**End to End flow of equation solver.**

1. Read an image of a linear equation.
2. Get the bounding box for each individual characters.
3. Extract the bounding box from the image.
4. Preprocess every extracted sub-image – S’
5. Load the saved model which is stored in pickle format.
6. Run every S’ through model.
7. Get the predictions for each image.
8. Use the sympy for solving the equation.

**Algorithms implemented.**

1. Fully connect Deep neural network

RELU activation function

Softmax activation function

Cross entropy loss function

Convolutional neural network

Flattening layer

MaxPool layer.

Sequential Model

1. Challenges faced during the implementation.
   1. The biggest hurdle was implementing the convolution neural network especially the backpropagation algorithm for convolution algorithm.
   2. Even though the algorithm is straightforward it was tough to implement as we were frequently encountering issue in managing the gradients for each activation layer.
   3. Vectorizing the code was another challenge to solve, NumPy really helped in performing the vectorization and speeding up the code.
   4. In order to validate the correct working of each layer, we tried training each layer individually on the known dataset such as MNIST, after we validated that the layer is able to learn features by looking into it’s validation accuracy during training we integrated it our sequential model. We frequently faced issues while integrating the layers together too.

**Dataset**

Original dataset - <https://www.kaggle.com/datasets/xainano/handwrittenmathsymbols/code>

Dataset distribution –

A graph of a graph

Description automatically generated with medium confidence

We are using a dataset named [“Handwritten math symbol dataset”](https://www.kaggle.com/datasets/xainano/handwrittenmathsymbols). The class distribution is not balanced, so we may need to use data augmentation to balance the dataset.

Number of classes - 82

Features - As it is a gray scale image dataset the features are just the number of pixels present in an image, so it will be 2025 pixels / features.

**Main classes include**:

Digits: 0 to 9

Characters: - a - z (some include upper case characters)

Math operators: - +, -, /, \*, =

It also includes basic Greek alphabet symbols like: alpha, beta, gamma, mu, sigma, phi and theta.

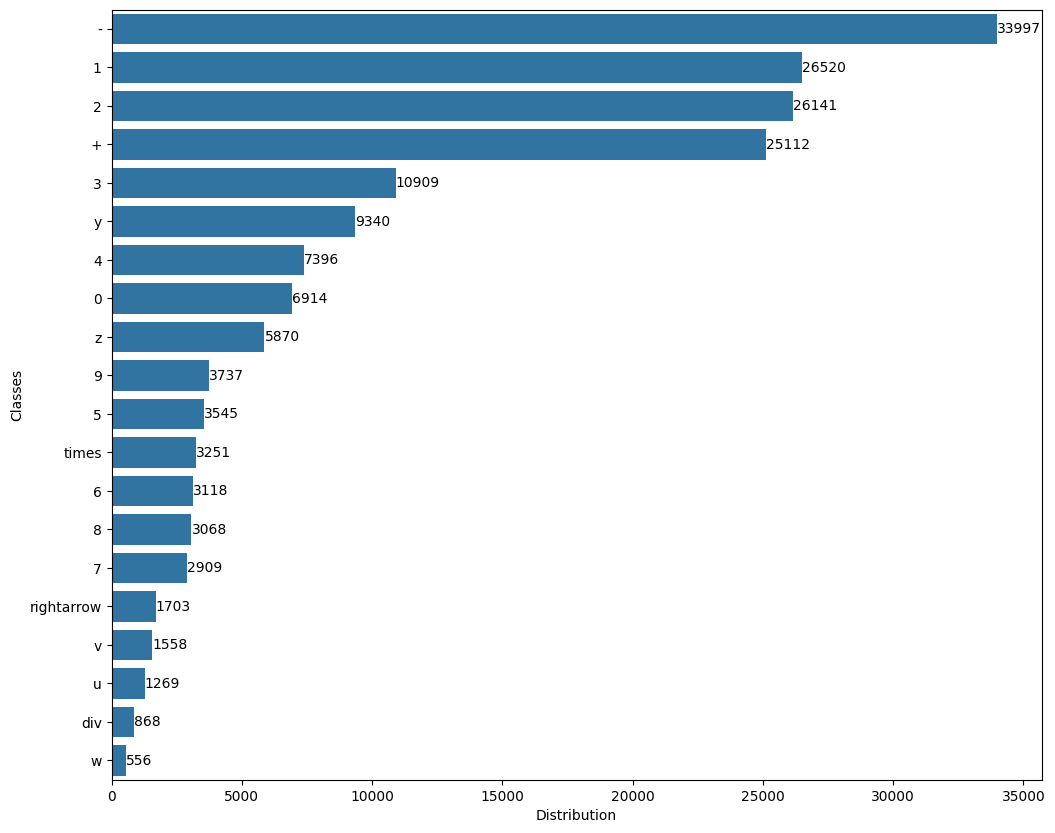
English alphanumeric symbols are included.

All math operators, set operators.

Basic pre-defined math functions like: log, lim, cos, sin, tan.

**Dataset preprocessing:**

We didn’t choose all the 82 classes from the original dataset; we chose the below 20 classes for our use case.



<Preprocessing stuff from Sreeman/bannman>

As the focus of this project was implementing algorithms, we didn’t spend much time in augmenting our dataset.

    img = cv2.GaussianBlur(img, (5, 5), 0)

    img = cv2.threshold(img, 0, 255, cv2.THRESH\_BINARY\_INV | cv2.THRESH\_OTSU)[1]

    img = cv2.copyMakeBorder(img, 5, 5, 5, 5, cv2.BORDER\_CONSTANT, *value*=0)

    img = cv2.resize(img, (55, 55), cv2.INTER\_CUBIC, *fx*=0.1, *fy*=0.1)

**Experiments:**

We implemented 4 models with different architecture and parameters:

#### Model 1: Flatten1, Dense1, softmax, cross entropy

#### Model 2: Dense1, relu, Dense2, softmax, cross entropy

#### Model 3: Conv1, relu, maxpool, flatten1,  Dense1, softmax, cross entropy

#### Model 4: Conv2, relu1, conv1, relu2, flatten1, dense1, softmax, cross entropy

Training metrics for each model

Note:

Relu is used in every hidden layer

Softmax as the last activation layer

**Model 1: Using only a single dense layer**

#### Model 1: Flatten1, Dense1, softmax, cross entropy

**Model information**

Layer Name -> Dense\_1

Weights shape -> (2025, 20)

\*\*\*\*\*\*\*\*\*\*

Epochs -> 5

Batch Size -> 16

Learning rate -> 0.0001

Accuracy and Loss at the end of 5 epochs

Train Accuracy -> 4.86424620080117, Validation Accuracy -> 5.035765379113019

Train Loss -> 28.74900236228159, Validation Loss -> 28.434207840170465

\*\*\*\*\*\*\*\*\*\*

Epochs -> 5

Batch Size -> 16

Learning rate -> 0.001

Accuracy and Loss at the end of 5 epochs

Train Accuracy -> 7.0674635976346405, Validation Accuracy -> 7.997138769670959

Train Loss -> 24.58266184480103, Validation Loss -> 24.015489364627967

\*\*\*\*\*\*\*\*\*\*

Epochs -> 20

Batch Size -> 16

Learning rate -> 0.001

Accuracy and Loss at the end of 20 epochs

Train Accuracy -> 15.695936923761684, Validation Accuracy -> 15.994277539341917

Train Loss -> 18.18884472589068, Validation Loss -> 18.38359080366181

\*\*\*\*\*\*\*\*\*\*

Epochs -> 30

Batch Size -> 16

Learning rate -> 0.001

Accuracy and Loss at the end of 30 epochs

Train Accuracy -> 18.78616392191772, Validation Accuracy -> 18.78397711015737

Train Loss -> 16.232349698472348, Validation Loss -> 16.212702294749086

\*\*\*\*\*\*\*\*\*\*

Epochs -> 30

Batch Size -> 16

Learning rate -> 0.01

Accuracy and Loss at the end of 30 epochs

Train Accuracy -> 55.83232657213709, Validation Accuracy -> 56.5379113018598

Train Loss -> 5.510358492691009, Validation Loss -> 5.375296873785613

\*\*\*\*\*\*\*\*\*\*

Epochs -> 60

Batch Size -> 16

Learning rate -> 0.01

Accuracy and Loss at the end of 60 epochs

Train Accuracy -> 66.40331913270172, Validation Accuracy -> 65.92274678111588

Train Loss -> 3.6693775343654065, Validation Loss -> 3.791698985400819

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

Train and Validation metrics

Epochs -> 100

Batch Size -> 16

Learning rate -> 0.01

Accuracy and Loss at the end of 100 epochs

Train Accuracy -> 72.4470655560501, Validation Accuracy -> 71.5450643776824

Train Loss -> 2.7025121853016123, Validation Loss -> 2.88509358657512

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

Train and Validation metrics – **Model 1.1**

Epochs -> 150

Batch Size -> 16

Learning rate -> 0.01

Accuracy and Loss at the end of 151 epochs

Train Accuracy -> 76.34164176257391, Validation Accuracy -> 74.89270386266095

Train Loss -> 2.138207131440353, Validation Loss -> 2.3585958681652954

Test accuracy, test loss -> (75.07296137339056, 2.344355587874297)

Model 1 with glorot activation – **Model 1.2**

Faster convergence than the best model, for 1 layer deep nn

\*\*\*\*\*\*\*\*\*\*

Train and Validation metrics

Epochs -> 20

Batch Size -> 16

Learning rate -> 0.01

Accuracy and Loss at the end of 20 epochs

Train Accuracy -> 84.8763273351561, Validation Accuracy -> 85.0071530758226

Train Loss -> 0.6012689837833791, Validation Loss -> 0.6103554523528566

Test accuracy, test loss (84.01716738197425, 0.6172994110007722)

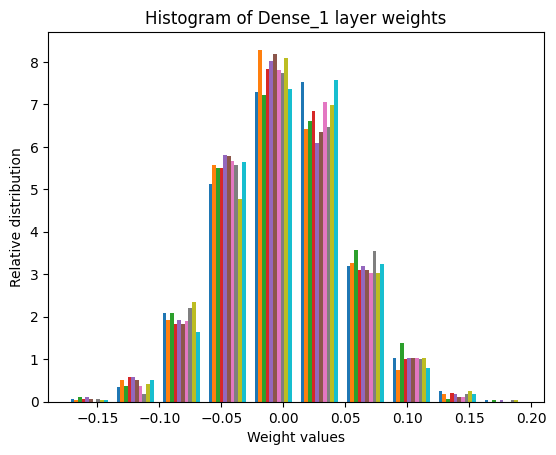
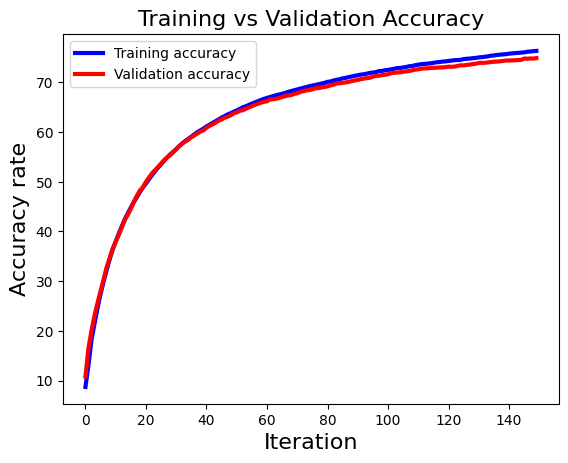
\*\*\*\*\*\*\*\*\*\*

**Observations for model:**

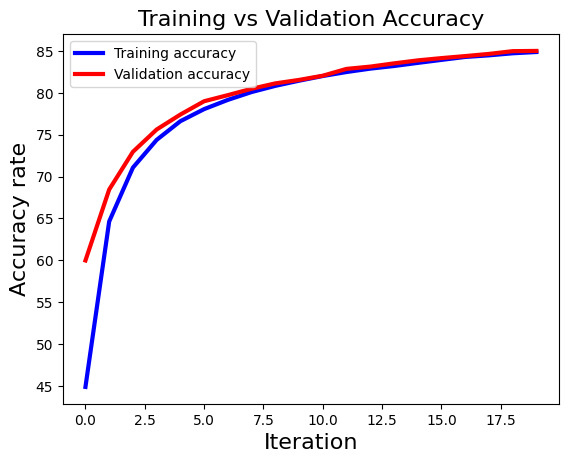
**1. Weight initialization: Without glorot/xavier weight initialization the model took more number of epochs for convergence compared to glorot/Xavier initialization.**

**2. As it is a single layer NN, it is the fastest compared to all models in training time.**

**3. The Model 1.2 was stuck in local minima i.e it’s accuracy didn’t improve above 74% after 100 epochs.**

** 4. But the same model (Model 1.2) with glorot init converged faster at 20 epochs, with validation and testing accuracy of 84% at the end of 20 epoch.**

**Graphs for the Model 1.2**

**A graph of training and validation

Description automatically generated**

**A graph with a blue line

Description automatically generated**

**A graph of numbers and a graph

Description automatically generated with medium confidence**

**Accuracy: 0.84**

**Classification Report:**

**precision recall f1-score support**

**0 0.92 0.95 0.93 1368**

**1 0.83 0.91 0.87 1385**

**2 0.80 0.80 0.80 1452**

**3 0.90 0.90 0.90 1439**

**4 0.76 0.83 0.79 1391**

**5 0.89 0.80 0.84 702**

**6 0.92 0.88 0.90 646**

**7 0.89 0.84 0.87 576**

**8 0.84 0.79 0.82 637**

**9 0.83 0.85 0.84 764**

**div 0.99 0.44 0.60 179**

**rightarrow 0.91 0.80 0.85 352**

**times 0.87 0.81 0.84 702**

**u 0.62 0.42 0.50 256**

**v 0.79 0.69 0.74 295**

**w 0.72 0.64 0.68 104**

**y 0.78 0.84 0.81 1329**

**z 0.76 0.72 0.74 1135**

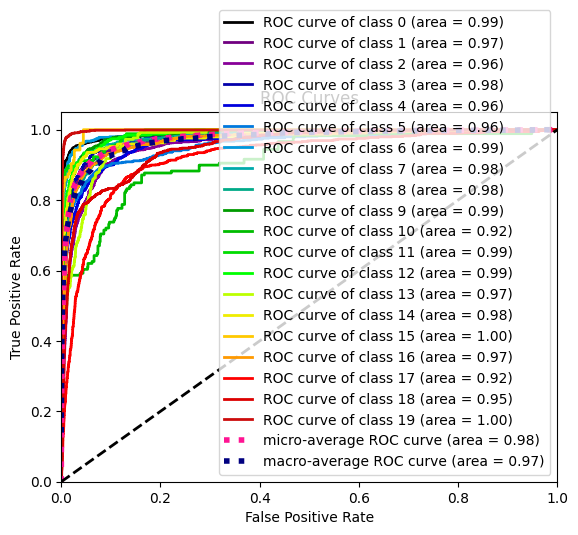
**+ 0.80 0.83 0.82 1378**

**- 0.94 0.99 0.96 1385**

**accuracy 0.84 17475**

**macro avg 0.84 0.79 0.80 17475**

**weighted avg 0.84 0.84 0.84 17475**

****

**Model 2:**

**Dense1, relu, Dense2, softmax, cross entropy**

Number of Layers 4

----------

Layer 1

Layer Name -> flatten\_1

----------

----------

Layer 2

Layer Name -> Dense\_1

Weights shape -> (2025, 50)

----------

----------

Layer 3

Layer Name -> relu\_1

----------

----------

Layer 4

Layer Name -> Dense\_2

Weights shape -> (50, 20)

Without glorot initialization

\*\*\*\*\*\*\*\*\*\*

Train and Validation metrics

Epochs -> 5

Batch Size -> 8

Learning rate -> 0.01

Accuracy and Loss at the end of 6 epochs

Train Accuracy -> 43.96261206841737, Validation Accuracy -> 41.77396280400572

Train Loss -> 2.3183348149889746, Validation Loss -> 2.370206448735256

Model 2.1

Epochs -> 20

Batch Size -> 16

Learning rate -> 0.01

Accuracy and Loss at the end of 20 epochs

Train Accuracy -> 48.55662236917403, Validation Accuracy -> 48.769670958512165

Train Loss -> 1.9402072302483715, Validation Loss -> 1.9619108513641768

\*\*\*\*\*\*\*\*\*\*

Model 2.2

Train and Validation metrics

Epochs -> 20

Batch Size -> 16

Learning rate -> 0.01

Accuracy and Loss at the end of 20 epochs

Train Accuracy -> 73.22916004323774, Validation Accuracy -> 73.80543633762517

Train Loss -> 0.9714841499700786, Validation Loss -> 0.9560338631569683

Test acc, test acc = (73.45350500715307, 0.9648617300015497)

**Observations**

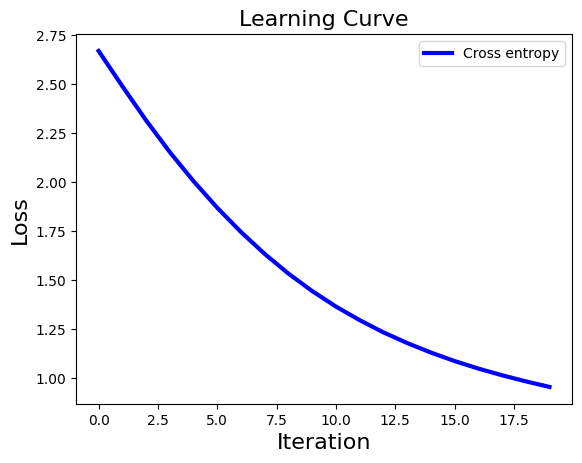
1. Model is stuck at local minima (Model 2.1) but with glorot init (Model 2.2) it improved.
2. Model 1.2 has better metrics than Model 2.2 which has the same hyperparameters but Model 2.2 has 2 Dense layers whereas Model 1.2 has only one dense layer

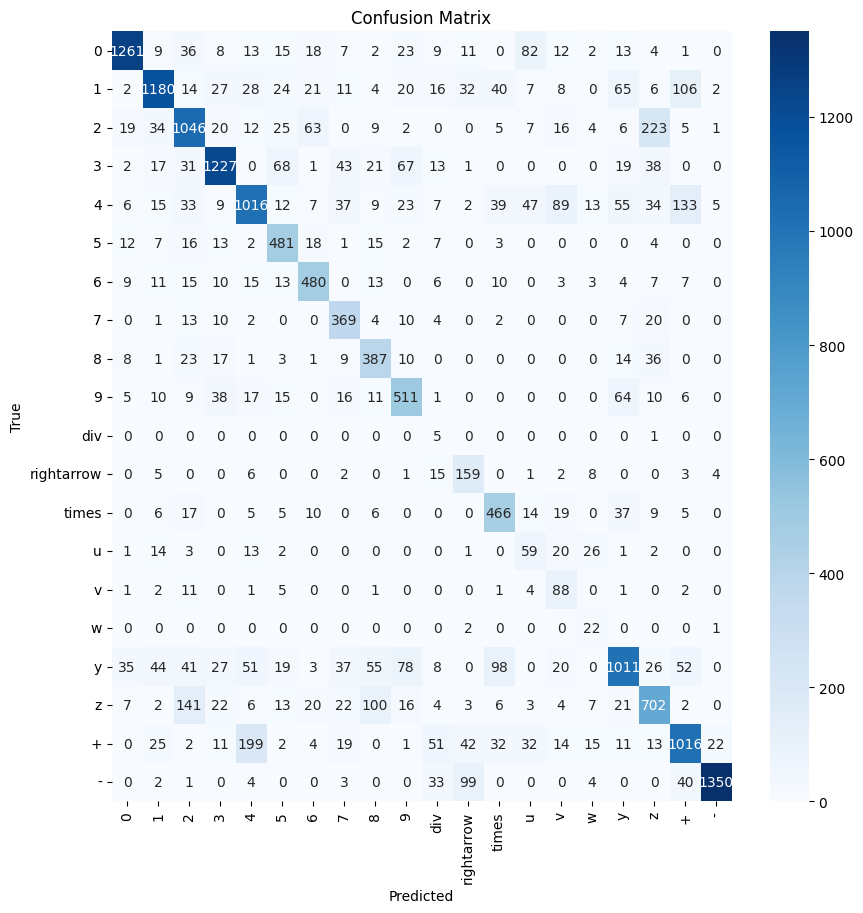
A graph of training and validation accuracy

Description automatically generated

A graph of training and validation

Description automatically generated





**Accuracy: 0.73**

**Classification Report:**

**precision recall f1-score support**

**0 0.83 0.92 0.87 1368**

**1 0.73 0.85 0.79 1385**

**2 0.70 0.72 0.71 1452**

**3 0.79 0.85 0.82 1439**

**4 0.64 0.73 0.68 1391**

**5 0.83 0.69 0.75 702**

**6 0.79 0.74 0.77 646**

**7 0.83 0.64 0.72 576**

**8 0.76 0.61 0.67 637**

**9 0.72 0.67 0.69 764**

**div 0.83 0.03 0.05 179**

**rightarrow 0.77 0.45 0.57 352**

**times 0.78 0.66 0.72 702**

**u 0.42 0.23 0.30 256**

**v 0.75 0.30 0.43 295**

**w 0.88 0.21 0.34 104**

**y 0.63 0.76 0.69 1329**

**z 0.64 0.62 0.63 1135**

**+ 0.67 0.74 0.70 1378**

**- 0.88 0.97 0.92 1385**

**accuracy 0.73 17475**

**macro avg 0.74 0.62 0.64 17475**

**weighted avg 0.74 0.73 0.72 17475**

Model 3:

#### Model 3: Conv1, relu, maxpool, flatten1,  Dense1, softmax, cross entropy

Number of Layers 5

----------

Layer 1

Layer Name -> conv\_1

Number of filters -> 8

Filters shape -> (8, 1, 3, 3)

Stride -> 1

----------

----------

Layer 2

Layer Name -> relu\_1

----------

----------

Layer 3

Layer Name -> max\_pool\_1

Pool size -> 2

Stride -> 2

----------

----------

Layer 4

Layer Name -> flatten\_1

----------

----------

Layer 5

Layer Name -> Dense\_1

Weights shape -> (3528, 20)

**Train and Validation metrics**

Epochs -> 10

Batch Size -> 64

Learning rate -> 0.01

Accuracy and Loss at the end of 11 epochs

Train Accuracy -> 66.41603611623323, Validation Accuracy -> 67.18168812589414

Train Loss -> 1.3163063072758185, Validation Loss -> 1.280657085268447

Test accuracy -> 68%

\*\*\*\*\*\*\*\*\*\*

Epochs -> 30

Batch Size -> 64

Learning rate -> 0.01

Accuracy and Loss at the end of 30 epochs

Train Accuracy -> 81.16614738983913, Validation Accuracy -> 81.40200286123033

Train Loss -> 0.7258183737357372, Validation Loss -> 0.7287216197045958

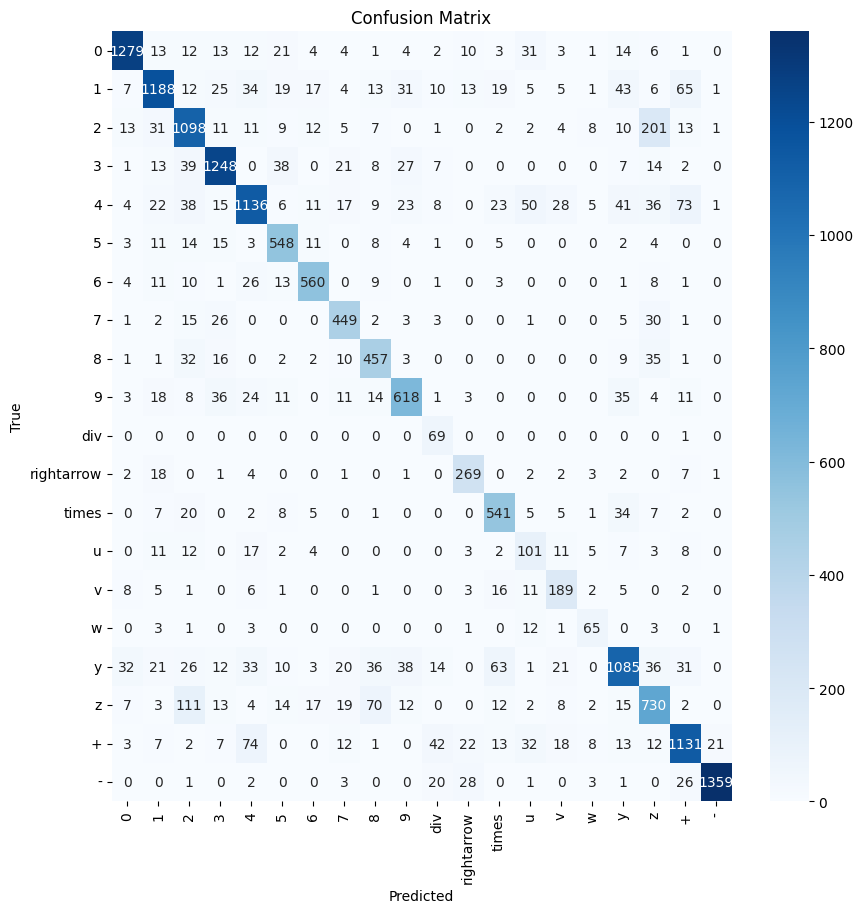
Test Accuracy -> 81%

**Observations for Model 3:**

1. Took more time to train as convolution operation was involved.
2. Achieved similar metric to Model 1.2 but took longer time for each epoch.
3. Observed a spike in loss, during the 11 epoch

A graph with a blue line

Description automatically generated



Accuracy: 0.81

Classification Report:

precision recall f1-score support

0 0.89 0.93 0.91 1368

1 0.78 0.86 0.82 1385

2 0.76 0.76 0.76 1452

3 0.88 0.87 0.87 1439

4 0.73 0.82 0.77 1391

5 0.87 0.78 0.82 702

6 0.86 0.87 0.87 646

7 0.83 0.78 0.81 576

8 0.80 0.72 0.76 637

9 0.78 0.81 0.79 764

div 0.99 0.39 0.55 179

rightarrow 0.86 0.76 0.81 352

times 0.85 0.77 0.81 702

u 0.54 0.39 0.46 256

v 0.76 0.64 0.69 295

w 0.72 0.62 0.67 104

y 0.73 0.82 0.77 1329

z 0.70 0.64 0.67 1135

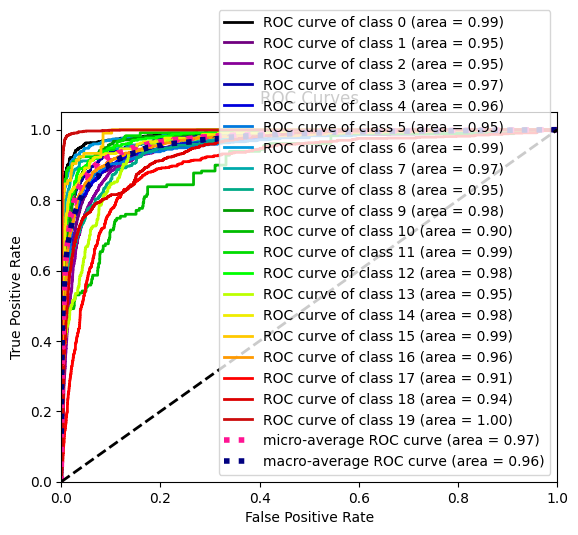
+ 0.80 0.82 0.81 1378

- 0.94 0.98 0.96 1385

accuracy 0.81 17475

macro avg 0.80 0.75 0.77 17475

weighted avg 0.81 0.81 0.81 17475



Model4:

Conv2, relu1, maxpool1, conv1, relu2, maxpool2, flatten1, dense1, softmax, cross entropy

Observations:

1. Performed better than CNN model with single layer.
2. A single epoch took nearly an hour on 16GB RAM computer.
3. The saved pickle object of this network is around 2GB.

\*\*\*\*\*\*\*\*\*\*

**Layers**

Number of Layers 8

----------

Layer 1

Layer Name -> conv\_1

Number of filters -> 8

Filters shape -> (8, 1, 3, 3)

Stride -> 1

----------

----------

Layer 2

Layer Name -> relu\_1

----------

----------

Layer 3

Layer Name -> max\_pool\_1

Pool size -> 2

Stride -> 2

----------

----------

Layer 4

Layer Name -> conv\_2

Number of filters -> 12

Filters shape -> (12, 8, 5, 5)

Stride -> 1

----------

----------

Layer 5

Layer Name -> relu\_2

----------

----------

Layer 6

Layer Name -> max\_pool\_2

Pool size -> 2

Stride -> 2

----------

----------

Layer 7

Layer Name -> flatten\_1

----------

----------

Layer 8

Layer Name -> Dense\_1

Weights shape -> (768, 20)

----------

Train and Validation metrics

Epochs -> 20

Batch Size -> 128

Learning rate -> 0.1

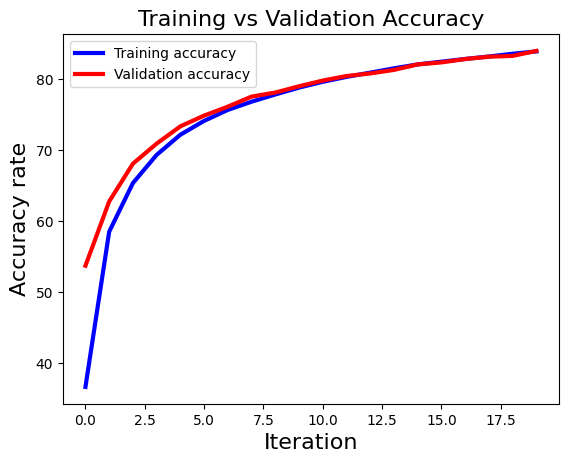
Accuracy and Loss at the end of 21 epochs

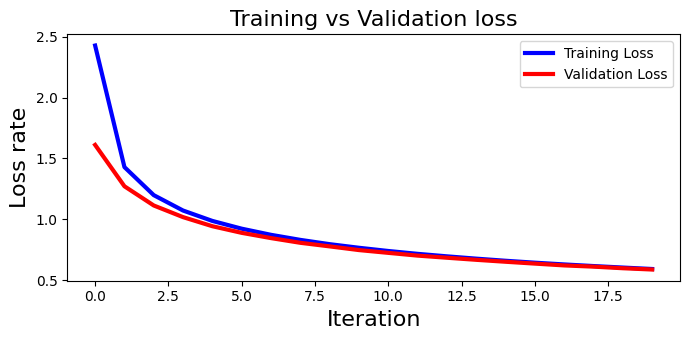
Train Accuracy -> 83.9209639473517, Validation Accuracy -> 83.99141630901288

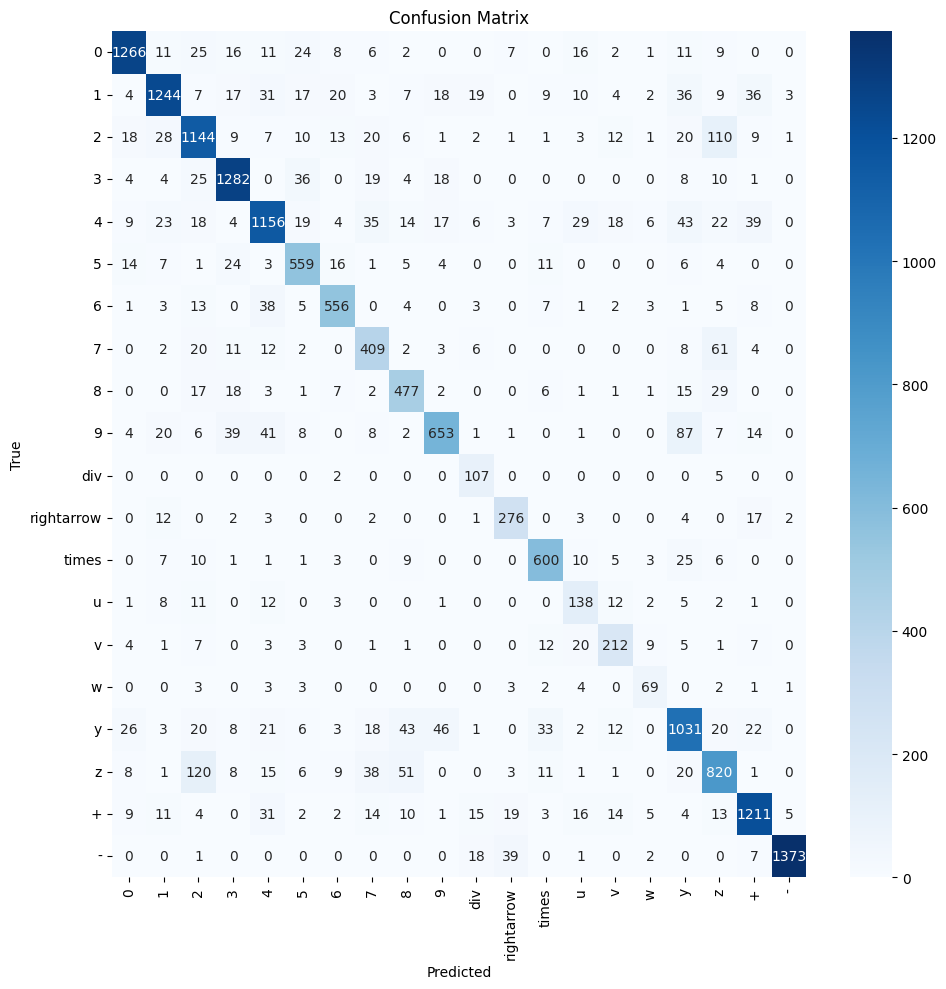
Train Loss -> 0.5917024023176323, Validation Loss -> 0.5874054145405015

Test accuracy, Test loss -> (83.45064377682402, 0.5987922885351344)

Metrics for model 4







Accuracy: 0.83

Classification Report:

precision recall f1-score support

0 0.89 0.93 0.91 1368

1 0.83 0.90 0.86 1385

2 0.81 0.79 0.80 1452

3 0.91 0.89 0.90 1439

4 0.79 0.83 0.81 1391

5 0.85 0.80 0.82 702

6 0.86 0.86 0.86 646

7 0.76 0.71 0.73 576

8 0.82 0.75 0.78 637

9 0.73 0.85 0.79 764

div 0.94 0.60 0.73 179

rightarrow 0.86 0.78 0.82 352

times 0.88 0.85 0.87 702

u 0.70 0.54 0.61 256

v 0.74 0.72 0.73 295

w 0.76 0.66 0.71 104

y 0.78 0.78 0.78 1329

z 0.74 0.72 0.73 1135

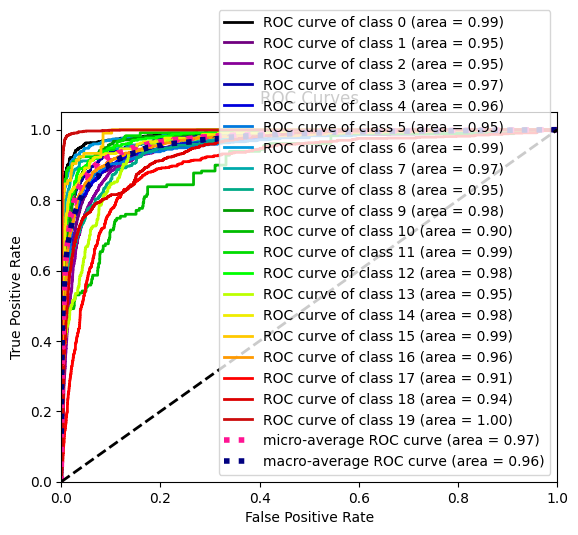
+ 0.87 0.88 0.88 1378

- 0.95 0.99 0.97 1385

accuracy 0.83 17475

macro avg 0.82 0.79 0.80 17475

weighted avg 0.83 0.83 0.83 17475



Visualizing activations for Model 4 –

A collage of numbers

Description automatically generated

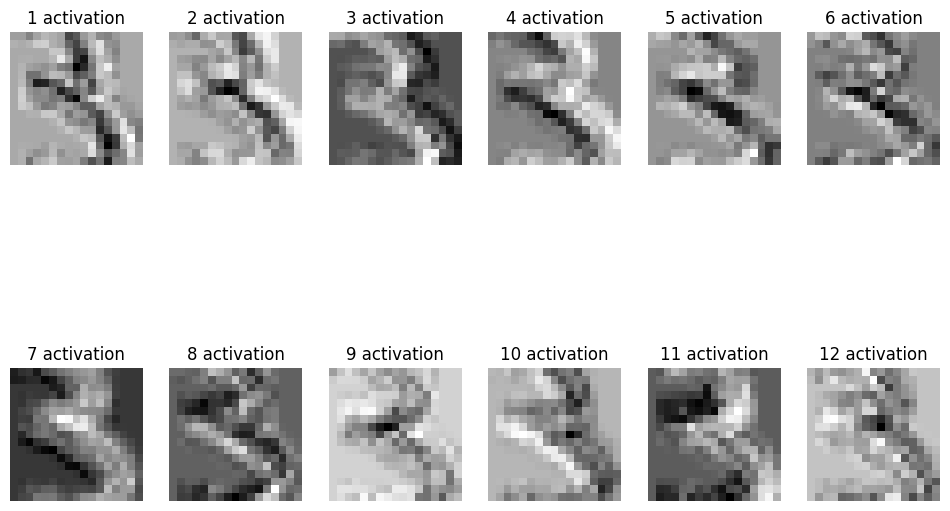
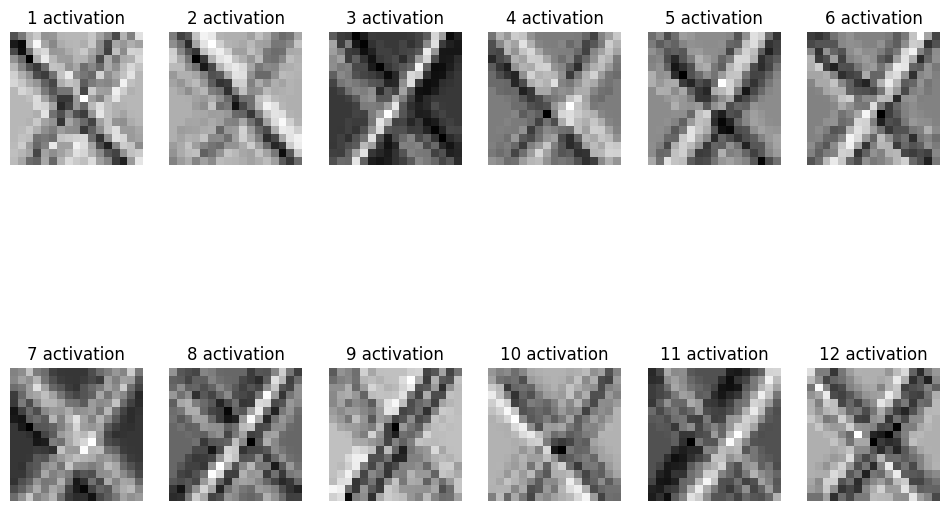
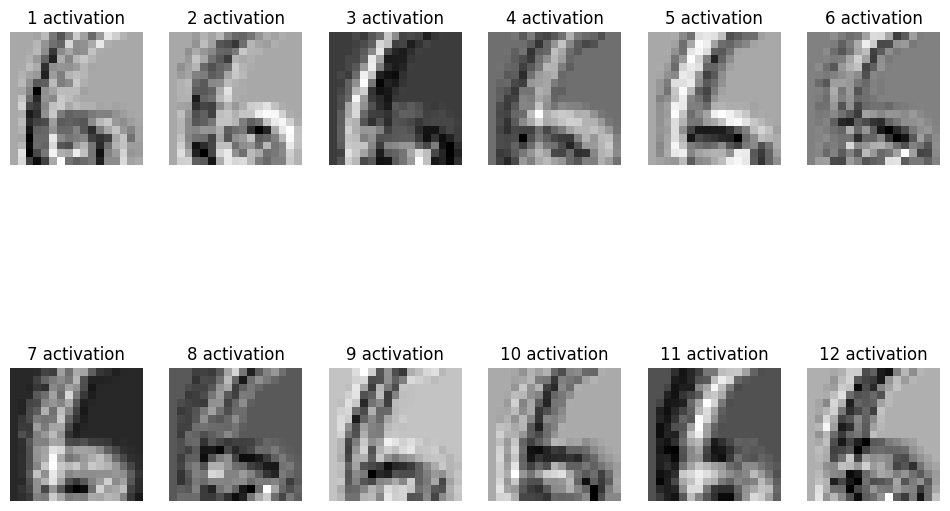
A collage of x symbols

Description automatically generated

Convolutional layer 1 output activations

A collage of images of numbers

Description automatically generated

Convolutional layer 2 output activations

**Filter activations**

Convolutional layer 1 filters output

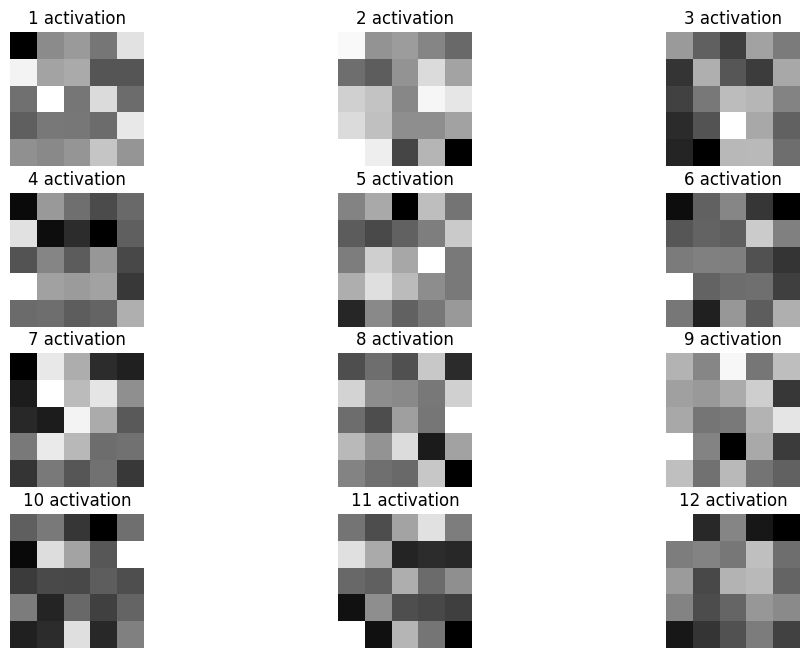
8 filters of size 1,3,3

A diagram of different types of squares

Description automatically generated with medium confidence

Convolutional layer 2 filters output

12 filters of size 8,5,5 (channels, w, h). Here we are visualizing only 12x1x5x5



Conclusion and future work:

1. The best performing model on real world images didn’t work as well as it did on test set. So there might be some issue while preprocessing real-images before feeding into the model. So, the model is overfitting as it’s not working on real world images.
2. More work need to be done on dataset preparation as the model on real world images weren’t good, so we need to augment the images for the classes which are of less numbers.
3. Forward propagation is vectorized in CNNs but during backward propagation we can vectorize the code even for calculating the gradients for each channel.
4. Currently we only solve linear equations with single variables, we can work on extending the equation solver for quadratic and trigonometric equations.
5. CNN layers are pretty slow, so we can look into improving the training speed of these layers by using Cython and vectorizing the code over the image channels.

References

<https://www.parasdahal.com/softmax-crossentropy>

<https://jmlb.github.io/ml/2017/12/26/Calculate_Gradient_Softmax/>

<https://www.youtube.com/watch?v=5-rVLSc2XdE&ab_channel=SmartAlphaAI>

<https://pyimagesearch.com/2021/05/06/understanding-weight-initialization-for-neural-networks/>

<https://www.youtube.com/watch?v=3TdBtI9dh2I>

<https://deeplearning.cs.cmu.edu/F21/document/recitation/Recitation5/CNN_Backprop_Recitation_5_F21.pdf>

<https://www.educative.io/answers/how-to-backpropagate-through-max-pooling-layers>

<https://pyimagesearch.com/2015/04/20/sorting-contours-using-python-and-opencv/>