Date out : June 6, 2020 Due on: June 15, 2020

ENIN 880CA – Spring/Summer 2020

Exercise #4 – Deep Learning for Computer Vision – First Exercise with the CNN; Using LeNet and MNIST Dataset for Recognizing Handwritten Digits

dataIn this exercise, we explore about LeNet architecture (developed in 1998). LeNet renders itself as a seminal work in the deep learning. While seminal, LeNet is considered to be a "shallow" network, with only 2 convolutional and two fully-connected layers, in comparison to networks such as ResNet or VGG.

Students are supposed to use the heavily-processed MNIST dataset. MNIST dataset is considered as a test bench dataset for evaluating new classification algorithms. If your classification method scores lower than 95% accuracy on this dataset, something is wrong with your network (logic or algorithm).

In this exercise, students are supposed to develop a CNN-based classifier that can recognize hand-written digits with high accuracy.

Students should read chapters 11 to 14 (all inclusive) from the pdf file sent to the students' list via email.

Students should watch videos on chapters 11 to 14, shared via Dropbox.

Students should learn about the code explained in chapter 14. For instance, students should reproduce results provided in Figure 14.2.

To show how well your network can recognize the digits, write some digits on a blank piece of paper, and subject them to different lighting conditions, ranging from dark, to bright and report on the classification accuracy.

NOTE: In the next exercise, students will be tasked to use a Long Short-Term Memory (LSTM) Network to recognize spoken digits. You will be asked to record your voice when calling a certain number (in English), and show that your LSTM network can recognize it. This will give students an exposure to the Recursive Neural Networks (RNN) in general, and the LSTM in particular.

1.Theory [1]

1.1 Traditional Feedforward Neural Networks vs. Convolutional Neural Networks

In traditional feedforward neural networks, each neuron in the input layer is connected to every output neuron in the next layer – we call this a *fully connected* (FC) layer. However, in CNNs, we don't use FC layers until the *very last layer(s)* in the network. We can thus define a CNN as a neural network that swaps in a specialized "convolutional" layer in place of "fully-connected" layer for at least *one* of the layers in the network.

A nonlinear activation function, such as ReLU, is then applied to the output of these convolutions and the process of convolution => activation continues (along with a mixture of other layer types to help reduce the width and height of the input volume and help reduce overfitting) until we finally reach the end of the network and apply one or two FC layers where we can obtain our final output classifications. Each layer in a CNN applies a different set of filters, typically hundreds or thousands of them, and combines the results, feeding the output into the next layer in the network.

The last layer in a CNN uses these higher-level features to make predictions regarding the contents of the image.

In practice, CNNs give us two key benefits: *local invariance* and *compositionality*. *Local Invariance*

- **Local invariance**

Local invariance allows us to classify an image as containing a particular object *regardless* of where in the image the object appears. We obtain this local invariance through the usage of "pooling layers" (discussed later in this chapter) which identifies regions of our input volume with a high response to a particular filter.

- **Compositionality**

Each filter composes a local patch of lower-level features into a higher-level representation, similar to how we can compose a set of mathematical functions that build on the output of previous functions: $f\ (g(h(x)))$ – this composition allows our network to learn more rich features deeper in the network. For example, our network may build edges from pixels, shapes from edges, and then complex objects from shapes – all in an automated fashion that happens *naturally* during the training process. The concept of building higher-level features from lower-level ones is exactly why CNNs are so powerful in computer vision.

1.2 CNN Layer Types

There are many types of layers used to build Convolutional Neural Networks, but the ones you are most likely to encounter include:

- Convolutional (CONV)
- Activation (ACT or RELU)
- Pooling (POOL)
- Fully-connected (FC)
- Batch normalization (BN)
- Dropout (DO)

Stacking a series of these layers in a specific manner yields a CNN such as

CNN: INPUT => CONV => RELU => FC => SOFTMAX

Of these layer types, CONV and FC, (and to a lesser extent, BN) are the only layers that contain parameters that are learned during the training process.

- ### Convolutional (CONV)

The CONV layer is the core building block of a Convolutional Neural Network. The CONV layer parameters consist of a set of *K* learnable filters (i.e., "kernels"), where each filter has a width and a height, and are nearly always square.

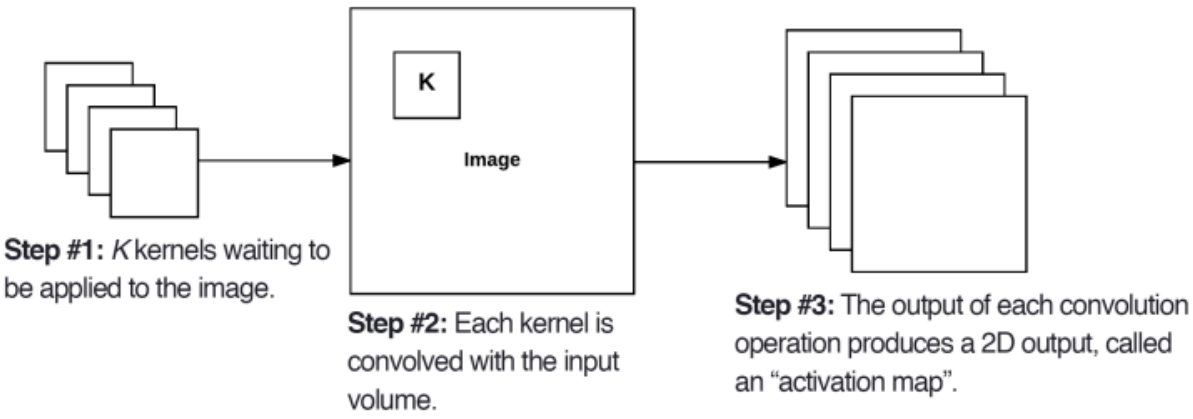


Figure 1: Convolutional (CONV) Layer

- ### Activation (ACT or RELU)

After each CONV layer in a CNN, we apply a nonlinear activation function, such as ReLU, ELU, or any of the other Leaky ReLU variants. (INPUT => CONV => ACT.)

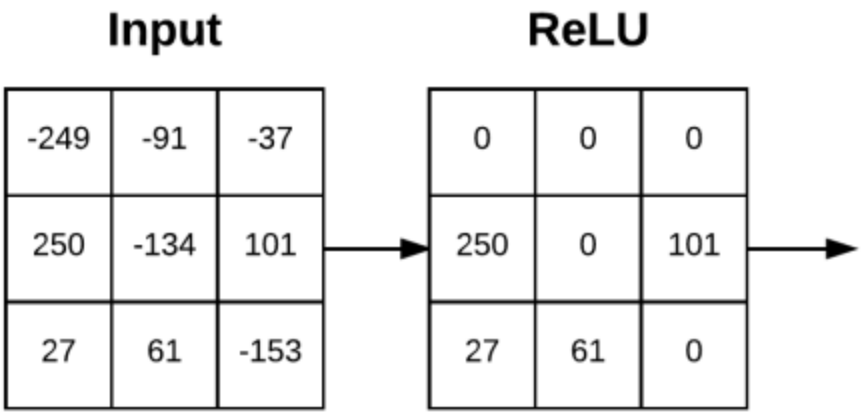


Figure 2: Applying ReLU activation to an image section

- **Pooling (POOL)**

There are two methods to reduce the size of an input volume – CONV layers with a stride > 1 and POOL layers.

It is common to insert POOL layers in-between consecutive CONV layers in a CNN architectures:
INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC
The primary function of the POOL layer is to progressively reduce the spatial size (i.e., width and height) of the input volume. Doing this allows us to reduce the amount of parameters and computation in the network – pooling also helps us control overfitting.

Pooling is done using *max* or *average* functions. Max pooling is typically done in the middle of the CNN architecture to reduce spatial size, whereas average pooling is normally used as the final layer of the network where we wish to avoid using FC layers entirely.

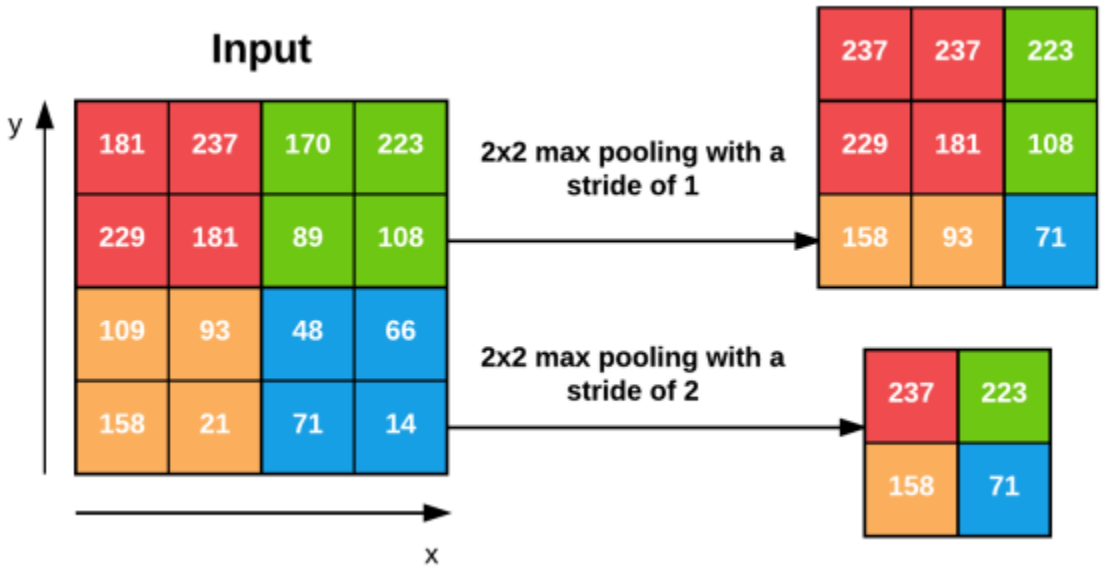


Figure 3: Applying 2×2 max pooling

- **Fully-connected (FC)**

Neurons in FC layers are fully-connected to all activations in the previous layer, as is the standard for feedforward neural networks.

FC layers are *always* placed at the end of the network. It's common to use one or two FC layers prior to applying the softmax classifier, as the following (simplified) architecture demonstrates:

INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => FC

- **Batch normalization (BN)**

As the name suggests, are used to normalize the activations of a given input volume before passing it into the next layer in the network.

According to the original paper by Ioffe and Szegedy, they placed their batch normalization (BN) before the activation:

INPUT => CONV => BN => RELU ...

However, placing the BN after the RELU yields slightly higher accuracy and lower loss:

INPUT => CONV => RELU => BN ...

- **Dropout (DO)**

Dropout is actually a form of *regularization* that aims to help prevent overfitting by increasing testing accuracy, perhaps at the expense of training accuracy.

For each mini-batch in our training set, dropout layers, with probability *p*, randomly disconnect inputs from the preceding layer to the next layer in the network architecture.

It is most common to place dropout layers with p = 0:5 in-between FC layers of an architecture where the final FC layer is assumed to be our softmax classifier:

... CONV => RELU => POOL => FC => DO => FC => DO => FC

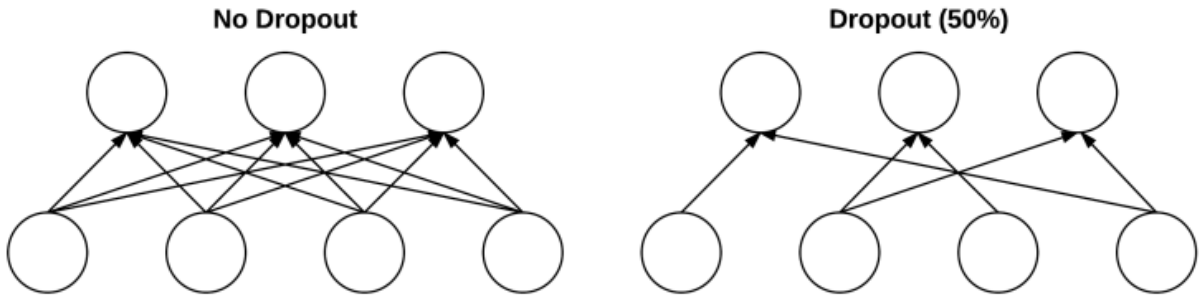


Figure 4: Drop out: randomly disconnecting (with probability p = 0.5) the connections between two FC layers

1.3 The LeNet Architecture

The LeNet network is small and easy to understand— yet large enough to provide interesting results.

The LeNet architecture consists of the following layers, using a pattern of CONV => ACT => POOL:
INPUT => CONV => TANH => POOL => CONV => TANH => POOL => FC => TANH => FC
Notice how the LeNet architecture uses the tanh activation function rather than the more popular ReLU.

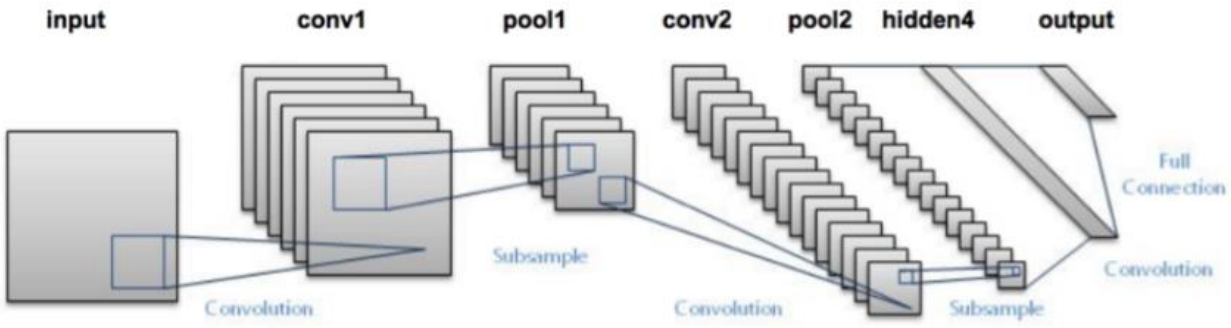


Figure 5: The LeNet architecture

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$28 \times 28 \times 1$	
CONV	$28 \times 28 \times 20$	$5 \times 5, K = 20$
ACT	$28 \times 28 \times 20$	
POOL	$14 \times 14 \times 20$	2×2
CONV	$14 \times 14 \times 50$	$5 \times 5, K = 50$
ACT	$14 \times 14 \times 50$	
POOL	$7 \times 7 \times 50$	2×2
FC	500	
ACT	500	
FC	10	
SOFTMAX	10	

Figure 6: A table summary of the LeNet architecture

2.Implementation of LeNet on MNIST [1]

Implementing the LeNet architecture on MNIST dataset using the Keras library takes following steps:

2.1 Loading the MNIST dataset

The MNIST dataset consists of 60,000 training images and 10,000 testing images (figure 7).

Figure 7: A sample of the MNIST dataset

Python Code:

```
((trainData, trainLabels), (testData, testLabels)) = mnist.load_data()
trainData = trainData.reshape((trainData.shape[0], 28, 28, 1))
testData = testData.reshape((testData.shape[0], 28, 28, 1))
trainData = trainData.astype("float32") / 255.0
testData = testData.astype("float32") / 255.0
# convert the labels from integers to vectors
le = LabelBinarizer()
trainLabels = le.fit_transform(trainLabels)
testLabels = le.transform(testLabels)
```

2.2 Instantiating the LeNet architecture

The LeNet network will be then built with the architecture described in table using Keras Library. The build method requires following four parameters:

- The width of the input image: width = 28
- The height of the input image: height = 28
- The number of channels (depth) of the image: depth = 1
- The number class labels in the classification task: classes = 10

Table 1: Lenet Network Parameters and Implementation

Layer Type	Output Size	Filter Size / Stride	Python Code
INPUT IMAGE	28×28×1		model = Sequential() inputShape = (height, width, depth)
CONV	28×28×20	5×5;*K* = 20	model.add(Conv2D(20, (5, 5), padding="same", input_shape=inputShape))
ACT (ReLU)	28×28×20		model.add(Activation("relu"))
POOL	14×14×20	2×2	model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
CONV	14×14×50	5×5;*K* = 50	model.add(Conv2D(50, (5, 5), padding="same"))
ACT (ReLU)	14×14×50		model.add(Activation("relu"))
POOL	7×7×50	2×2	model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
FC	500		model.add(Flatten()) model.add(Dense(500))
ACT (ReLU)	500		model.add(Activation("relu"))
FC	10		model.add(Dense(classes))
SOFTMAX	10		model.add(Activation("softmax"))

Python Code:

```
model = LeNet.build(width=28, height=28, depth=1, classes=10)
```

2.3 Training and Fitting LeNet

Model is trained using the training batch from MNIST dataset.

Python Code:

```
opt = SGD(lr=0.01)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
# train the network
print("[INFO] training network...")
H = model.fit(trainData, trainLabels,
validation_data=(testData, testLabels), batch_size=128, epochs=20, verbose=1)
```

2.4 Evaluating network performance

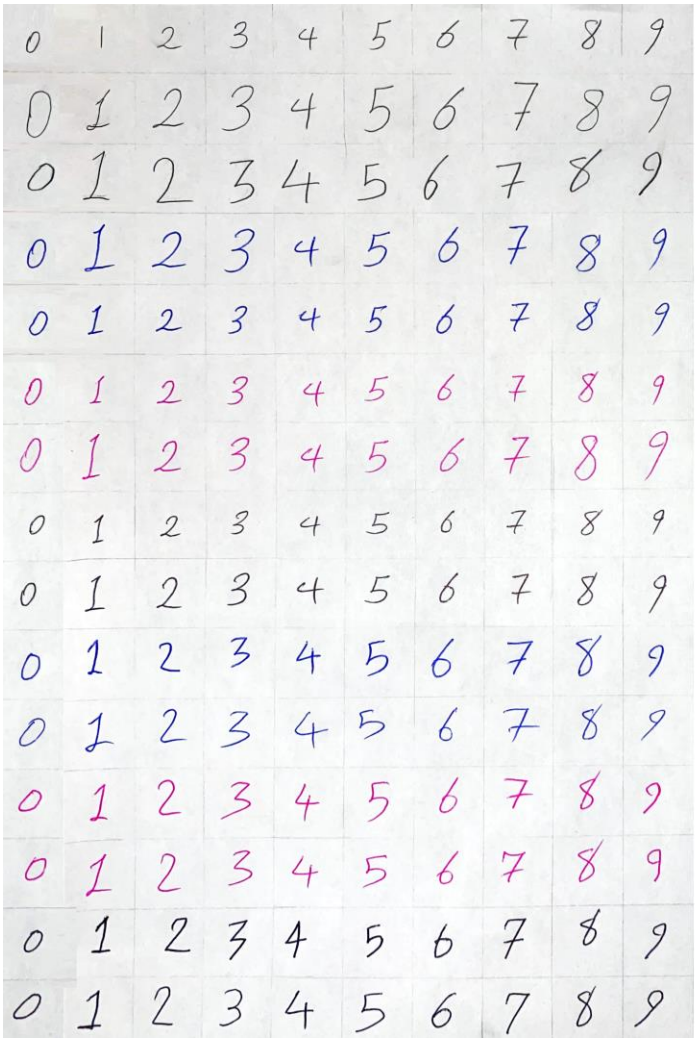
Then, the trained model will be evaluated using the testing batch from MNIST dataset.

Python Code:

```
print("[INFO] evaluating network...")
predictions = model.predict(testData, batch_size=128)
print(classification_report(testLabels.argmax(axis=1), predictions.argmax(axis=1),
target_names=[str(x) for x in le.classes_]))
```

2.5 Handwritten digits classification

Next, we need to test the performance of LeNet model on data outside the MNIST dataset. To this end, we prepared a small dataset of handwritten digits. It includes 150 images of digits (15 images per class) written by different individuals and with different pens and pencil (figure 8).



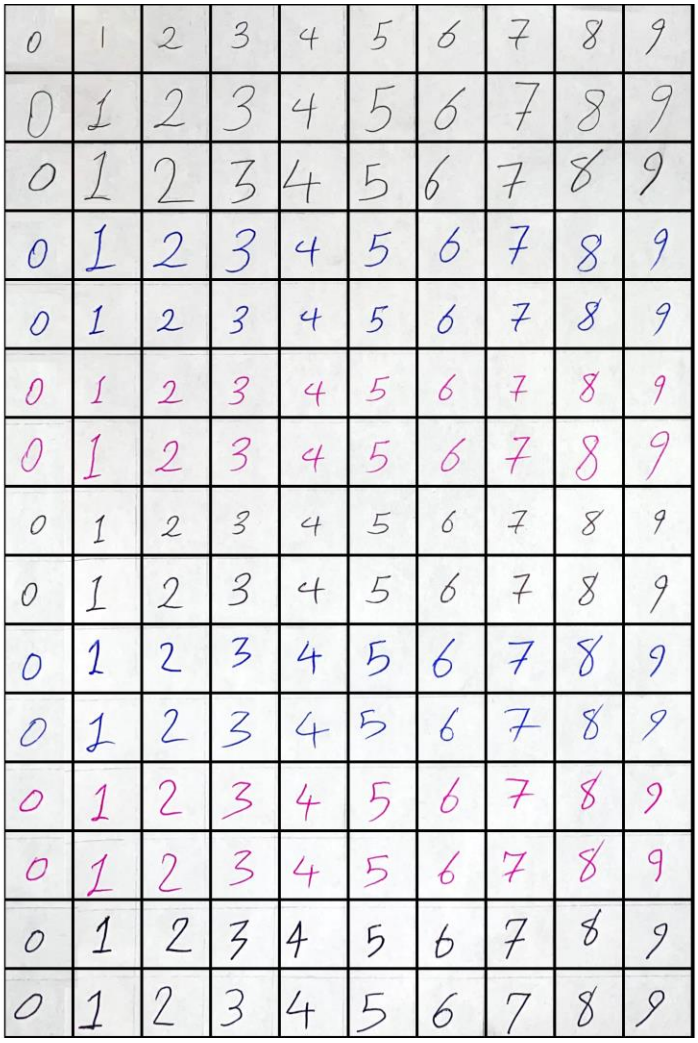


Figure 8: Hand written digits- Left: orginal sacnned image. Right: partioned image

The digits are spited using python code and then pre-processed in different ways resulting in 15 different batches of data (figure 9):

- **'none' batch** => Plain scanned images
- **'thresh' batch** => Smoothing and thresholding scanned images
- **'crop_tight' batch** => Cropping empty background around digits
- **'crop_sq' batch** => Padding cropped digits to make them squared
- **'pad_[scale]' batch** => Padding squared images with scales = {1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0}

All above data batches were resized to 28x28 and then classified by trained model.

Python codes are presented in load_digits.py and lenet_digits.py files.

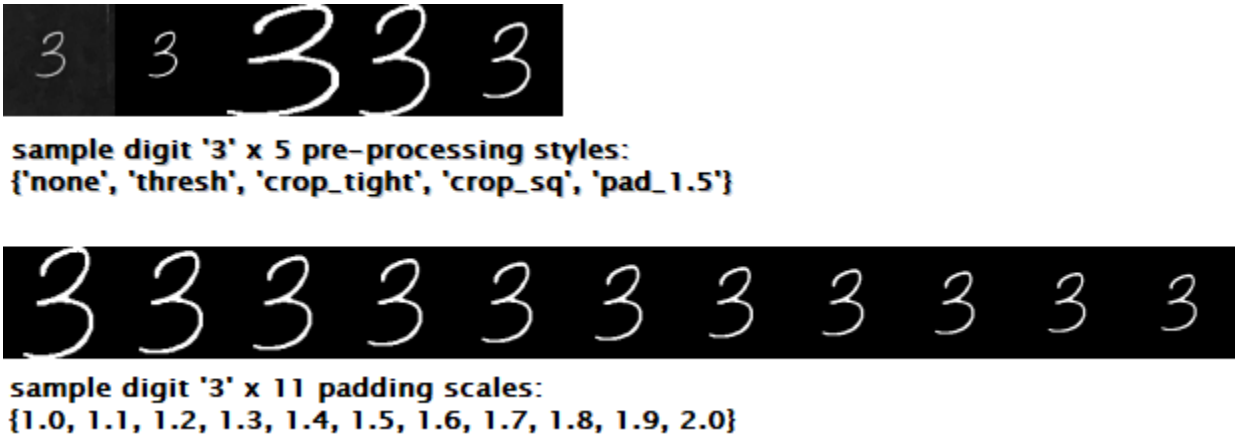


Figure 9: Sample digit '3' with different pre-processing styles

3. Results and Discussion

3.1 Training LeNet on MNIST

The LeNet network was trained and evaluated using MNIST dataset and the results are presented in below figure.

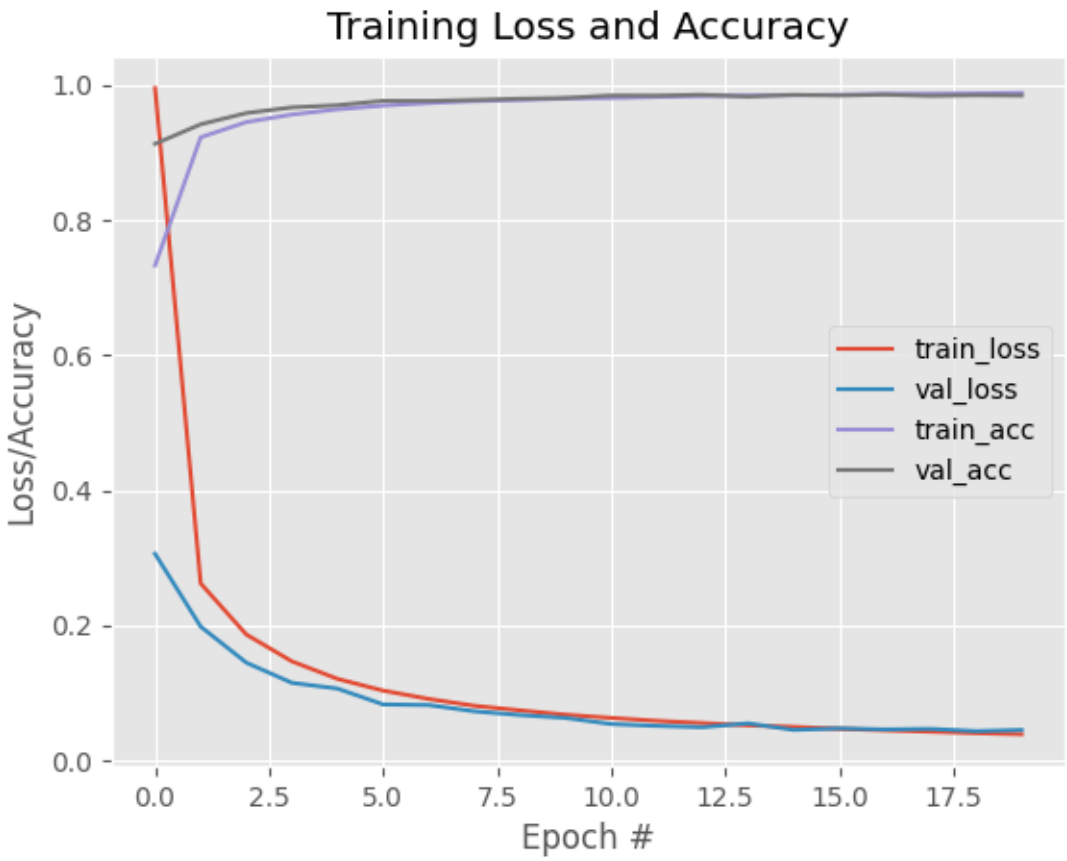


Figure 10: Training LeNet on MNIST

```
Epoch 20/20


60000/60000 - 282s - loss: 0.0396 - accuracy: 0.9879 - val_loss: 0.0457 - val_a
ccuracy: 0.9840
[INFO] evaluating network...
2020-06-08 23:51:36.767062: W tensorflow/core/framework/cpu_allocator_impl.cc:8
1] Allocation of 31360000 exceeds 10% of system memory.
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.99      0.98      0.98      1032
           3       0.98      0.99      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.99      0.98      0.99       892
           6       0.99      0.98      0.99       958
           7       0.99      0.97      0.98      1028
           8       0.95      0.99      0.97       974
           9       0.98      0.97      0.98      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

Figure 11: Evaluation LeNet on MNIST

As illustrated in above figure:

The overall performance of the network is very good.

- **Loss**
  Training and validation loss are have both decreased to 0.05. There are some fluctuations in validation loss until epoch 15 but the amplitude reduces to an acceptable amount. We can safely say that this network is not overfitted.

- **Accuracy**
  With a similar pattern, training and validation accuracy have increased to 98% which is impressive. Again, there are some rise and fall in validation accuracy until epoch 15 but the amplitude reduces to a negligible amount.

- **Number of epochs**
  It is worth mentioning that the required number of epochs is comparatively low. The network reaches 96% accuracy at epoch 5 and the final 98% validation accuracy is obtained with only 20 epochs.

3.2 Testing Trained LeNet Network on Handwritten Digits

The trained LeNet has an excellent accuracy for testing dataset from MNIST. However, the MNIST dataset is heavily pre-processed. We are interested in the performance of this network on data outside MNIST. As previously described, we prepared a small dataset of handwritten digits consisting of 150 images (10 images per class). The digits are written with different pens and hands. The dataset is lightly pre-processed contributing to 14 data batches. The results are demonstrated in below figures.

3.2.1 Trained LeNet on Handwritten Digits [150 handwritten digits x 5 pre-processing styles]

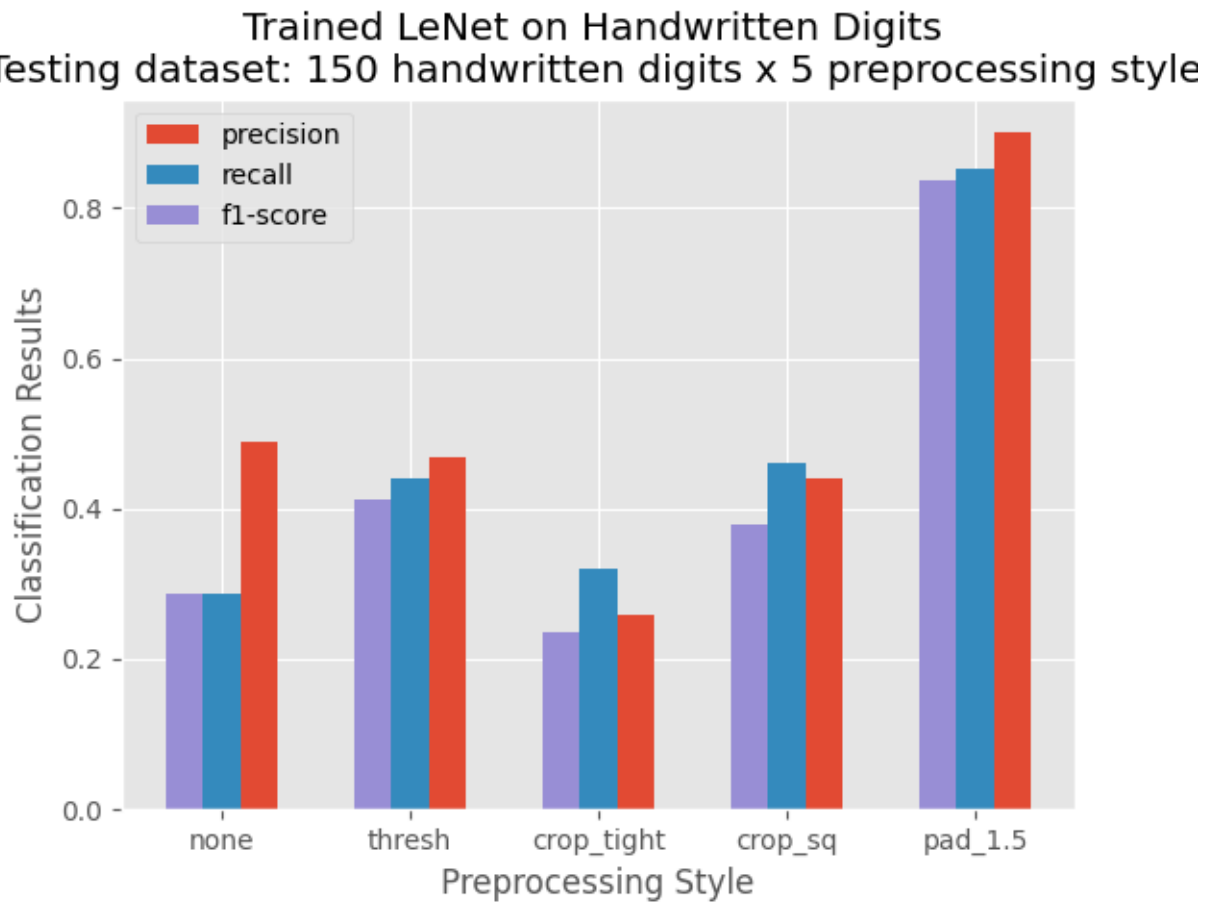


Figure 12: Trained LeNet on Handwritten Digits [150 handwritten digits x 5 pre-processing styles]

As illustrated in above figure:

- **'none' => 49%**
  Plain scanned images are classified with 49% precision, significantly lower than 98% precision for MNIST testing dataset.

- **'thresh' => 47%**
  Smoothing and thresholding alone do not have significant effect on classification precision.
- **'crop_tight' => 26%**
  Tightly cropped digits do not have equal width and height and therefore will lose their original ration when being resized to 28x28. This change of width/height ratio has significantly reduced the precision to 26%.
- **'crop_sq' => 44%**
  By padding cropped digits to squared dimension, their width/height ratio will remain constant during resizing. However, the classification accuracy is still lower than plain images.
- **'pad_1.5' => 90%**
  Padding squared images with scales higher than 1 has a significant positive effect on the accuracy. Scale 1.5 results in highest precision which is 90%. This effect is further studied in next section.

3.2.2 Trained LeNet on Handwritten Digits [150 handwritten digits x 11 padding scales]

Due to the significant effect of padding scale on classification precision, we tested 11 scales ranging from 1.0 to 2.0 ({1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0})

As illustrated in figure 13:

- As mentioned before, padding squared images with scales higher than 1 has a significant positive effect on the accuracy.
- The 44% precision for scale 1.0 has increased to 90% for scale 1.5 results. This is probably because this scale is closest to the scale of MNIST dataset.

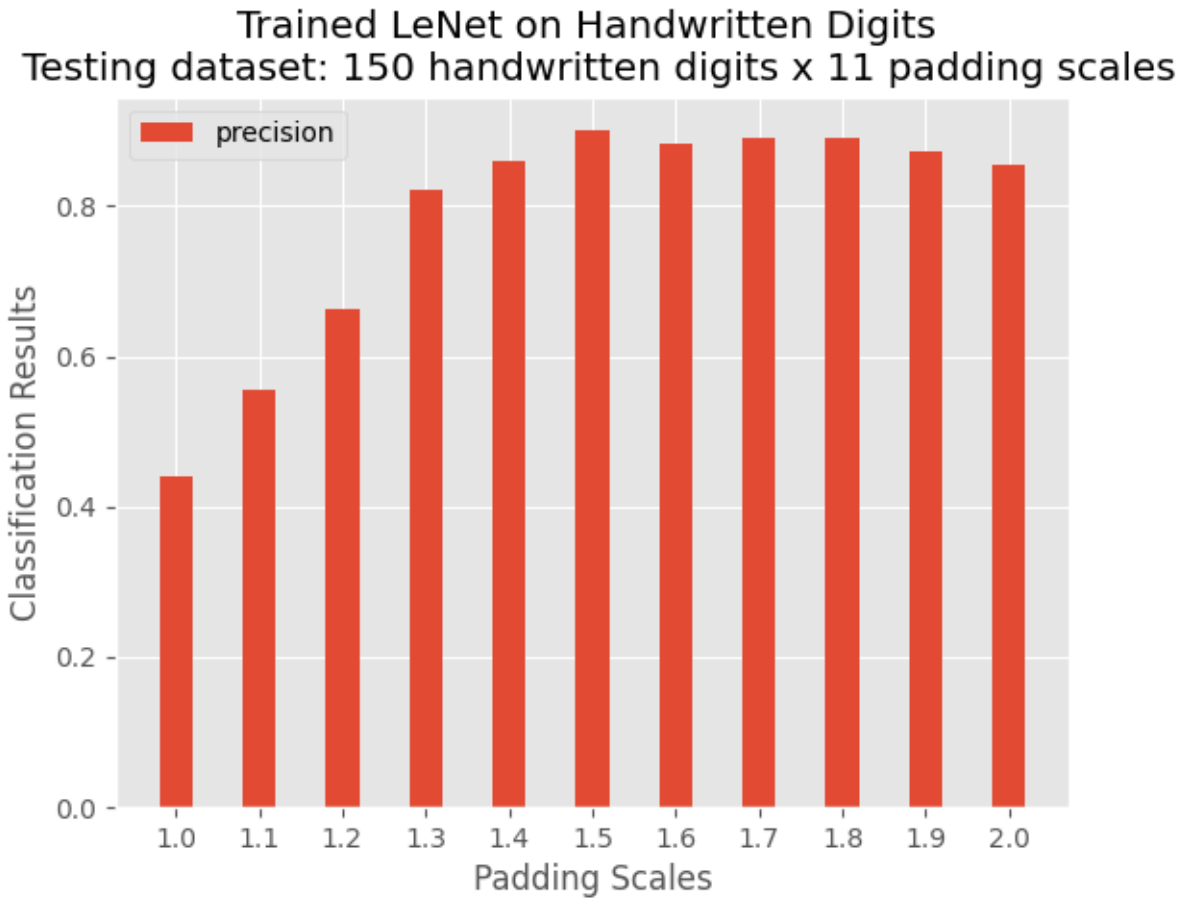


Figure 13: Trained LeNet on Handwritten Digits [150 handwritten digits x 11 padding scales]

3.3 Conclusion

- LeNet network trained and tested using MNIST dataset obtains an excellent accuracy of 98% with only 20 epochs.
- LeNet model on handwritten images:
  - Plain scanned images are classified with 49% precision, significantly lower than 98% precision for MNIST testing dataset;
  - Thresholding and smoothing scanned digits do not have any significant effect on classification precision
  - Change of width/height ratio decreases the precision to 26%
  - Padding squared images with scales higher than 1 has a significant positive effect on the accuracy. Scale 1.5 results in highest precision which is 90%.

References

[1] book 'Deep Learning for Computer Vision - Starter Bundle' by Adrian Rosebrock