**Mohit Kaushik**

**205320011**

**FOSS LAB Project1 Report**

**Handwritten Character Recognition using CNN**

# Problem Definition

The main objective of this project is to solve the problem of handwritten character recognition. It is a multi-class image classification problem where the task is to correctly recognize the given handwritten character (the character can be a digit (0-9) or a capital alphabet (A-Z)).

Character recognition, usually abbreviated to optical character recognition or shortened OCR, is the mechanical or electronic translation of images of handwritten, typewritten or printed text (usually captured by a scanner) into machine-editable text. It is an open problem in the fields of computer vision and deep learning. It is a problem which looks easy, but is hard to implement. Even with so many advances in the fields of computer vision and deep learning, 100% accuracy in this problem has not yet been achieved.

This project targets an easier problem than proper handwriting recognition. Here, the objective is to recognize separate characters rather than cursive handwriting.

Since image processing and training neural networks is generally a heavy task, and given the large training set size, parallel computing via CUDA for training the network on GPU has also been explored in this project.

# Analysis

The problem is approached using Convolutional Neural Networks (CNNs) and coded in Python. The framework used for CNNs is Pytorch, which is an open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab.

2 datasets have been combined to form the training data for this problem. The first one is the MNIST dataset containing 60,000 images for handwritten digits. The second one is a modified version of the NIST Special Database 19, called the Kaggle A-Z dataset (by Sachin Patel). It contains 3,72,450 images of handwritten alphabets (A-Z) in a CSV

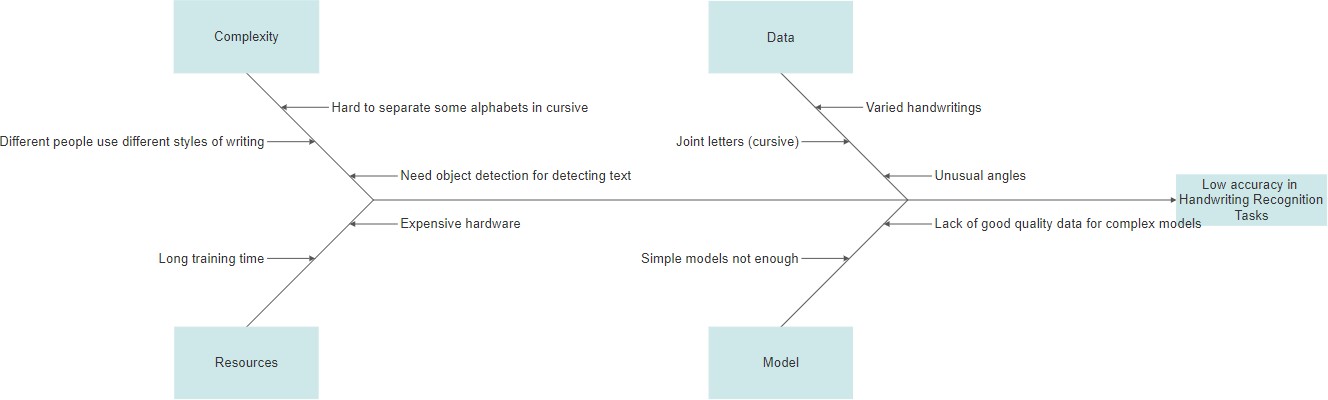
format, making it easy to load and pre-process data. Each of these datasets contains grayscale images (1-channel) of shape 28x28.

The model developed follows a CNN architecture with Convolutional layers for feature extraction, Pooling and Dropout layers for regularization (to prevent overfitting) and finally Fully Connected layers for classifying the images. The model has a bit more than 5 Million trainable parameters.

The model uses a Negative Log Likelihood loss function, which is a commonly used loss function for image classification tasks. The optimizer used is Adam, which is known to provide better results than simple optimizers like SGD.

The output of the model is log-probabilities for each class. The maximum of these is taken as the predicted class for the image.

This model is not meant for cursive handwriting. It is meant to classify only **single capital English letters (A-Z)** and **digits (0-9)**.



To achieve a desirable accuracy, taking advantage of the fact that training data is abundant, a bit complex architecture comprising several Convolutional and Dense layers has been constructed. To minimize training times on this complex architecture, the model has been trained on a GPU via Pytorch’s API for CUDA.

# Design

The CNN architecture developed contains the following layers:

1. 2D Convolutional Layer (1, 32, 5x5)
2. Max Pooling Layer (2x2)
3. 2D Convolutional Layer (32, 64, 5x5)
4. 2D Convolutional Layer (64, 128, 3x3)
5. Dense Fully Connected Layer (4608, 1024)
6. Dropout Layer (P = 0.3)
7. Dense Fully Connected Layer (1024, 256)
8. Dropout layer (P = 0.3)
9. Dense Fully Connected Layer (256, 128)
10. Dense Fully Connected Layer (128, 36)

The first layer is a 2D Convolutional layer with 1 input channel, 32 output channels and a kernel size of 5x5. Then, there is a max pooling layer for regularization with both kernel size and stride of 2. This is followed by 2 more Convolutional layers, the first one with 32 input channels, 64 output channels and 5x5 kernel, and second one with 64 input channels, 128 output channels and 3x3 kernel. This architecture is based on the intuition that the earlier layers will learn high level features while the deeper layers will learn more specific and complex features and thus have more channels.

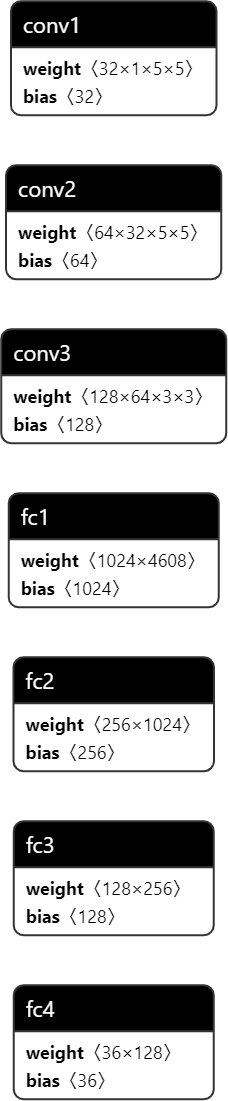
These Convolutional layers are then connected to Dense layers. The first one is a layer with 4608 input neurons and 1024 output neurons. Then, a dropout layer with probability = 0.3 is placed, to prevent overfitting. Then another pair of Dense and Dropout layers is present. Finally, two more Dense layers are placed. The output layer contains 36 contains, each of which corresponds to one of the class (10 nodes for digits and 26 nodes for alphabets).

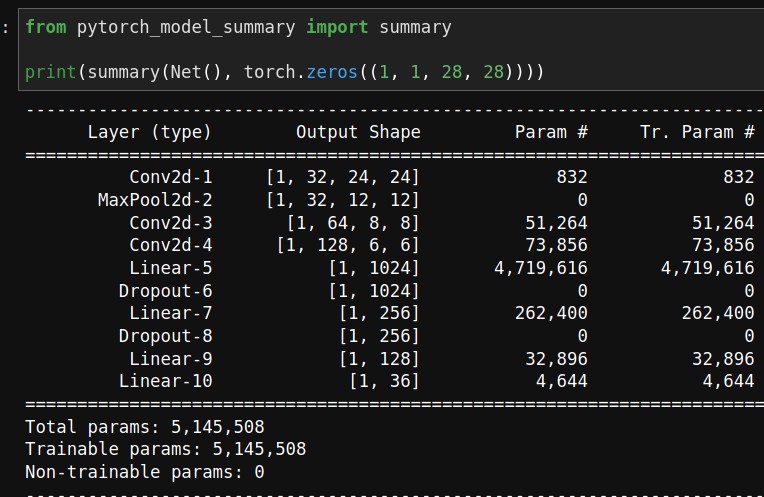
The activation function used throughout is Rectified Linear Unit (ReLU), because of its known efficient performance and fast convergence. To calculate likelihoods, *log\_softmax* activation function has been used in the output layer, with the loss function being *Negative Log-Likelihood* and optimizer being *Adam*.

*Negative Log-Likelihood* is chosen because of its numerical stability and suitability to this problem of image classification. *Adam* is chosen because of its more efficient performance than simple optimizers like Stochastic Gradient Descent.

The images are first preprocessed to make the training process more efficient. The pixel values initially range from 0-255. They are first brought to 0-1 range by dividing all pixel values by 255. Then, they are normalized. These steps help in faster convergence and training process.

A chart for the model architecture is shown below. *Conv* refers to Convolutional layers and *fc* refers to Dense Fully Connected layers.





The Pytorch framework provides Data Loaders to make the process of passing the input in batches to the model smooth and easy. The entire dataset (digits + alphabets) has been first wrapped in the data loader. 3 transformations have been applied to the data:

1. divide the pixel values by 255
2. normalize the pixel values
3. convert the resulting numpy matrices to pytorch tensors.

Then, the data is shuffled and splitted into training and validation sets. Since the number of samples is huge (greater than 4 lakh images), a majority of the dataset has been kept for training (4,00,00 images) and the remaining images have been taken as the validation set (32,451 images).

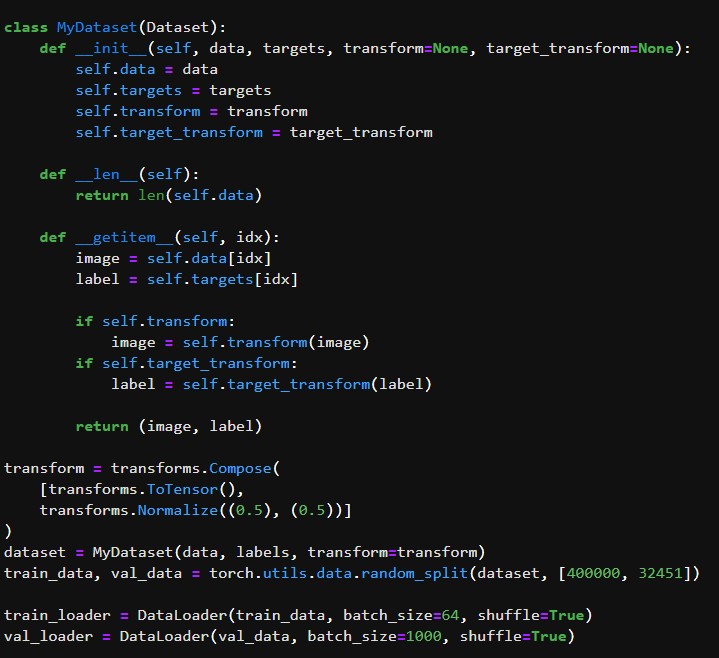
After loading the data in Data Loader, the model has been trained for 2 epochs by passing the input in batches of size 64 to the model. One iteration involves passing the batch, calculating outputs, then calculating loss, and then passing that loss backwards in the network to adjust the gradients (i.e. taking a step of optimizer).

# Implementation and Testing

As stated earlier, the project is implemented using Python. The CNN model is built using Pytorch. The input images for training the model are stored in *inputs* folder. Training script is stored in *src* folder, while the modules for testing the model have been stored in a Jupyter Notebook stored in *notebooks* folder. Any custom images to

be tested can be placed inside the *custom\_images* folder. The trained model weights are stored in *models* folder.

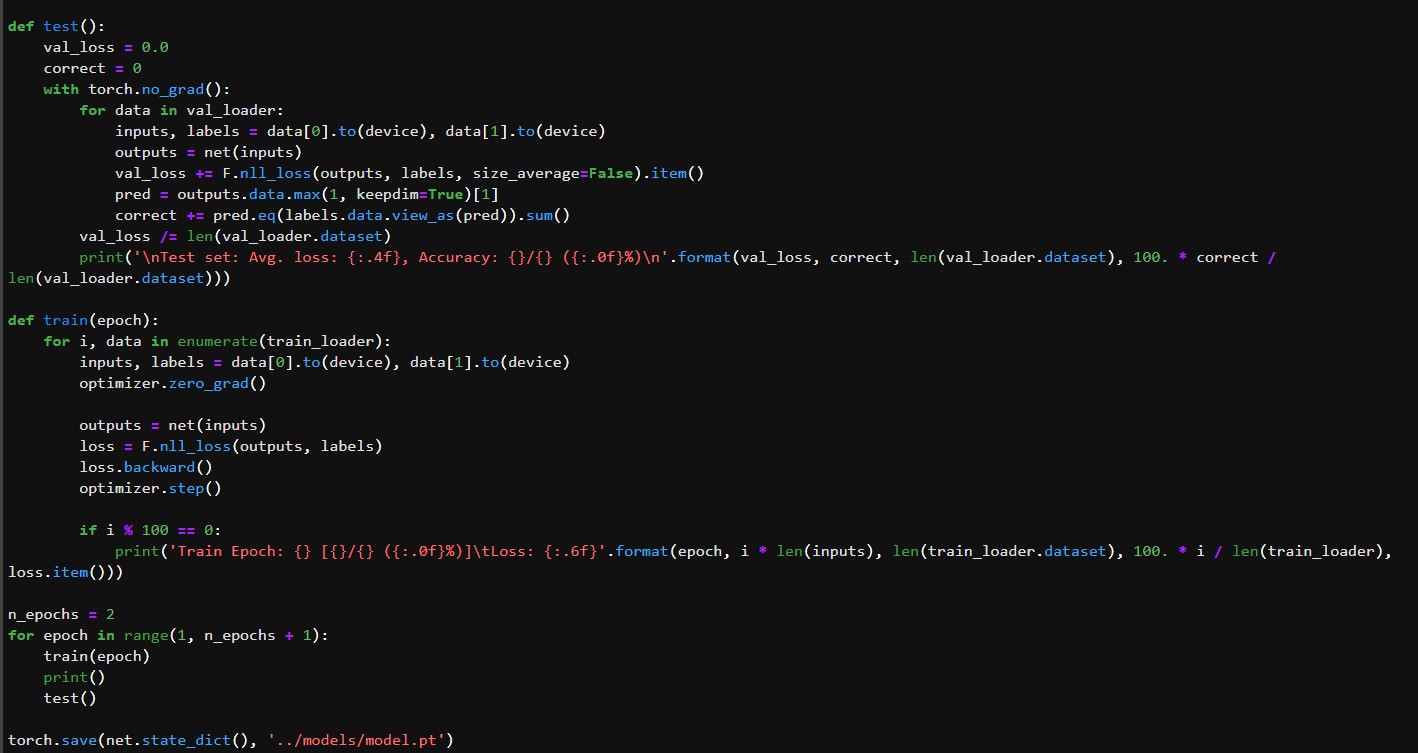
For training, a 6GB Nvidia GeForce GTX 1660Ti GPU was used. The code has been written in such a way that it will automatically detect if CUDA is available and will train on GPU, otherwise it will use CPU.



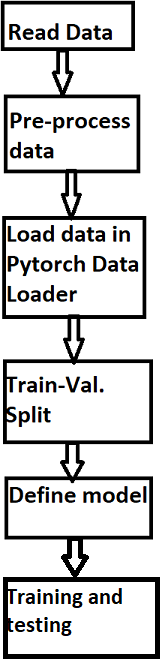
The above code first wraps the data inside a *Dataset* class, as required by Pytorch Data Loaders. Then, the data is split into training and validation sets (4,00,000 and 32,451 examples respectively). Finally, both the training and validation datasets are passed into *DataLoader*.



Then, the above code defines the CNN architecture used in this project. All the layers have already been described earlier. It also sets the optimizer to *Adam* and device to *CUDA* for training the model on GPU.



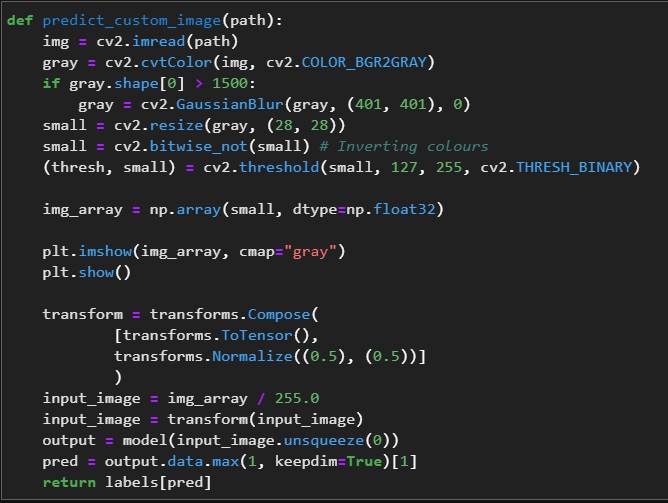
The training process involves first obtaining the current batch via the Pytorch Data Loader(the batch size has been set to 64, i.e. on a single iteration, 64 images will be passed to the model for efficient computation). The batch size can be increased depending upon the RAM and other computing resources available. Then, if CUDA is available, the data (images and the corresponding labels) are transferred to the GPU. The outputs are calculated via the current weights of the network, and the loss is computed via Negative Log Likelihood loss function. Then, a backward step is taken for training by the Backpropagation algorithm. The weights of the model are adjusted according to the loss. The optimizer function used for this is Adam. This process is repeated for 2 epochs over the entire training set (thus a total of 2 x 4,00,000 = 8,00,000 times). Since the training set is huge, the training process is observed to be much faster when run on a GPU than a CPU.



For testing on the validation set, again the data is first transferred to GPU (if available). Then the outputs are calculated by passing the input to the model. The model outputs log likelihoods. For getting the output label, the maximum of these likelihoods is taken.

Testing on custom images is a bit more complex, since most modern cameras take high resolution RGB (3-channel) pictures. First, the images are reduced from 3 channels to

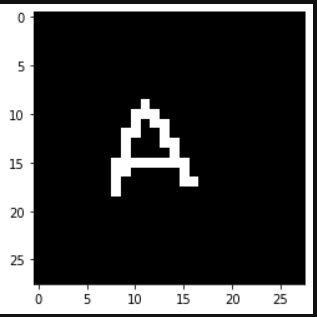
1 channel (i.e. from RGB to grayscale). If the images are of a very high resolution (greater than 1500 pixels), then Gaussian Blurring is applied to smoothen the image. Then, the images are reshaped to 28x28 pixels (since the model was trained on 28x28 shape images). Normally, custom images will have a white background (white paper) and black ink, but the model had images with black background and white ink. So, the colours of all images are inverted (so that they have black background with white ink on top). Then, to sharpen the image and remove noise, all pixels with a value above 127 are converted to 255 (white) and below 127 are converted to 0. i.e. the image is converted to pure black and white to remove all noise. Finally, the transformations applied to training images are applied to these images too, i.e. pixel values are divided by 255, normalized and converted to Pytorch tensors. Finally, prediction is made using these tensors. Pytorch Data Loaders have not been used when testing the model on individual images.



Original image:



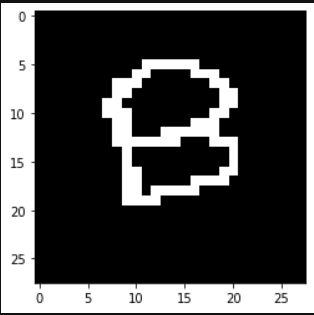
Pre-processed image:



Original image:



Pre-processed image:



For best results, the custom images should have less noise (background must be as clean as possible), and the ink used must be thick, preferably a sketch pen instead of a regular gel/ball pen (because thin ink combined with high resolution will lead to a poor quality image when resized to 28x28). The provided custom images were taken from a mobile camera producing images of resolution 3472x4624. The digits were written with a black marker on a whiteboard.

The model achieves an overall training accuracy of 98.2% and validation accuracy of 98%. Since the difference is not significantly large, it is verified that the model is not overfitting. The results can be further improved through techniques like image

augmentation, regularization, building a deeper architecture and getting more training data.

# Summary

In this project, a CNN model with more than 5 million parameters was successfully trained to recognize single handwritten capital English alphabets (A-Z) and digits (0- 9). The model achieves a satisfactory accuracy on the dataset and performs reasonably well on custom images. Performance on custom images can be improved through various steps described earlier. Further, it was noticed that the training time was significantly shorter when the model was trained on GPU than CPU. This model classifies only single characters. To classify a complete line of text consisting both alphabets and digits (in non-cursive form), this program can be extended via *opencv*’s functionalities and some pre-built object detection models to detect where the text is written, isolate them and classify each of the characters separately.

# Future Works

The model developed here classifies single characters, and has a relatively simple architecture. It can be improved with a slightly more complex architecture, but too complex an architecture will lead to poor results since the training data for digits is quite less (60,000 examples). A more complex architecture with 4 Convolutional layers with upto 256 channels and 5 Fully Connected layers was also tried, but it only achieved an accuracy of 90% on MNIST data due to lack of data, and was thus discarded.

This problem can be fixed to some extent by image augmentation. State-of-the-art Deep Learning models have achieved > 99% accuracy on MNIST dataset, and the same can be explored here for alphabets as well. Pre-trained networks like the ResNet can be used and Transfer Learning can be leveraged to achieve near perfect accuracies.

The next natural step is to extend this problem to *cursive handwriting recognition*. It is a much more complex problem and requires the use of more complex architectures like LSTMs.

The learning outcomes of this project include how various deep learning architectures work, their training process, image processing in python using *opencv* library and the use of deep learning library Pytorch. Also, training neural networks on GPUs using CUDA was explored and it was found that it led to a significant speedup, and thus allowed for easy experimentation with various architectures.

# References

* Official Pytorch documentation - <https://pytorch.org/tutorials/>
* Notes from Stanford’s course CS231n - https://cs231n.github.io/
* [https://www.thinkautomation.com/bots-and-ai/why-is-handwriting-recognition-](https://www.thinkautomation.com/bots-and-ai/why-is-handwriting-recognition-so-difficult-for-ai/) [so-difficult-for-ai/](https://www.thinkautomation.com/bots-and-ai/why-is-handwriting-recognition-so-difficult-for-ai/)
* OpenCV tutorials - [https://opencv-python-](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_table_of_contents_imgproc/py_table_of_contents_imgproc.html) [tutroals.readthedocs.io/en/latest/py\_tutorials/py\_imgproc/py\_table\_of\_contents](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_table_of_contents_imgproc/py_table_of_contents_imgproc.html)

[\_imgproc/py\_table\_of\_contents\_imgproc.html](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_table_of_contents_imgproc/py_table_of_contents_imgproc.html)

# Links to Datasets Used

* MNIST: <https://www.kaggle.com/oddrationale/mnist-in-csv>
* Modified NIST Special Database 19: https://[www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format](http://www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format)