Project Part 3: Classification using Neural Networks and Deep Learning

Data Set: SVHN Dataset

Shashank Davalgi Omprakash (1219510734)

Overview:

Building and Training a small convolutional neural network for a classification task.

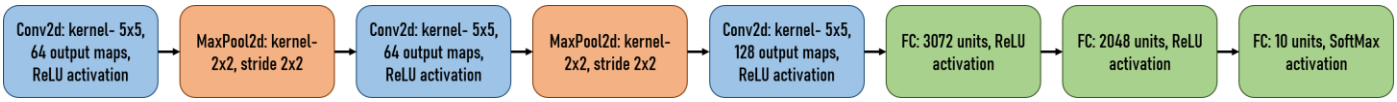
Dataset: SVHN Dataset

Dataset Description:

- It consists of printed digits cropped from the house plate pictures belonging to 10 classes.
  - Training Samples: 73,275 images
  - Testing Samples: 26,032 images
- The input image resolution is 32x32 and consists of 3 (RGB) channels.

Architecture:

The below diagram shows the CNN architecture implemented in this phase of the project.



Implementation:

1. The training and testing images are loaded along with the training and testing labels as shown below:

```
trX = io.loadmat('train_32x32.mat')['X']
trY = io.loadmat('train_32x32.mat')['y']
tsX = io.loadmat('test_32x32.mat')['X']
tsY = io.loadmat('test_32x32.mat')['y']
```

2. The training and testing images are converted to ‘float64’ type and the training and testing labels are converted into ‘int64’ type.

3. We then normalize the data to bring it in the range of 0-1. It is done as shown below:

```
training_images /= 255.0
testing_images /= 255.0
```

4. We build the model as per the architecture provided above using the Keras Sequential model.
5. The first block of the architecture is implemented as shown below:

```
model = Sequential()
model.add(Conv2D(64, (5, 5), strides=(1, 1), padding='same', activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2))) # MaxPooling size 2, 2 stride
```

   - The Sequential() function creates a Keras sequential model for 2D CNN.
   - We add a 2D CNN to the model with output feature maps as 64, kernel size as 5x5, strides as 1.
   - The padding=’same’ gives the output feature maps as the same size of input shape.
   - ReLU activation function is used for this model with input shape = (32x32x3)
6. Next, we add the MaxPooling for the model after the first block with the pool_size = 2x2 and strides = 2.
7. Similarly, the model is build based on the architecture by adding 2 more CNN layer with one more MaxPooling layer.
8. Once these layers are built, we build a Fully Connected Dense layer after the 3rd CNN layer with 3078 units and ReLU activation. Post that, we create 2 more Dense layers and output with the SoftMax activation function. It is implemented as shown below:

```
model.add(Flatten())
model.add(Dense(3072, activation='relu'))
model.add(Dense(2048, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

9. model.summary() gives the summary of the model and is shown below:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 32, 32, 64)        4864
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 64)        102464
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 128)         204928
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_3 (Dense)              (None, 3072)              25168896
_________________________________________________________________
dense_4 (Dense)              (None, 2048)              6293504
_________________________________________________________________
dense_5 (Dense)              (None, 10)                20490
=================================================================
Total params: 31,795,146
Trainable params: 31,795,146
Non-trainable params: 0
_________________________________________________________________
```

10. Then we build the model using the Stochastic Gradient Descent optimizer with a learning rate = 0.01
11. The model is then fit and validated on the training sample for 20 epochs. The implementation is :

```
network_opt = keras.optimizers.SGD(learning_rate=0.01)

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

predictor = model.fit(training_images, training_labels, epochs=20, validation_data= (testing_images, testing_labels))
```

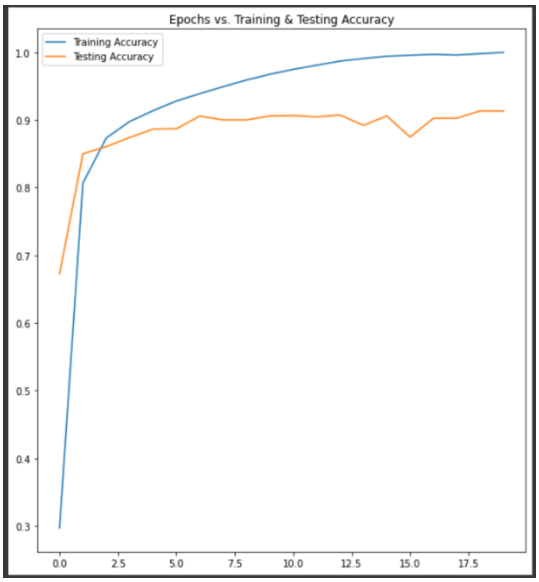
12. Once the model is executed, we find the training and testing accuracies and losses.

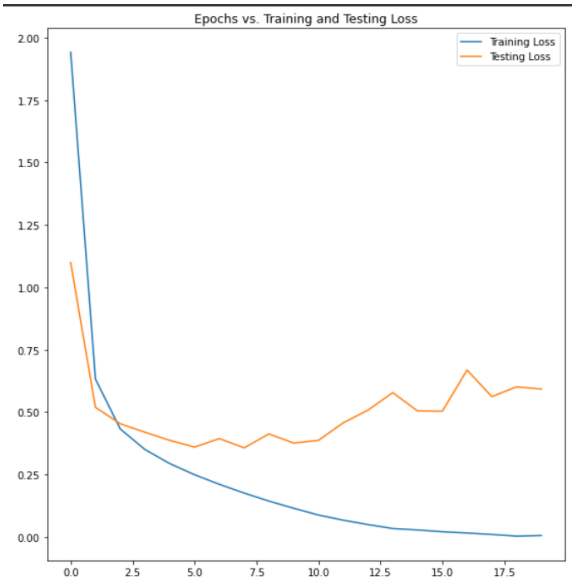
```
train_set_acc = predictor.history['accuracy']
test_set_acc = predictor.history['val_accuracy']

train_set_loss = predictor.history['loss']
test_set_loss = predictor.history['val_loss']
```

13. The output of the 20 epochs is shown below:

```
Epoch 1/20
2290/2290 [==============================] - 55s 10ms/step - loss: 2.2011 - accuracy: 0.2067 - val_loss: 1.0892 - val_accuracy: 0.6726
Epoch 2/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.7908 - accuracy: 0.7630 - val_loss: 0.5338 - val_accuracy: 0.8499
Epoch 3/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.4548 - accuracy: 0.8668 - val_loss: 0.4924 - val_accuracy: 0.8606
Epoch 4/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.3646 - accuracy: 0.8945 - val_loss: 0.4397 - val_accuracy: 0.8739
Epoch 5/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.3001 - accuracy: 0.9130 - val_loss: 0.3906 - val_accuracy: 0.8863
Epoch 6/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.2478 - accuracy: 0.9299 - val_loss: 0.4155 - val_accuracy: 0.8869
Epoch 7/20
2290/2290 [==============================] - 22s 9ms/step - loss: 0.2096 - accuracy: 0.9404 - val_loss: 0.3366 - val_accuracy: 0.9055
Epoch 8/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.1735 - accuracy: 0.9517 - val_loss: 0.3606 - val_accuracy: 0.8999
Epoch 9/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.1470 - accuracy: 0.9598 - val_loss: 0.3870 - val_accuracy: 0.8999
Epoch 10/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.1119 - accuracy: 0.9687 - val_loss: 0.3675 - val_accuracy: 0.9059
Epoch 11/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0865 - accuracy: 0.9767 - val_loss: 0.3775 - val_accuracy: 0.9063
Epoch 12/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0663 - accuracy: 0.9824 - val_loss: 0.4165 - val_accuracy: 0.9044
Epoch 13/20
2290/2290 [==============================] - 22s 9ms/step - loss: 0.0467 - accuracy: 0.9883 - val_loss: 0.4451 - val_accuracy: 0.9072
Epoch 14/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0320 - accuracy: 0.9923 - val_loss: 0.5323 - val_accuracy: 0.8919
Epoch 15/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0230 - accuracy: 0.9948 - val_loss: 0.5117 - val_accuracy: 0.9058
Epoch 16/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0157 - accuracy: 0.9964 - val_loss: 0.7739 - val_accuracy: 0.8746
Epoch 17/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0140 - accuracy: 0.9973 - val_loss: 0.5785 - val_accuracy: 0.9023
Epoch 18/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0207 - accuracy: 0.9952 - val_loss: 0.5680 - val_accuracy: 0.9024
Epoch 19/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0099 - accuracy: 0.9980 - val_loss: 0.5587 - val_accuracy: 0.9133
Epoch 20/20
2290/2290 [==============================] - 21s 9ms/step - loss: 0.0035 - accuracy: 0.9997 - val_loss: 0.5894 - val_accuracy: 0.9130
```

14. Using the results from the above functions, we plot the epochs against the training and testing accuracies and epochs against training and testing losses. The graphs are shown below:





Results and Accuracy:

The Training model accuracy is **99.93%**
The Testing model accuracy is **91.14%**

The training model loss is very negligible.
The testing model loss is **59.18%**