Project Part 3: Classification Using Neural Networks and Deep Learning

CSE 575: Statistical Machine Learning

Arizona State University – Summer 2022

Course Instructor: Masudul Quraishi

Submitted By:

Aishwarya Baalaji Rao

ASU ID: 1222423228

Date of Submission: 3rd July 2022

ABSTRACT: This document serves as the report to the third part of the project. In this part, we implement classification using neural networks and deep learning on the SVHN dataset. The classification task is carried out using Google CoLab and the libraries used are: Keras, PyTorch and TensorFlow. The goal of the implementation is to build and train the CNN model on the given dataset, plot the training and testing curves as a function of epochs and to report the final test accuracy.

KEYWORDS: Classification, CNN, Deep Learning, SGD, ReLU, Kernels, Feature Maps

1. Introduction

Convolutional Neural Network (CNN) is an algorithm in the field of deep learning, which is a subset of statistical machine learning. It can take visuals or images as input, assign appropriate weights and biases to accurately classify the objects in their respective classes. This technique requires far less preprocessing than other classification algorithms. The architecture of CNN involves convolution layers, pooling layers, and fully connected layers to learn spatial features adaptively.

2. Problem Definition and Algorithm

2.1 Task Definition

In the third part of the project, we build a small CNN for a classification task. The dataset used is SVHN, which consists of ten classes of printed digits extracted from photographs of home plates. There are 73,257 training samples and 26,032 test samples in all. The input picture has a 32x32 resolution and three channels (RGB). Using the CNN architecture mentioned, we train the model using the Stochastic Gradient Descent optimizer and report the final test accuracy along with the plot for training error and the testing error as a function of epochs.

2.2 CNN Architecture

The function of the CNN is to transform the input pictures into a form that is simpler to process, without sacrificing elements that are essential for making accurate predictions. Input, hidden, and output layers make up a convolutional neural network. Any hidden layers in a feed-forward neural network are veiled by the activation function and final convolution. The hidden layer typically conducts a dot product of the convolution kernel with its input matrix with ReLU activation. As the convolution kernel slides along the layer's input matrix, it creates a feature map that feeds the next layer. This is followed by pooling, fully connected, and normalization layers.

2.2.1 **Convolutional Layers** – This is the first layer of the CNN and is used to extract the features of every input image fed to it. This layer performs the mathematical function of convolution between the input picture and a filter of a certain size (NxN). By dragging the filter over the input picture, the dot product is calculated between the filter and the portions of the input image proportional to the filter's size. The result is the Feature map, which provides

information about the picture, including its corners and edges. This is then fed to the next pooling layer.

2.2.2 **Pooling Layers** – This layer's major objective is to lower the size of the convolved feature map to reduce computational expenses. This is achieved by reducing the interlayer connections and individually operating on each feature map. In this project, we are using Max Pooling which takes the largest cell value from the feature map of a given size. This layer serves as a bridge between the convolutional layer and the next fully connected layer.

2.2.3 **Fully Connected Layers** – Fully Connected (FC) layer connects neurons between layers using weights and biases, this is where the classification task begins. These layers come before the output layer in a CNN Architecture. The FC layer receives the flattened picture from the preceding levels. The flattened vector then goes through many FC levels where mathematical functions are performed. Here, categorization starts.

2.2.4 **Activation Functions** – This is one of the significant parts of the CNN architecture. These functions are used to decide which information of the model goes through in the forward direction and which ones do not. It establishes a non-linearity attribute to the network. ReLU, Softmax, tanH, and Sigmoid are typical activation functions.

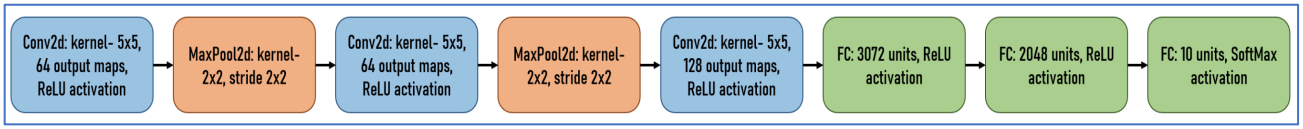


Fig 1. *The CNN architecture implemented in this experiment*

3. Experimental Evaluation

3.1 Methodology

Google CoLab was used to train and run the CNN model. After the initial setup and uploading our test and train dataset, following are the steps and variables used to build the architecture:

- The first convolutional layer is used as a hidden layer with 64 feature maps. The convolutional kernel size is 5x5, stride of 1, and the activation function is ReLU.
- This was followed by a max pooling layer with a pooling size of 2x2 and a stride of 2.
- The output of the previous pooling layer is fed to another convolutional layer with 64 feature maps, kernel size of 5x5, stride of 1 and activation function as ReLU.

- This second convolutional layer is followed by yet another max pooling layer with the same pooling size and strides.
- After this, the output is connected to a third convolutional layer with 128 feature maps, kernel size of 5x5, stride of 1 and ReLU as the activation function.
- This output is then fed to a FC layer with 3,072 nodes and ReLU as the activation function.
- It is followed by yet another FC layer with 2,048 nodes and ReLU activation.
- The last layer is another FC layer with 10 output nodes and SoftMax activation which classifies the result between 0-9 in the dataset.
- The entire network is trained with a Stochastic Gradient Descent optimizer with a **batch size** of **128**, **learning rate** = __ and **epochs** = __

3.1.1 Stochastic Gradient Descent Optimizer (SGD)

This optimizer works great on massive datasets. Stochastic can be translated to mean "randomness" in the algorithm. This means that instead of taking the entire dataset for every iteration, a batch of data is considered for processing according to the batch size mentioned.

The SGD algorithm first sets initial parameters w and learning rate. Then the data is randomly shuffled at every iteration to obtain a minimum. SGD iterates a lot more to find the local minima due to this randomness. But overall, the computation cost is still lower compared to gradient descent optimizer (for large data). A momentum value could be used to speed up SGD functions.

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Fig 2. *SGD Formula*

3.1.2 ReLU Activation

To train a deep neural network with SGD optimizer, we need an activation function that behaves like a linear function but is non-linear that allows for complicated data associations to be made. This is done using the Rectified Linear activation function, known as, ReLU. The function returns 0 for negative values and allows positive values to flow forward as is.

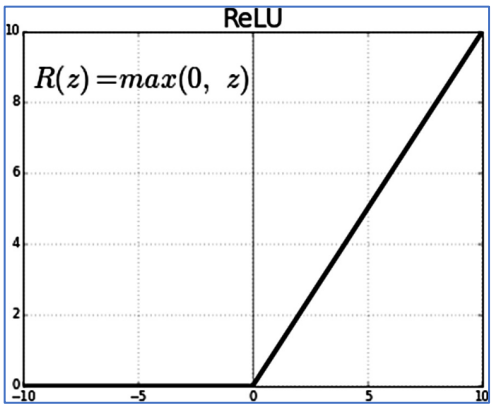


Fig 3. *ReLU Activation Function*

3.1.3 SoftMax Activation

Softmax turns a vector of integers into a vector of probabilities, with probabilities proportional to the scale of each value in the given vector. The softmax function is most often used as a neural network activation function. The network is designed to produce N values, one for each class in the classification job. The softmax function normalizes the outputs, transforming weighted sum values into probabilities that total to one. Each softmax output value represents the class membership probability.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Fig 4. *SoftMax Activation Formula*

4. Results and Conclusion

The outcomes of the trained CNN model according to the architecture mentioned, is discussed in this section. The final test loss, the accuracy on the test dataset and the plots for Loss vs Epochs (for test and train) and Accuracy vs Epochs (for test and train) are shown below along with their interpretations:

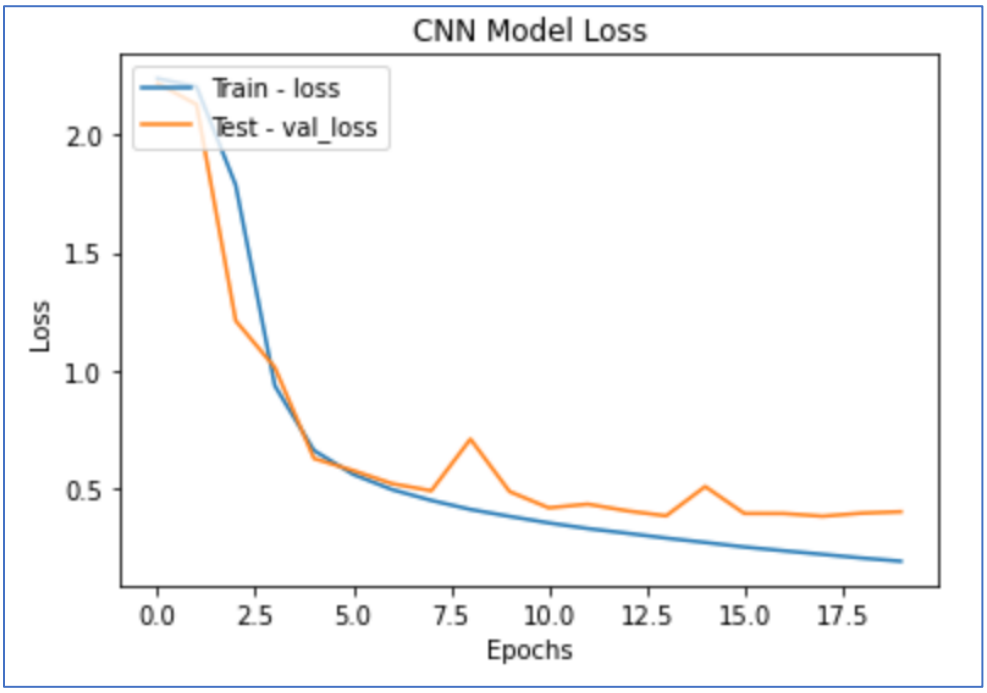
Parameters Chosen:

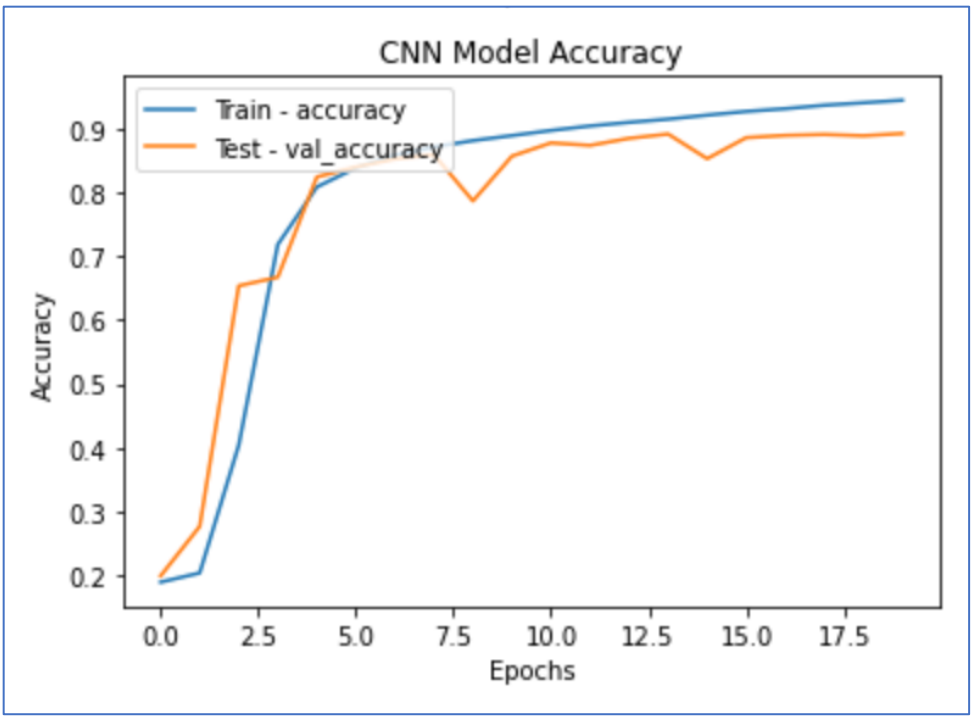
- Learning rate = 0.01
- Batch size = 128
- Epochs = 20

Using the above parameters to train the model, following was the results observed:

```
Epoch 1/20
573/573 [==============================] - 19s 16ms/step - loss: 2.2377 - accuracy: 0.1886 - val_loss: 2.2173 - val_accuracy: 0.1985
Epoch 2/20
573/573 [==============================] - 8s 14ms/step - loss: 2.2024 - accuracy: 0.2030 - val_loss: 2.1272 - val_accuracy: 0.2761
Epoch 3/20
573/573 [==============================] - 8s 14ms/step - loss: 1.7858 - accuracy: 0.4036 - val_loss: 1.2112 - val_accuracy: 0.6538
Epoch 4/20
573/573 [==============================] - 8s 14ms/step - loss: 0.9355 - accuracy: 0.7187 - val_loss: 1.0131 - val_accuracy: 0.6674
Epoch 5/20
573/573 [==============================] - 8s 14ms/step - loss: 0.6613 - accuracy: 0.8090 - val_loss: 0.6274 - val_accuracy: 0.8247
Epoch 6/20
573/573 [==============================] - 8s 14ms/step - loss: 0.5600 - accuracy: 0.8390 - val_loss: 0.5764 - val_accuracy: 0.8404
Epoch 7/20
573/573 [==============================] - 8s 14ms/step - loss: 0.4953 - accuracy: 0.8589 - val_loss: 0.5205 - val_accuracy: 0.8531
Epoch 8/20
573/573 [==============================] - 8s 14ms/step - loss: 0.4496 - accuracy: 0.8719 - val_loss: 0.4906 - val_accuracy: 0.8597
Epoch 9/20
573/573 [==============================] - 8s 14ms/step - loss: 0.4113 - accuracy: 0.8819 - val_loss: 0.7095 - val_accuracy: 0.7871
Epoch 10/20
573/573 [==============================] - 8s 14ms/step - loss: 0.3819 - accuracy: 0.8898 - val_loss: 0.4870 - val_accuracy: 0.8575
Epoch 11/20
573/573 [==============================] - 8s 14ms/step - loss: 0.3543 - accuracy: 0.8978 - val_loss: 0.4182 - val_accuracy: 0.8783
Epoch 12/20
573/573 [==============================] - 8s 14ms/step - loss: 0.3303 - accuracy: 0.9049 - val_loss: 0.4339 - val_accuracy: 0.8743
Epoch 13/20
573/573 [==============================] - 8s 14ms/step - loss: 0.3105 - accuracy: 0.9105 - val_loss: 0.4055 - val_accuracy: 0.8854
Epoch 14/20
573/573 [==============================] - 8s 14ms/step - loss: 0.2899 - accuracy: 0.9156 - val_loss: 0.3842 - val_accuracy: 0.8922

Epoch 15/20
573/573 [==============================] - 8s 14ms/step - loss: 0.2714 - accuracy: 0.9219 - val_loss: 0.5086 - val_accuracy: 0.8535
Epoch 16/20
573/573 [==============================] - 8s 14ms/step - loss: 0.2522 - accuracy: 0.9277 - val_loss: 0.3943 - val_accuracy: 0.8865
Epoch 17/20
573/573 [==============================] - 8s 15ms/step - loss: 0.2363 - accuracy: 0.9320 - val_loss: 0.3941 - val_accuracy: 0.8901
Epoch 18/20
573/573 [==============================] - 8s 15ms/step - loss: 0.2210 - accuracy: 0.9374 - val_loss: 0.3826 - val_accuracy: 0.8915
Epoch 19/20
573/573 [==============================] - 8s 14ms/step - loss: 0.2055 - accuracy: 0.9412 - val_loss: 0.3956 - val_accuracy: 0.8895
Epoch 20/20
573/573 [==============================] - 8s 14ms/step - loss: 0.1919 - accuracy: 0.9453 - val_loss: 0.4017 - val_accuracy: 0.8930
Test loss: 0.401680052280426
Test accuracy: 0.8929778933525085
```





The results above show that there is a steep rise in accuracy and a steep low in error from 1st to 3rd epoch. The test loss for the model was **40.16**% and the accuracy is at **89.3**% which shows a healthy classification neural network. As observed in the accuracy graph, the test accuracy is close to the training curve with insignificant or minimal overfitting of the curve. The goal of training the model is to minimize the loss after every epoch run which is acceptably accomplished in this experiment.

5. Software Specifications

Google CoLab, Keras, PyTorch and TensorFlow

References

1. https://en.wikipedia.org/wiki/Convolutional_neural_network
2. https://keras.io/getting_started/
3. http://ufldl.stanford.edu/housenumbers/