CSE 472
Machine Learning Sessional

REPORT ON
Assignment 4: Convolutional Neural Network
Bengali Character Recognition Challenge

Saif Ahmed Khan
Student ID: 1705110
Section: B
Department of CSE, BUET

Dataset

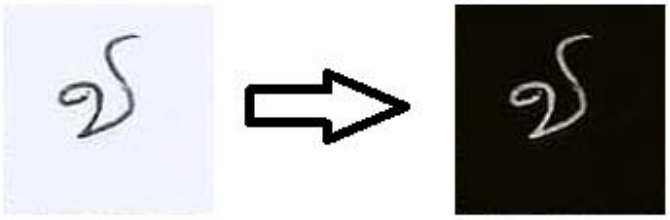
Numta Handwritten Bengali Digits
Find the data description here:
https://www.kaggle.com/competitions/numta/data

Task

Implement a CNN (Convolutional Neural Network) from scratch and use it to build the best model to train it for the given dataset on Kaggle.

Preprocessing the Data

The images read were converted to square dimensions and resize to 32 by 32 for faster training. Another improvement was to invert the image pixels (255 – pixel value). This made the images clearer and better for classification:



```
# resize all images to c_dim x c_dim
mode = cv2.IMREAD_GRAYSCALE if gray_sc else cv2.IMREAD_COLOR
X = [
    cv2.resize(
        255 - cv2.imread(path, mode), (c_dim, c_dim)
    ) for path in tqdm(image_paths, desc='Loading images:')
]
X = np.array(X)

# add channel dimension if grayscale
if gray_sc:
    X = np.expand_dims(X, axis=-1) # add channel dimension

# X.shape is (batch, c_dim, c_dim, channels)
# convert to (batch, channels, c_dim, c_dim)
X = np.transpose(X, (0, 3, 1, 2))
X = X / 255.0
```

Finally, the pixel values were normalized and transposed to the following shape: (number of batches, number of channels, image dimension, image dimension)

Taking Baby Steps: Vanilla CNN

After the CNN building blocks (Layers: **Convolution, MaxPooling, Flattening, FullyConnected, ReLU** and **SoftMax**) were built, I used a simple model that included all of the Layers each once just to test if the model was working like it should be.

A useful observation was to see if accuracy and macro f1 scores increased if a larger dataset was used (just 'training-c' instead of just 'training-a')

```
convolutional_network.build_netword([
    Conv2D(no_of_output_channels=3, filter_dim=5, strides=1, padding=1),
    ReLU(),
    MaxPooling(stride=2, filter_dim=2),
    Flatten(),
    Dense(no_of_outputs=10),
    Softmax()
])
# train network and save the model
convolutional_network.train(X_test, y_train, X_test, y_test, lr=0.01, epoch=10, batch_size=32)
```

The vanilla CNN model above gave **23.3% accuracy** and **17.6% macro f1** scores when trained over the specified data and validation set (a test-train split of training-a, training-b, training-c combined).

Vanilla CNN: Increasing Filters and Dense Layers

Increasing the number of filters increases the number of filters, increases the features extracted from the images. Increasing the Dense or FullyConnected Layers would also help classify these increased number of features:

```
convolutional_network.build_network([
    Conv2D(no_of_output_channels=32, filter_dim=5, strides=1, padding=1),
    ReLU(),
    MaxPooling(stride=2, filter_dim=2),
    Flatten(),
    Dense(no_of_outputs=128),
    Dense(no_of_outputs=10),
    Softmax()
])
# train network and save the model
convolutional_network.train(X_test, y_train, X_test, y_test, lr=0.01, epoch=10, batch_size=32)
```

Hence, the enhanced vanilla CNN now performed a lot better with an **accuracy score of 55.5%** and a **macro f1 of 48.7%.** However, it was still poor to be declared as a useful model.

Please note that I used these vanilla architectures only for testing the implementation.

Old is Gold: The Power of LeNet-5

Now that the implementation was all good, I built the original LeNet-5 model as shown below. Note that the learning rate 'lr' has been set to 0.001 initially and epochs to 50 (such that the runtime is optimum enough to finish training within Kaggle's limit's)

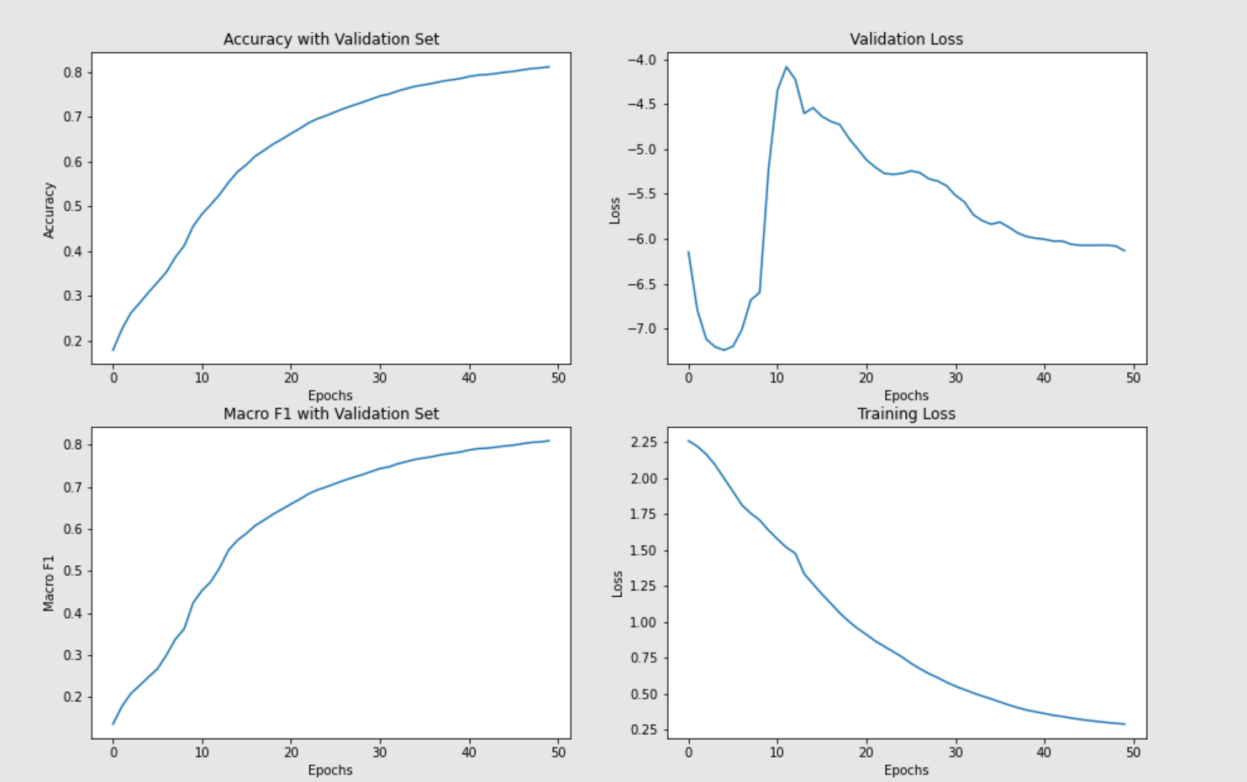
```
convolutional_network = ConvolutionalNetwork()
convolutional_network.build_netword([
    Conv2D(6, 5, 1, 0),
    ReLU(),
    MaxPooling(2, 2),
    Conv2D(16, 5, 1, 0),
    ReLU(),
    MaxPooling(2, 2),
    Flatten(),
    Dense(120),
    ReLU(),
    Dense(84),
    ReLU(),
    Dense(10),
    ReLU(),
    Softmax()
])

# train the network
convolutional_network.train(X_train, y_train, X_test, y_test, 0.001, 50, 32)
```

The batch size was set to 32. This architecture was evidently the best one could get due to the time limit of training such an implementation (using only the specified numpy library) that did not support any parallelism.

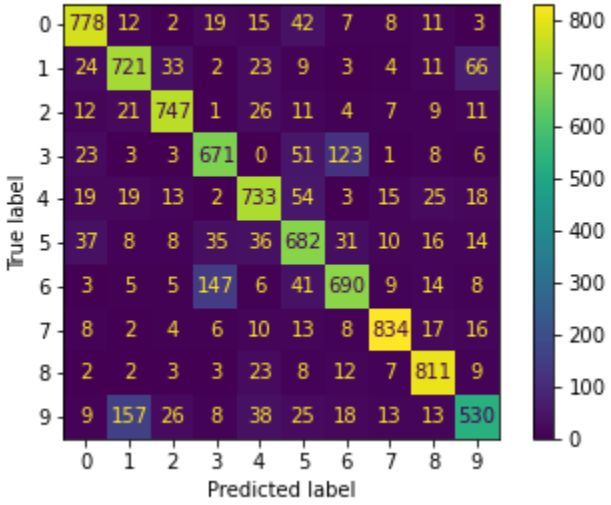
Training Results:

Runtime: 23046.0s
Accuracy: 0.8112037871956718,
Validation Loss: -6.131650824414729,
Macro_f1: 0.8097590817276288
Training Loss: 0.29020864589119066

Metric Graphs:



Confusion Matrix:



Tuning LeNet-5: *lr -> 0.01*

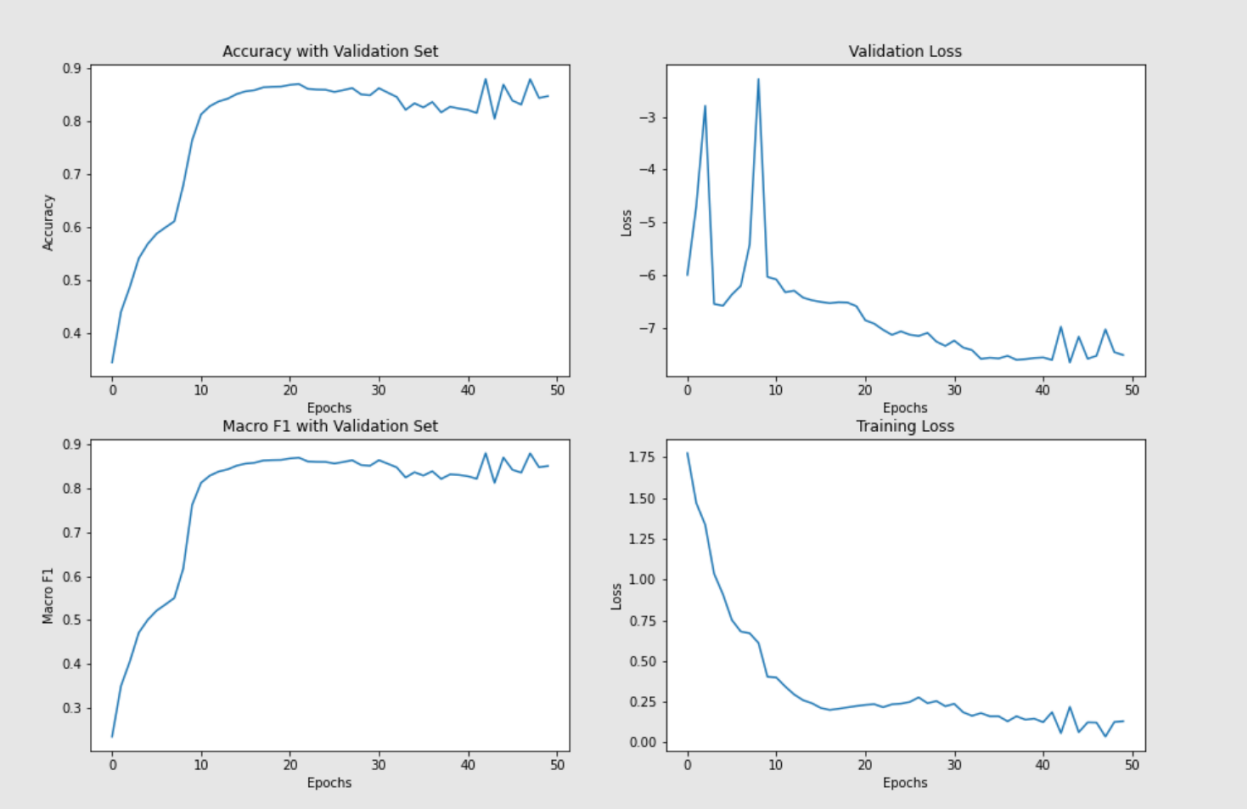
Now that a model with very decent metrics was established, the learning rate was tuned to 0.01 to see if that achieved better results and as expected it produced a macro f1 score of 85.0%.

```
convolutional_network = ConvolutionalNetwork()
convolutional_network.build_network([
    Conv2D(6, 5, 1, 0),
    ReLU(),
    MaxPooling(2, 2),
    Conv2D(16, 5, 1, 0),
    ReLU(),
    MaxPooling(2, 2),
    Flatten(),
    Dense(120),
    ReLU(),
    Dense(84),
    ReLU(),
    Dense(10),
    ReLU(),
    Softmax()
])

# train the network
convolutional_network.train(X_train, y_train, X_test, y_test, 0.01, 50, 32)
```

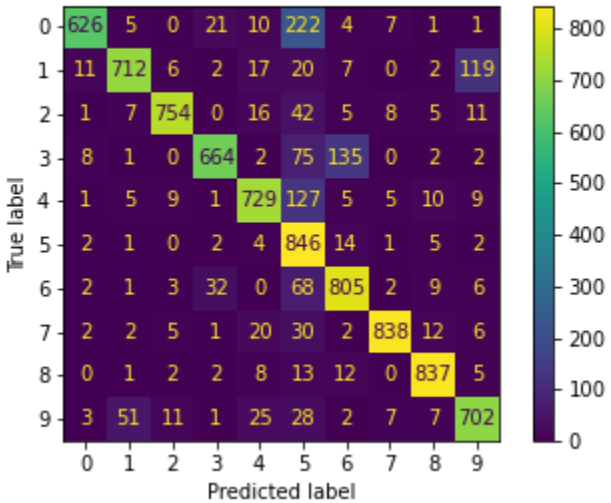
Training Results:

Runtime: 21762.9s
Accuracy: 0.8468214607754734,
Validation Loss: -7.519407394772421,
Macro_f1: 0.8506605225768343,
Training Loss: 0.130923227441977

Metric Graphs:



Confusion Matrix:



LeNet-5: *Clearer Images*

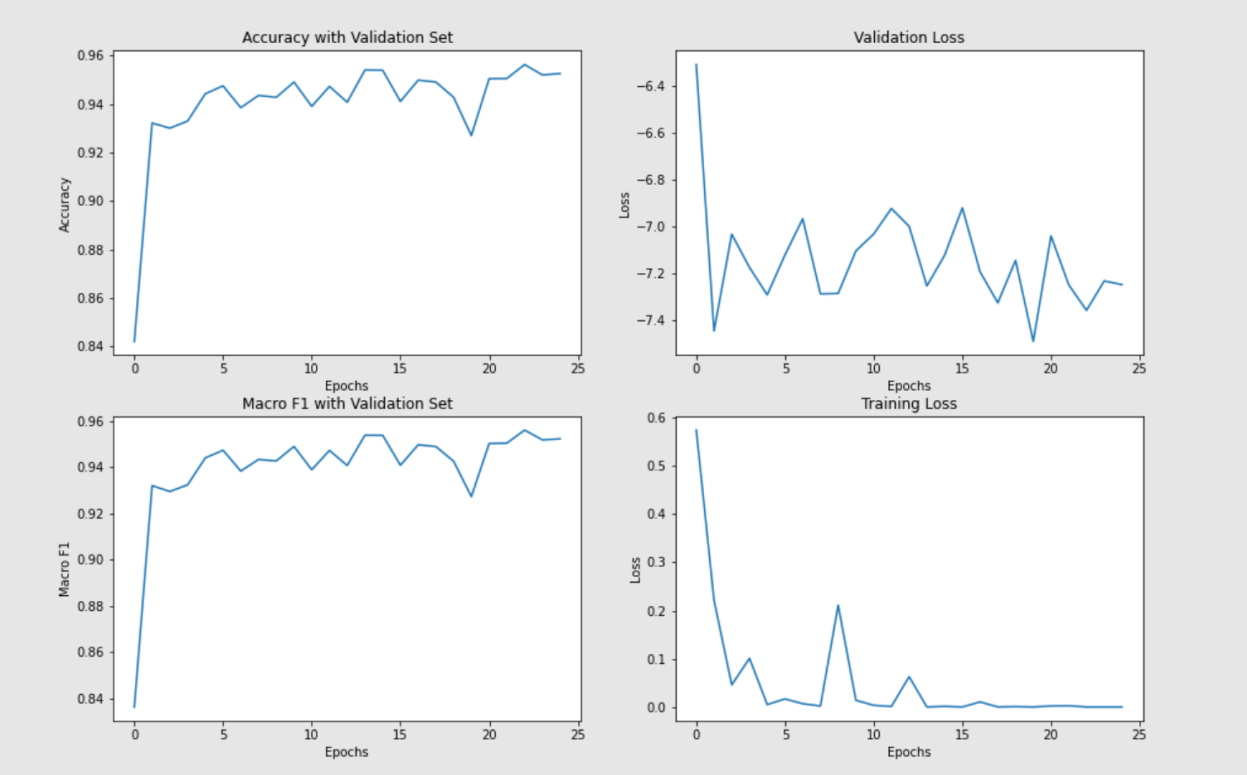
One could not go beyond to use more complex CNN architectures like **AlexNet** because it would take a lot of time to train using this implementation and lack of support for a GPU. So, the last improvement, I did was to use scale down the input images to 64 by 64 instead of 32 by 32.

This increased the training time per epoch by a large margin, so I had to reduce the epochs to 25 and increase learning rate to 0.1. This produced the best results I have recorded, a **macro f1 of 95.2%**.

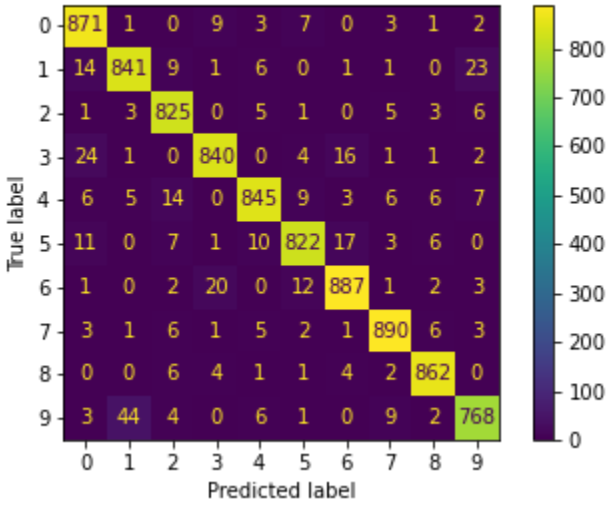
Training Results:

Runtime: 32487.7s
Accuracy: 0.9525473399458972,
Validation Loss: -7.247408912811393,
Macro_f1: 0.9523544234682374
Training Loss: 2.0793054652897394e-05

Metric Graphs:



Confusion Matrix:



LeNet-5: *Testing the best model*

The last model had the best macro f1 scores, so the dataset's 'training-d' folder was tested using the same params. The weights and biases were loaded from a saved pickle file to avoid retraining.

Independent Test Results:
Accuracy: 0.9540704070407041,
Macro_f1: 0.9541251631006855

Some Challenges Faced:

A few difficulties and challenges I faced were the following:

1. When implementing the Layers, I would often run into dimensional mismatch errors, primarily because I did not hot-encode the y-values at first.
2. The training took a lot of time, so an error after the training function call would just ruin the results. So, a useful advice would be to set the epoch to 1 to see if the notebook is successful in completion or just test it locally.
3. Numpy does not support or provide any scope for parallelism and hence training was done on CPUs. The only convenience Kaggle provided was that nothing was running on my local machine.