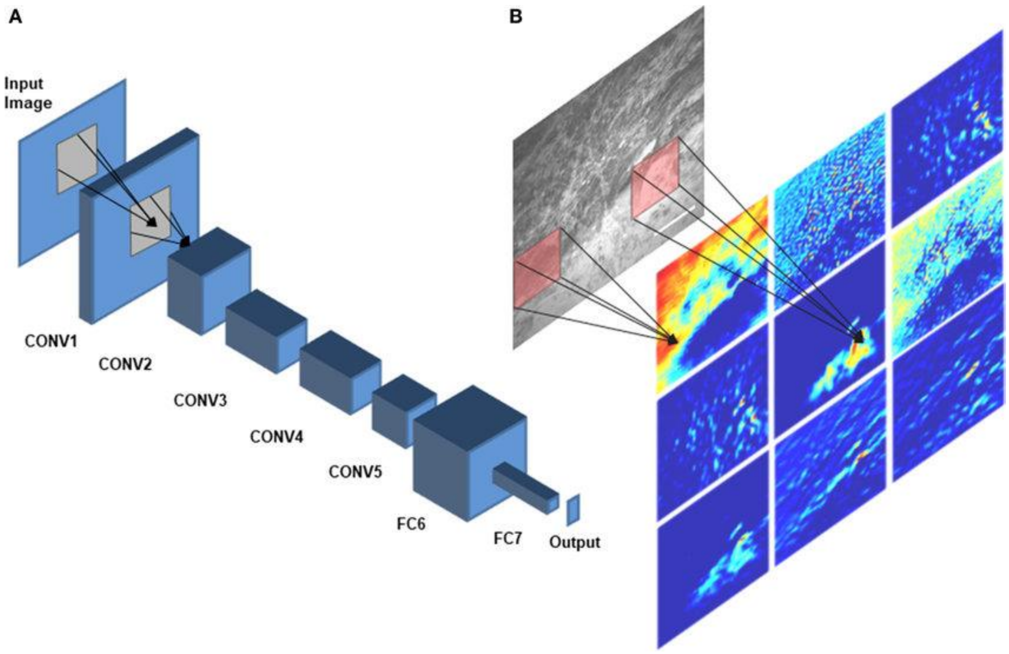
CS6005 Deep Learning Techniques

SVHN Image Classification with CNN using Keras-Based Implementation



Name	Gokul. S
Roll Number	2018103026
Batch	P
Semester	6 (RUSA)
Date of Submission	18. 04. 2021

Problem Statement: To classify images of the SVHN dataset into one of the ten output classes using convolutional neural networks. This task is more complex when compared to traditional image classification as the dataset comprises of purely natural real world images of door numbers from houses.

Dataset: The Street View House Numbers(SVHN) Dataset → (Stanford University and Google Street View Images)

Description: SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data pre - processing and formatting. It is obtained from house numbers in Google Street View images. It is basically an image digit recognition dataset of over 600,000 digit images coming from real world data. Images are cropped to 32x32.

- 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10.
- 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data
- Comes in two formats:

1. Original images with character level bounding boxes.
2. MNIST-like 32-by-32 images centred around a single character (many of the images do contain some distractors at the sides).

URL: http://ufldl.stanford.edu/housenumbers/

Code:

```
# Importing required modules and setting seed
import numpy as np
import keras
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.io import loadmat
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import confusion_matrix
from keras.preprocessing.image import ImageDataGenerator
%matplotlib inline
np.random.seed(20)
```

```python
# Loading the images ( from .mat form) and shifting axis appropriately (for training)

train_raw = loadmat('/content/drive/MyDrive/DL_Project/train_32x32.mat')
test_raw = loadmat('/content/drive/MyDrive/DL_Project/test_32x32.mat')

train_images = np.array(train_raw['X'])
test_images = np.array(test_raw['X'])

train_labels = train_raw['y']
test_labels = test_raw['y']

print(train_images.shape)
print(test_images.shape)

train_images = np.moveaxis(train_images, -1, 0)
test_images = np.moveaxis(test_images, -1, 0)

print(train_images.shape)
print(test_images.shape)

plt.imshow(train_images[12500])
plt.show()

print('Label: ', train_labels[12500])
train_images = train_images.astype('float64')
test_images = test_images.astype('float64')

train_labels = train_labels.astype('int64')
test_labels = test_labels.astype('int64')

# Rescaling images between 0 and 1

print('Min: {}, Max: {}'.format(train_images.min(), train_images.max()))

train_images /= 255.0
test_images /= 255.0

# Label encoding followed by one - hot encoding

lb = LabelBinarizer()
train_labels = lb.fit_transform(train_labels)
test_labels = lb.fit_transform(test_labels)

# Train - Test Split

X_train, X_val, y_train, y_val = train_test_split(train_images, train_labels,
test_size=0.15, random_state=22)
```

```python
# Module to find the appropriate learning rate

keras.backend.clear_session()

aux_model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), padding='same',
                           activation='relu',
                           input_shape=(32, 32, 3)),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(32, (3, 3), padding='same',
                        activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.3),

    keras.layers.Conv2D(64, (3, 3), padding='same',
                           activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(64, (3, 3), padding='same',
                        activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.3),

    keras.layers.Conv2D(128, (3, 3), padding='same',
                           activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(128, (3, 3), padding='same',
                        activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.3),

    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.4),
    keras.layers.Dense(10,  activation='softmax')
])


lr_schedule = keras.callbacks.LearningRateScheduler(
              lambda epoch: 1e-4 * 10**(epoch / 10))
optimizer = keras.optimizers.Adam(lr=1e-4, amsgrad=True)
aux_model.compile(optimizer=optimizer,
                  loss='categorical_crossentropy',
                 metrics=['accuracy'])

history = aux_model.fit_generator(datagen.flow(X_train, y_train, batch_size=128),
                              epochs=30, validation_data=(X_val, y_val),
                              callbacks=[lr_schedule])
```

```python
plt.semilogx(history.history['lr'], history.history['loss'])
plt.axis([1e-4, 1e-1, 0, 4])
plt.xlabel('Learning Rate')
plt.ylabel('Training Loss')
plt.show()


# Defining the final CNN architecture with the best learning rate found previously

keras.backend.clear_session()

model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), padding='same',
                           activation='relu',
                           input_shape=(32, 32, 3)),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(32, (3, 3), padding='same',
                        activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.3),

    keras.layers.Conv2D(64, (3, 3), padding='same',
                           activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(64, (3, 3), padding='same',
                        activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.3),

    keras.layers.Conv2D(128, (3, 3), padding='same',
                           activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(128, (3, 3), padding='same',
                        activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.3),

    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.4),
    keras.layers.Dense(10,  activation='softmax')
])

early_stopping = keras.callbacks.EarlyStopping(patience=8)
optimizer = keras.optimizers.Adam(lr=1e-3, amsgrad=True)
model_checkpoint = keras.callbacks.ModelCheckpoint(
                   'cnn.h5',
                   save_best_only=True)
```

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

model.summary()

# Final training for 70 epochs with early stopping to prevent overfitting

history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=128),
                              epochs=70, validation_data=(X_val, y_val),
                              callbacks=[early_stopping, model_checkpoint])

# Retrieving and plotting metrics

train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

train_loss = history.history['loss']
val_loss = history.history['val_loss']

plt.figure(figsize=(20, 10))

plt.subplot(1, 2, 1)
plt.plot(train_acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.legend()
plt.title('Epochs vs. Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(train_loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.legend()
plt.title('Epochs vs. Training and Validation Loss')

plt.show()

# Finding the final test metrics and printing the overall performance of the model

test_loss, test_acc = model.evaluate(x=test_images, y=test_labels, verbose=0)
print('Test accuracy is: {:0.4f} \nTest loss is: {:0.4f}'.format(test_acc,
test_loss))
```

Modules:

1) Loading and pre-processing (scaling) training dataset.
2) One – hot encoding the labels followed by train – test split.
3) Finding the best learning rate by optimizing it.
4) Training the CNN with the defined architecture and the most optimal learning rate.
5) Retrieve the corresponding metrics post training and plot their graphs (to make sure overfitting has not occurred).
6) Find out the overall test accuracy and test loss.

P.T.O

CNN Model Summary:

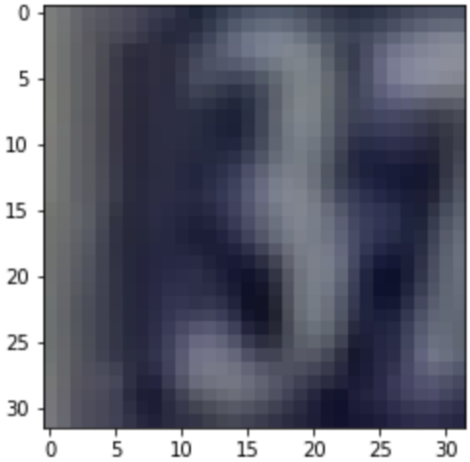
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_2 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 128)               262272
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 551,466
Trainable params: 551,018
Non-trainable params: 448
_________________________________________________________________
```

Coding/Output Snapshots:

- **Sample training image:**

```
plt.imshow(train_images[12500])
plt.show()

print('Label: ', train_labels[12500])
```




```
Label:  [3]
```

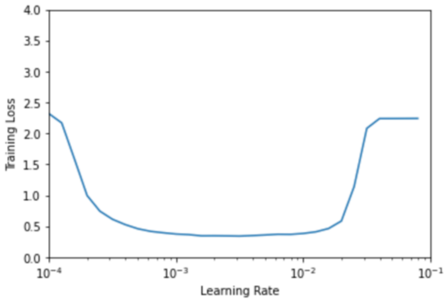
- **Performing image scaling:**

```
[10] print('Min: {}, Max: {}'.format(train_images.min(), train_images.max()))

     train_images /= 255.0
     test_images /= 255.0
```

```
Min: 0.0, Max: 255.0
```

- **Finding the appropriate learning rate from the graph after implementing a variation of the elbow method (by observing the graph)**

```
[17] plt.semilogx(history.history['lr'], history.history['loss'])
     plt.axis([1e-4, 1e-1, 0, 4])
     plt.xlabel('Learning Rate')
     plt.ylabel('Training Loss')
     plt.show()
```



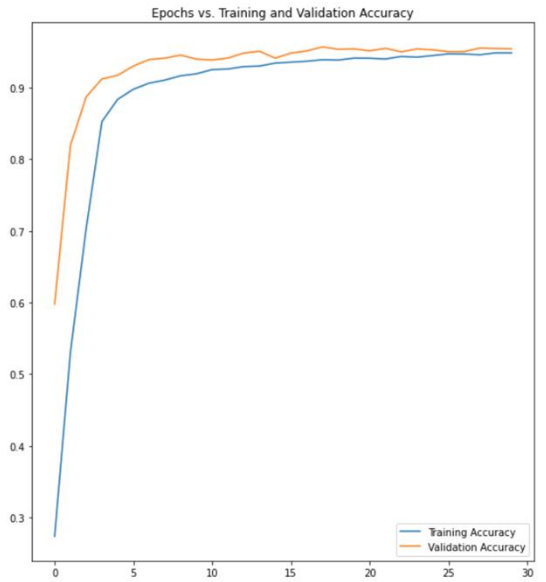
- **Final training with the most appropriate learning rate found from the previous step (10^-3) (with early stopping)**

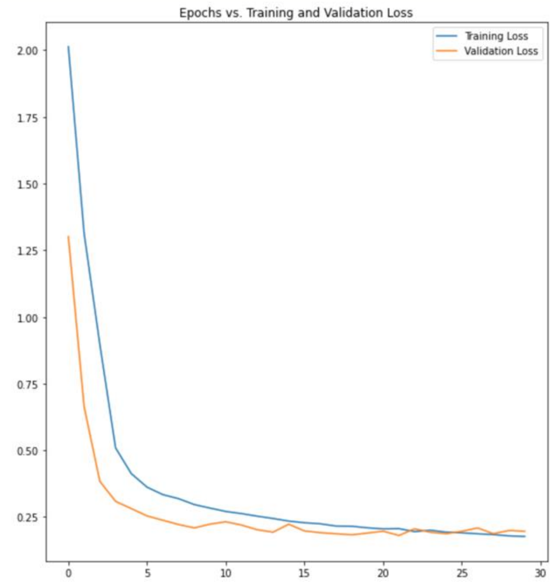
```
history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=128),
                              epochs=70, validation_data=(X_val, y_val),
                              callbacks=[early_stopping, model_checkpoint])

/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1844: UserWarning: `Model.fit_generator` is deprecated
  warnings.warn('`Model.fit_generator` is deprecated and '
Epoch 1/70
487/487 [==============================] - 33s 65ms/step - loss: 2.2482 - accuracy: 0.2040 - val_loss: 1.3011 - val_accuracy: 0.5978
Epoch 2/70
487/487 [==============================] - 31s 64ms/step - loss: 1.4070 - accuracy: 0.4935 - val_loss: 0.6633 - val_accuracy: 0.8188
Epoch 3/70
487/487 [==============================] - 32s 65ms/step - loss: 1.0346 - accuracy: 0.6464 - val_loss: 0.3834 - val_accuracy: 0.8867
Epoch 4/70
487/487 [==============================] - 32s 65ms/step - loss: 0.5449 - accuracy: 0.8401 - val_loss: 0.3083 - val_accuracy: 0.9115
Epoch 5/70
487/487 [==============================] - 32s 65ms/step - loss: 0.4223 - accuracy: 0.8803 - val_loss: 0.2811 - val_accuracy: 0.9167
Epoch 6/70
487/487 [==============================] - 32s 65ms/step - loss: 0.3681 - accuracy: 0.8953 - val_loss: 0.2536 - val_accuracy: 0.9297
Epoch 7/70
487/487 [==============================] - 32s 65ms/step - loss: 0.3409 - accuracy: 0.9042 - val_loss: 0.2376 - val_accuracy: 0.9387
Epoch 8/70
487/487 [==============================] - 32s 65ms/step - loss: 0.3159 - accuracy: 0.9100 - val_loss: 0.2215 - val_accuracy: 0.9407
Epoch 9/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2961 - accuracy: 0.9151 - val_loss: 0.2086 - val_accuracy: 0.9449
Epoch 10/70
487/487 [==============================] - 31s 65ms/step - loss: 0.2804 - accuracy: 0.9191 - val_loss: 0.2229 - val_accuracy: 0.9393
Epoch 11/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2731 - accuracy: 0.9245 - val_loss: 0.2318 - val_accuracy: 0.9381
Epoch 12/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2617 - accuracy: 0.9247 - val_loss: 0.2194 - val_accuracy: 0.9408
Epoch 13/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2515 - accuracy: 0.9281 - val_loss: 0.2019 - val_accuracy: 0.9478
Epoch 14/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2365 - accuracy: 0.9317 - val_loss: 0.1928 - val_accuracy: 0.9502
Epoch 15/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2322 - accuracy: 0.9337 - val_loss: 0.2225 - val_accuracy: 0.9408
Epoch 16/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2242 - accuracy: 0.9363 - val_loss: 0.1974 - val_accuracy: 0.9477
Epoch 17/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2204 - accuracy: 0.9371 - val_loss: 0.1909 - val_accuracy: 0.9507
Epoch 18/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2071 - accuracy: 0.9400 - val_loss: 0.1869 - val_accuracy: 0.9563
Epoch 19/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2133 - accuracy: 0.9390 - val_loss: 0.1827 - val_accuracy: 0.9530
Epoch 20/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2009 - accuracy: 0.9416 - val_loss: 0.1894 - val_accuracy: 0.9535
Epoch 21/70
487/487 [==============================] - 31s 65ms/step - loss: 0.2035 - accuracy: 0.9411 - val_loss: 0.1965 - val_accuracy: 0.9509
Epoch 22/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2028 - accuracy: 0.9402 - val_loss: 0.1804 - val_accuracy: 0.9543
Epoch 23/70
487/487 [==============================] - 32s 65ms/step - loss: 0.1927 - accuracy: 0.9438 - val_loss: 0.2053 - val_accuracy: 0.9494
Epoch 24/70
```

- **Training stops at epoch #30 due to early stopping by keras callbacks to prevent overfitting**

```
Epoch 9/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2961 - accuracy: 0.9151 - val_loss: 0.2086 - val_accuracy: 0.9449
Epoch 10/70
487/487 [==============================] - 31s 65ms/step - loss: 0.2804 - accuracy: 0.9191 - val_loss: 0.2229 - val_accuracy: 0.9393
Epoch 11/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2731 - accuracy: 0.9245 - val_loss: 0.2318 - val_accuracy: 0.9381
Epoch 12/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2617 - accuracy: 0.9247 - val_loss: 0.2194 - val_accuracy: 0.9408
Epoch 13/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2515 - accuracy: 0.9281 - val_loss: 0.2019 - val_accuracy: 0.9478
Epoch 14/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2365 - accuracy: 0.9317 - val_loss: 0.1928 - val_accuracy: 0.9502
Epoch 15/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2322 - accuracy: 0.9337 - val_loss: 0.2225 - val_accuracy: 0.9408
Epoch 16/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2242 - accuracy: 0.9363 - val_loss: 0.1974 - val_accuracy: 0.9477
Epoch 17/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2204 - accuracy: 0.9371 - val_loss: 0.1909 - val_accuracy: 0.9507
Epoch 18/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2071 - accuracy: 0.9400 - val_loss: 0.1869 - val_accuracy: 0.9563
Epoch 19/70
487/487 [==============================] - 31s 64ms/step - loss: 0.2133 - accuracy: 0.9390 - val_loss: 0.1827 - val_accuracy: 0.9530
Epoch 20/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2009 - accuracy: 0.9416 - val_loss: 0.1894 - val_accuracy: 0.9535
Epoch 21/70
487/487 [==============================] - 31s 65ms/step - loss: 0.2035 - accuracy: 0.9411 - val_loss: 0.1965 - val_accuracy: 0.9509
Epoch 22/70
487/487 [==============================] - 32s 65ms/step - loss: 0.2028 - accuracy: 0.9402 - val_loss: 0.1804 - val_accuracy: 0.9543
Epoch 23/70
487/487 [==============================] - 32s 65ms/step - loss: 0.1927 - accuracy: 0.9438 - val_loss: 0.2053 - val_accuracy: 0.9494
Epoch 24/70
487/487 [==============================] - 31s 64ms/step - loss: 0.1993 - accuracy: 0.9420 - val_loss: 0.1925 - val_accuracy: 0.9535
Epoch 25/70
487/487 [==============================] - 32s 65ms/step - loss: 0.1932 - accuracy: 0.9451 - val_loss: 0.1870 - val_accuracy: 0.9522
Epoch 26/70
487/487 [==============================] - 32s 65ms/step - loss: 0.1929 - accuracy: 0.9458 - val_loss: 0.1960 - val_accuracy: 0.9497
Epoch 27/70
487/487 [==============================] - 32s 65ms/step - loss: 0.1853 - accuracy: 0.9474 - val_loss: 0.2085 - val_accuracy: 0.9498
Epoch 28/70
487/487 [==============================] - 32s 65ms/step - loss: 0.1804 - accuracy: 0.9469 - val_loss: 0.1871 - val_accuracy: 0.9547
Epoch 29/70
487/487 [==============================] - 31s 65ms/step - loss: 0.1745 - accuracy: 0.9485 - val_loss: 0.1990 - val_accuracy: 0.9540
Epoch 30/70
487/487 [==============================] - 31s 65ms/step - loss: 0.1704 - accuracy: 0.9501 - val_loss: 0.1960 - val_accuracy: 0.9537
```

- **Overall graphs of training V/S testing accuracies and errors**





Results: The final model achieves an accuracy of 95.88 % and a loss of 0.1741 which is really good considering the naturality of the real world dataset that has been considered.

```
[23] test_loss, test_acc = model.evaluate(x=test_images, y=test_labels, verbose=0)
     print('Test accuracy is: {:0.4f} \nTest loss is: {:0.4f}'.format(test_acc, test_loss))

     Test accuracy is: 0.9588
     Test loss is: 0.1741
```

Conclusion:

CNN model has performed extremely well (around 96% accurate) compared to traditional ANNs by substantially reducing the number of parameters to be trained and capturing each and every aspect of the image onto a separate feature map.

References:

[1] https://en.wikipedia.org/wiki/Convolutional_neural_network

[2] https://faroit.com/keras-docs/1.2.0/

[3] http://ufldl.stanford.edu/housenumbers/

[4] https://medium.com/@ksusorokina/image-classification-with-convolutional-neural-networks-496815db12a8

[5] Deep Learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville