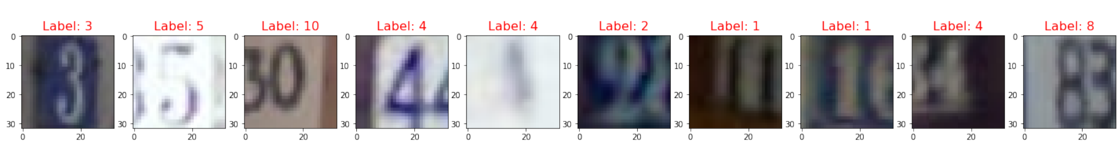
Then we do the same for the test dataset

```
In [4]: X_test = test["X"].transpose(-1, 0,1,2)
        y_test = test["y"]
        X_test.shape
```

```
Out[4]: (26032, 32, 32, 3)
```

Randomly selecting *n_im* images and corresponding labels and display them

```
In [5]: # import matplotlib
        import matplotlib.pyplot as plt
        import numpy as np
        %matplotlib inline

        def random_display_nimg(img:np.ndarray, labels:np.ndarray, n_img:int, seed:int=100, gra
            """
            This function display n_img randomly selected and
            their corresponding labels
            """
            np.random.seed(seed)
            samples = np.random.choice(np.arange(img.shape[0]), size=n_img)

            # create a figure and sho image within it
            fig, ax = plt.subplots(ncols=n_img, figsize=(25,20))
            if gray_scale:
                for i in range(n_img):
                    ax[i].imshow(img[samples[i], :, :, 0], cmap="gray")
                    ax[i].set_title(f"Label: {labels[samples[i], 0]}", color="r", fontsize=16)
            else:
                for i in range(n_img):
                    ax[i].imshow(img[samples[i]])
                    ax[i].set_title(f"Label: {labels[samples[i], 0]}", color="r", fontsize=16)
            # return samples if condition verified
            if ret_sample:
                return samples

        random_display_nimg(img=X_train, labels=y_train, n_img=10, seed=0)
```



We can see from the above image that images with 0 are labeled 10. Since in tensorflow, using softmax labels range between 0 and n_{class} (the number of class). We will replace 10 by 0. Otherwise our model will return error during training

```
In [6]: # for training targets
        y_train = np.where(y_train == 10, 0, y_train)
        # for test targets
        y_test = np.where(y_test == 10, 0, y_test)
        print(np.unique(y_train))
```

```
[0 1 2 3 4 5 6 7 8 9]
```

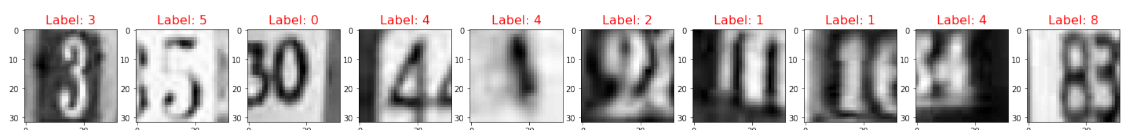
Converting training and test images to grayscale

```
In [7]: # for training images
        X_train = X_train.mean(axis=-1, keepdims=True)
        print(X_train.shape)
        #for test images
        X_test = X_test.mean(axis=-1, keepdims=True)
```

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

Displaying random selected grayscale images

```
In [8]: random_display_nimg(X_train, y_train, n_img=10, seed=0, gray_scale=True)
```



Let's normalized images before getting them to feed our ML NN

```
In [9]: X_train /= 255.
        X_test /= 255.
```

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.

First, let's import important libraries and functions

```
In [15]: from tensorflow.keras.models import Sequential
         from tensorflow.keras.layers import Dense, Flatten
         from tensorflow.keras.optimizers import Adam
```

```
In [11]: !rm -r check_point_best_only/
```

designing the model

```
In [29]: def get_mlp_model():
             model = Sequential([
                 Flatten(input_shape=(32, 32, 1)),
                 Dense(180, "relu", name="layer_1"),
                 Dense(180, "relu", name="layer_2"),
                 Dense(86, "relu", name="layer_3"),
                 Dense(10, "softmax", name="layer_out")
             ])
             return model
```

```
In [30]: model = get_mlp_model()
         model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
layer_1 (Dense)              (None, 180)               184500
_________________________________________________________________
layer_2 (Dense)              (None, 180)               32580
_________________________________________________________________
layer_3 (Dense)              (None, 86)                15566
_________________________________________________________________
layer_out (Dense)            (None, 10)                870
=================================================================
Total params: 233,516
Trainable params: 233,516
```

```
Non-trainable params: 0
_________________________________________________________________
```

Compile and train the model

```
In [31]: model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accu
```

Here I have chosen to use earlystopping and modelCheckpointCallback

```
In [32]: from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
```

- This checkpoint save the weights that give the best accuracy on the validation set doing this operation at each epoch

```
In [33]: checkpoint_path="check_point_best_only/checkpoint"
         checkpoint = ModelCheckpoint(filepath=checkpoint_path, save_best_only=True, save_freq=
                                      save_weights_only=True, monitor="val_accuracy")
```

- this second checkpoint stop the running whenever the accuracy does not improve much one 3 iteration

```
In [34]: early_stop = EarlyStopping(monitor="val_accuracy", patience=5)
```

```
In [35]: history = model.fit(X_train, y_train, epochs=30, batch_size=128, validation_split=0.09
                             callbacks=[checkpoint, early_stop], verbose=2)
```

```
Train on 66663 samples, validate on 6594 samples
Epoch 1/30
66663/66663 - 29s - loss: 2.0349 - accuracy: 0.2738 - val_loss: 1.5901 - val_accuracy: 0.4474
Epoch 2/30
66663/66663 - 18s - loss: 1.3616 - accuracy: 0.5485 - val_loss: 1.1958 - val_accuracy: 0.6194
Epoch 3/30
66663/66663 - 19s - loss: 1.1332 - accuracy: 0.6436 - val_loss: 1.0476 - val_accuracy: 0.6685
Epoch 4/30
66663/66663 - 18s - loss: 1.0277 - accuracy: 0.6822 - val_loss: 0.9836 - val_accuracy: 0.6912
Epoch 5/30
66663/66663 - 18s - loss: 0.9590 - accuracy: 0.7042 - val_loss: 0.9443 - val_accuracy: 0.6993
Epoch 6/30
66663/66663 - 18s - loss: 0.9055 - accuracy: 0.7227 - val_loss: 0.9029 - val_accuracy: 0.7191
Epoch 7/30
66663/66663 - 18s - loss: 0.8635 - accuracy: 0.7348 - val_loss: 0.8084 - val_accuracy: 0.7481
Epoch 8/30
66663/66663 - 18s - loss: 0.8311 - accuracy: 0.7441 - val_loss: 0.8175 - val_accuracy: 0.7433
Epoch 9/30
66663/66663 - 18s - loss: 0.8033 - accuracy: 0.7525 - val_loss: 0.7923 - val_accuracy: 0.7543
Epoch 10/30
66663/66663 - 18s - loss: 0.7804 - accuracy: 0.7605 - val_loss: 0.8095 - val_accuracy: 0.7516
Epoch 11/30
```

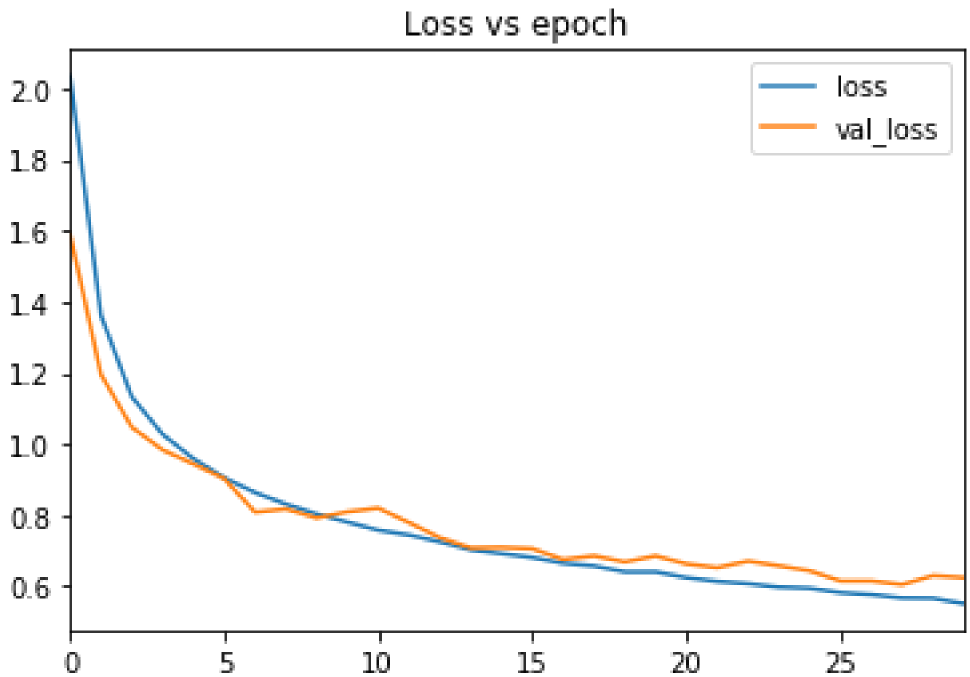
```
66663/66663 - 18s - loss: 0.7577 - accuracy: 0.7668 - val_loss: 0.8199 - val_accuracy: 0.7429
Epoch 12/30
66663/66663 - 19s - loss: 0.7441 - accuracy: 0.7702 - val_loss: 0.7789 - val_accuracy: 0.7548
Epoch 13/30
66663/66663 - 18s - loss: 0.7247 - accuracy: 0.7775 - val_loss: 0.7366 - val_accuracy: 0.7727
Epoch 14/30
66663/66663 - 18s - loss: 0.7023 - accuracy: 0.7820 - val_loss: 0.7078 - val_accuracy: 0.7809
Epoch 15/30
66663/66663 - 18s - loss: 0.6916 - accuracy: 0.7871 - val_loss: 0.7090 - val_accuracy: 0.7807
Epoch 16/30
66663/66663 - 18s - loss: 0.6809 - accuracy: 0.7895 - val_loss: 0.7057 - val_accuracy: 0.7828
Epoch 17/30
66663/66663 - 18s - loss: 0.6646 - accuracy: 0.7933 - val_loss: 0.6752 - val_accuracy: 0.7954
Epoch 18/30
66663/66663 - 18s - loss: 0.6569 - accuracy: 0.7964 - val_loss: 0.6854 - val_accuracy: 0.7901
Epoch 19/30
66663/66663 - 18s - loss: 0.6405 - accuracy: 0.8001 - val_loss: 0.6684 - val_accuracy: 0.7959
Epoch 20/30
66663/66663 - 18s - loss: 0.6405 - accuracy: 0.8009 - val_loss: 0.6852 - val_accuracy: 0.7853
Epoch 21/30
66663/66663 - 18s - loss: 0.6245 - accuracy: 0.8055 - val_loss: 0.6628 - val_accuracy: 0.8001
Epoch 22/30
66663/66663 - 18s - loss: 0.6138 - accuracy: 0.8079 - val_loss: 0.6524 - val_accuracy: 0.7995
Epoch 23/30
66663/66663 - 18s - loss: 0.6076 - accuracy: 0.8100 - val_loss: 0.6706 - val_accuracy: 0.7930
Epoch 24/30
66663/66663 - 18s - loss: 0.5977 - accuracy: 0.8132 - val_loss: 0.6574 - val_accuracy: 0.8030
Epoch 25/30
66663/66663 - 18s - loss: 0.5947 - accuracy: 0.8145 - val_loss: 0.6434 - val_accuracy: 0.8050
Epoch 26/30
66663/66663 - 18s - loss: 0.5823 - accuracy: 0.8189 - val_loss: 0.6151 - val_accuracy: 0.8145
Epoch 27/30
66663/66663 - 18s - loss: 0.5767 - accuracy: 0.8204 - val_loss: 0.6152 - val_accuracy: 0.8113
Epoch 28/30
66663/66663 - 18s - loss: 0.5674 - accuracy: 0.8235 - val_loss: 0.6051 - val_accuracy: 0.8104
Epoch 29/30
66663/66663 - 18s - loss: 0.5666 - accuracy: 0.8211 - val_loss: 0.6302 - val_accuracy: 0.8076
Epoch 30/30
66663/66663 - 18s - loss: 0.5515 - accuracy: 0.8273 - val_loss: 0.6249 - val_accuracy: 0.8097
```

```
In [36]: import pandas as pd

         df = pd.DataFrame(history.history)
```

Loss Vs epoch

```
In [37]: df.plot(y=["loss", "val_loss"], title="Loss vs epoch")
```

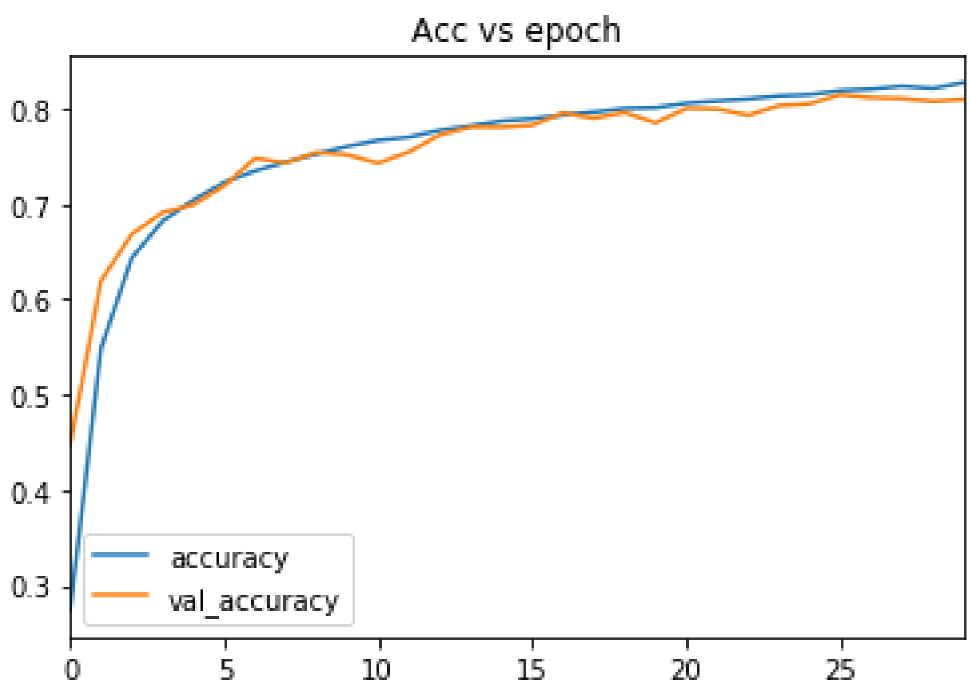
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x709b145e7358>



Accuracy Vs epoch*

```
In [38]: df.plot(y=["accuracy", "val_accuracy"], title="Acc vs epoch")
```

Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x709b14400d68>



Accuracy and loss on the test set through evaluation method

```
In [39]: loss, accuracy = model.evaluate(X_test, y_test, verbose=2)
```

```
26032/1 - 4s - loss: 0.6971 - accuracy: 0.7808
```

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.

Let's import important function and modules

```
In [12]: from tensorflow.keras.layers import Conv2D, MaxPool2D, BatchNormalization, Dropout
         from tensorflow.keras.optimizers import Adam
```

Building the model

```
In [16]: def get_cnn_model():
             model = Sequential([
                 Conv2D(32, kernel_size=3, activation="relu", padding="SAME", input_shape=(32,3
                 BatchNormalization(),
                 MaxPool2D(pool_size=3),
                 Conv2D(16, kernel_size=3, activation="relu", padding="SAME"),
                 BatchNormalization(),
                 MaxPool2D(pool_size=3),
                 #Dropout(0.5)
                 Flatten(),
                 Dense(128, "relu"),
                 Dropout(0.5),
                 Dense(128, "relu"),
                 Dense(10, "softmax")
             ])
             return model
```

```
In [17]: model = get_cnn_model()
         model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        320
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 16)        4624
_________________________________________________________________
batch_normalization_1 (Batch (None, 10, 10, 16)        64
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 3, 3, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 144)               0
_________________________________________________________________
dense (Dense)                (None, 128)               18560
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 41,498
Trainable params: 41,402
Non-trainable params: 96
_________________________________________________________________
```

we can now compile the model

```
In [18]: model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accu
```

Callbacks

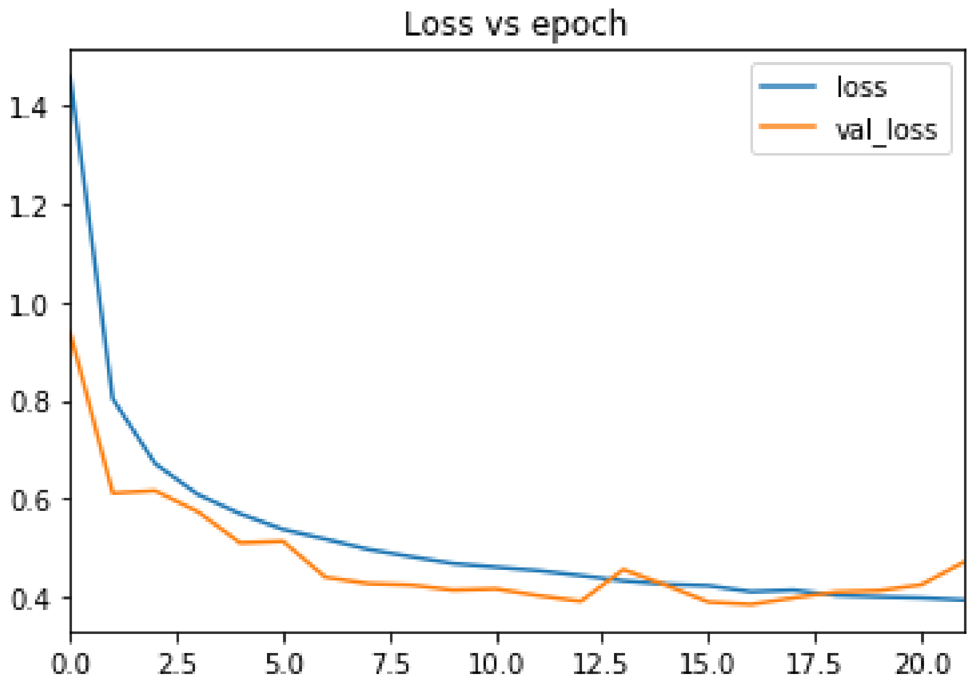
```
In [23]: # save best weights only - checkpoint
         checkpoint_path="check_point_best_only_CNN/checkpoint"
         checkpoint = ModelCheckpoint(filepath=checkpoint_path, save_best_only=True, save_freq=
                                      save_weights_only=True, monitor="val_accuracy")
         # early-stopping checkpoint
         early_stop = EarlyStopping(monitor="val_accuracy", patience=5)
```

```
In [24]: history_cnn = model.fit(X_train, y_train, epochs=30, batch_size=128, validation_split=
                                 callbacks=[checkpoint, early_stop], verbose=2)
```

```
Train on 65931 samples, validate on 7326 samples
Epoch 1/30
65931/65931 - 357s - loss: 1.4582 - accuracy: 0.5054 - val_loss: 0.9376 - val_accuracy: 0.7262
Epoch 2/30
65931/65931 - 353s - loss: 0.8032 - accuracy: 0.7432 - val_loss: 0.6115 - val_accuracy: 0.8140
Epoch 3/30
65931/65931 - 351s - loss: 0.6713 - accuracy: 0.7902 - val_loss: 0.6162 - val_accuracy: 0.8034
Epoch 4/30
65931/65931 - 355s - loss: 0.6089 - accuracy: 0.8109 - val_loss: 0.5738 - val_accuracy: 0.8189
Epoch 5/30
65931/65931 - 352s - loss: 0.5692 - accuracy: 0.8248 - val_loss: 0.5105 - val_accuracy: 0.8366
Epoch 6/30
65931/65931 - 338s - loss: 0.5379 - accuracy: 0.8353 - val_loss: 0.5139 - val_accuracy: 0.8453
Epoch 7/30
65931/65931 - 330s - loss: 0.5183 - accuracy: 0.8405 - val_loss: 0.4407 - val_accuracy: 0.8639
Epoch 8/30
65931/65931 - 338s - loss: 0.4975 - accuracy: 0.8450 - val_loss: 0.4279 - val_accuracy: 0.8718
Epoch 9/30
65931/65931 - 342s - loss: 0.4826 - accuracy: 0.8518 - val_loss: 0.4248 - val_accuracy: 0.8686
Epoch 10/30
65931/65931 - 355s - loss: 0.4686 - accuracy: 0.8560 - val_loss: 0.4146 - val_accuracy: 0.8743
Epoch 11/30
65931/65931 - 348s - loss: 0.4612 - accuracy: 0.8573 - val_loss: 0.4171 - val_accuracy: 0.8763
Epoch 12/30
65931/65931 - 347s - loss: 0.4541 - accuracy: 0.8602 - val_loss: 0.4033 - val_accuracy: 0.8735
Epoch 13/30
65931/65931 - 351s - loss: 0.4441 - accuracy: 0.8643 - val_loss: 0.3919 - val_accuracy: 0.8830
Epoch 14/30
65931/65931 - 347s - loss: 0.4334 - accuracy: 0.8660 - val_loss: 0.4563 - val_accuracy: 0.8539
Epoch 15/30
65931/65931 - 349s - loss: 0.4265 - accuracy: 0.8676 - val_loss: 0.4247 - val_accuracy: 0.8662
Epoch 16/30
65931/65931 - 348s - loss: 0.4235 - accuracy: 0.8683 - val_loss: 0.3903 - val_accuracy: 0.8803
Epoch 17/30
65931/65931 - 343s - loss: 0.4113 - accuracy: 0.8730 - val_loss: 0.3855 - val_accuracy: 0.8838
Epoch 18/30
65931/65931 - 349s - loss: 0.4145 - accuracy: 0.8731 - val_loss: 0.3988 - val_accuracy: 0.8750
Epoch 19/30
65931/65931 - 347s - loss: 0.4038 - accuracy: 0.8755 - val_loss: 0.4107 - val_accuracy: 0.8787
Epoch 20/30
65931/65931 - 347s - loss: 0.4009 - accuracy: 0.8760 - val_loss: 0.4134 - val_accuracy: 0.8687
Epoch 21/30
65931/65931 - 348s - loss: 0.3987 - accuracy: 0.8769 - val_loss: 0.4254 - val_accuracy: 0.8650
Epoch 22/30
65931/65931 - 349s - loss: 0.3945 - accuracy: 0.8778 - val_loss: 0.4715 - val_accuracy: 0.8479
```

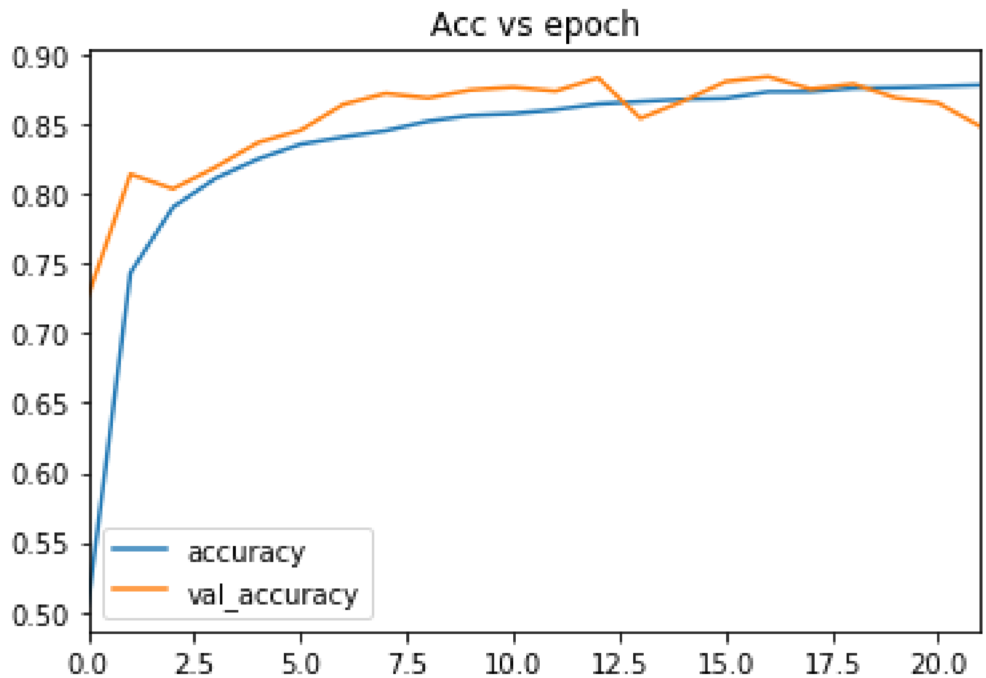
Loss vs epoch

```
In [40]: # stocking history into a dataframe
         df = pd.DataFrame(history_cnn.history)
         df.plot(y=["loss", "val_loss"], title="Loss vs epoch")

Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0x709b143a7198>
```



Accuracy vs epoch

```
In [41]: df.plot(y=["accuracy", "val_accuracy"], title="Acc vs epoch")

Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x709b143c9240>
```




```
In [49]: loss, accuracy = model_cnn.evaluate(X_test, y_test, verbose=2)
```

```
26032/1 - 46s - loss: 0.3328 - accuracy: 0.8791
```

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.

Loss and accuracy on training set

```
In [50]: # create MLP model with the same architecture as the model from which best weights we
         model_mlp = get_mlp_model()

         # create CNN model with the same architecture as the model from which best weights we
         model_cnn = get_cnn_model()
```

Load best weights for MLP and CNN

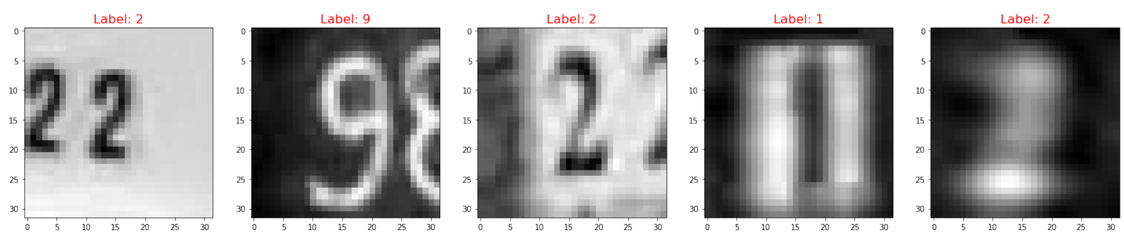
```
In [51]: # for MLP
         model_mlp.load_weights("check_point_best_only/checkpoint")
         # for CNN
         model_cnn.load_weights("check_point_best_only_CNN/checkpoint")
```

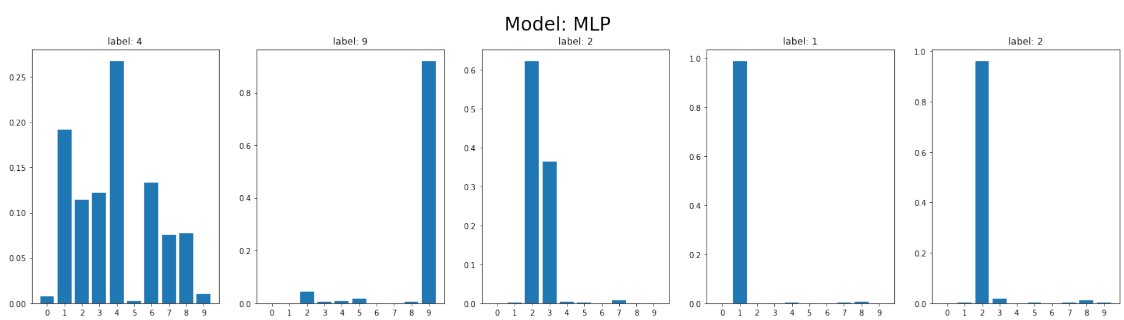
```
Out[51]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x709ad46ca400>
```

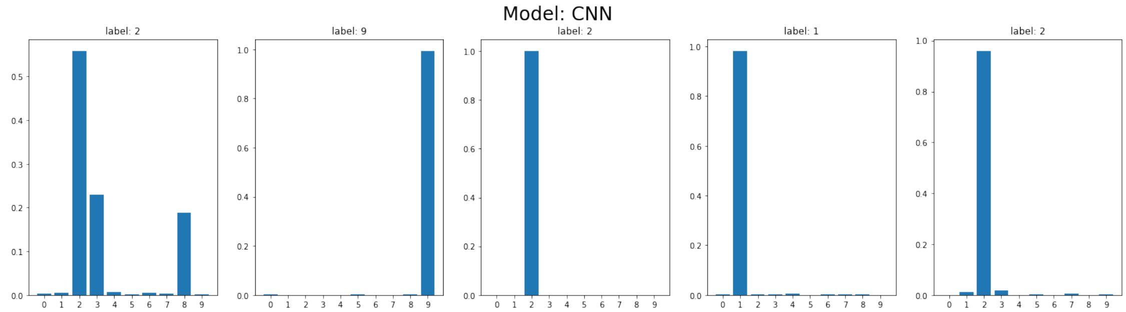
```
In [52]: def predictive__dist(predictions, title):
             """
             Bar plot of the predictive distribution
             Title: title of the figure
             """
             n_img = predictions.shape[0]
             x = np.arange(10)
             # create a figure and sho image within it
             fig, ax = plt.subplots(ncols=n_img, figsize=(25,6))

             for i in range(n_img):
                 ax[i].bar(x, height=predictions[i])
                 ax[i].set_xticks(x)
                 ax[i].set_title(f"label: {np.argmax(predictions[i])}")
             fig.suptitle(title, fontsize=24)
```

Random select and display images from test set

```
In [54]: samples = random_display_nimg(X_test, y_test, n_img=5, seed=0, ret_sample=True, gray_
         pred_mlp = model_mlp.predict(X_test[samples], verbose=False)
         pred_cnn = model_cnn.predict(X_test[samples], verbose=False)

         predictive__dist(pred_mlp, title="Model: MLP")
         predictive__dist(pred_cnn, title="Model: CNN")
```






Model: CNN
label: 2
label: 9
label: 2
label: 1
label: 2