Clab-2 Report
ENGN4528

Kirat Alreja
u7119530

02/05/2022

Task 1 : Harris Corner Detector

Approach & equations used in the algorithm

The given framework provides the following functions and resources :

1. `conv2` - takes an image and a convolution filter, and applies the convolution operation on the image with the given filter.
2. `fspecial` - creates a 2-D gaussian filter, with a 3x3 kernel and given sigma value
3. Sobel derivatives - the framework takes a 3x3 sobel kernel and applies it to the image in both x & y directions
4. Gaussian blur on second derivatives - the framework further applies a gaussian kernel on second derivatives, generating three second order derivatives as Ix2, Iy2 & Ixy.

The resources and functions given here make it convenient to define & work with harris corner response equations. Harris corner works by calculating the shift in intensities in all directions of the image, which is fundamentally the reason we need second order derivatives to define the equation. I used this second order derivate matrix [1] (Figure 1), to define the harris response equation.

$$M = \sum w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Figure 1 - Second order derivates matrix, with window function [1]

Here (Figure 2), w(x,y) stands for the window of pixels being considered in the image. The determinant & trace of this matrix will give us the harris response matrix [1].

$$R = \det M - k(trace\, M)^2$$

Figure 2 - Harris Response Equation [1]

k is called the harris response free parameter, and is a constant value which is empirically determined. I used a value of 0.05 for my algorithm. The determinant and trace are calculated as follows (Figure 3) [1]

$$A = I_x I_x \oplus w$$
$$B = I_y I_y \oplus w$$
$$C = I_x I_y \oplus w$$

$$det(M) = AB - C^2 \; or \; \lambda_1 \lambda_2$$
$$trace(M) = A + B \; or \; \lambda_1 + \lambda_2$$

Figure 3 - Calculating the harris response equation paramters [1]

In this R matrix, we can determine edges, corners and flat regions by comparing the value of each indice to zero and defining a certain threshold to classify corners [1]. (Figure 4)

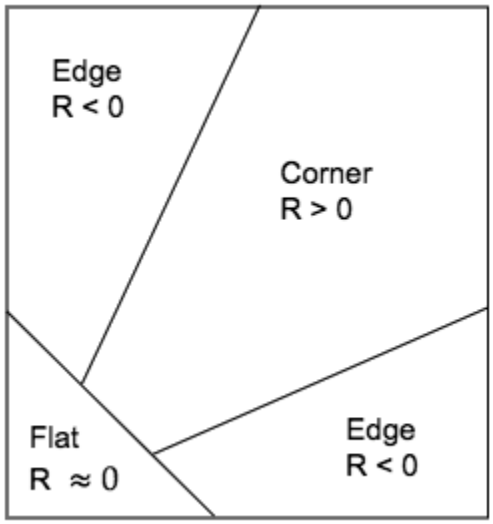

Figure 4 - Interpreting the harris response matrix values [1]

My algorithm implementation

The basic signature is as follows:

```
function harris (image, nms)

      Input
      image : image path
      nms : a boolean value to switch on/off non maximal suppression

      Output
      returns Nx2 array of coordinates of all the detected corners
```

Now, looking at some important code snippets of the implementation

Reading the image and preparing it for calculations [1]

```
#Reading the image with OpenCV in a (width,height,channel) format array, to
be able to convert it to grayscale by channel elimination
bw = cv.imread(image)

#Convert OpenCV's default BGR colorspace to RGB
bw = cv.cvtColor(bw, cv.COLOR_BGR2RGB)

#take only only a single channel
#essentially converting RGB images to grayscale
bw = bw[:,:,0]

#OpenCV’s default datatype is ‘unit8’ which is unsuitable for our operations.
Converting the same to ‘int64’
bw = np.array(bw * 255, dtype=int)
```

Preparing the derivatives

```
#sigma value for gaussian kernel, and threshold for corner detection
sigma = 2
thresh = 0.01

#defined the 3x3 sobel kernel
dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
dy = dx.transpose()

#convoluting Sobel kernel to in x direction
Ix = conv2(bw, dx)

#convoluting Sobel ketnel in y direction
Iy = conv2(bw, dy)
```

```
#Creates a 3x3 gaussian kernel with the given sigma value, which can be used in a convolution operation
#to create a gaussian blur on the sobel result, to smoothen & improve corner detection
g = fspecial((max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma)

#Applying the gaussian kernel on second derivatives of the sobel result in x and y directions
Iy2 = conv2(np.power(Iy, 2), g)
Ix2 = conv2(np.power(Ix, 2), g)
Ixy = conv2(Ix * Iy, g)

#constant value, emprically determined to calculate the harris response matrix
k = 0.05

#calculating the determinant & trace [1]
determinant = Ix2*Iy2 - Ixy**2
trace = Ix2 + Iy2

#calculating the harris response matrix [1]
r = determinant - k * trace**2

#calculating the maximum value in the harris response matrix [1]
max_r_value = r.max()

#an array where the corner indices will be valued at 1, while all the other indices will be 0
corners = np.zeros((r.shape[0], r.shape[1]))
```

Now, moving on to the actual algorithm for corner detection and non maximal suppression [4]

```
#check if non maximal supression is requested
   if nms:
       for i in range(r.shape[0] - 1):
           for j in range(r.shape[1] - 1):
#checks if the current response value is greater than the
threshold #and also greater than the four points surrounding it [4]

if r[i, j] > thresh * max_r_value and r[i, j] > r[i-1, j-1]
and r[i, j] > r[i-1, j+1] and r[i, j] > r[i+1, j-1]
and r[i, j] > r[i+1, j+1]:
                   corners[i, j] = 1

   else:
#if non maximal supression is not requested, then every value greater
#than the threshold is classified as a corner, without checking its
neighbours

       for i in range(r.shape[0]):
           for j in range(r.shape[1]):
               if r[i, j] > 0.01 * max_r_value:
                   corners[i, j] = 1
```

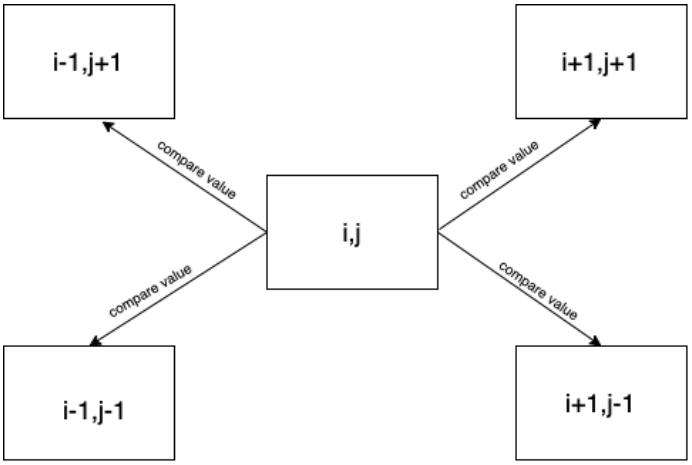

Figure 5 - My non-maximal suppression algorithm [4]

For non maximal suppression [4] (Figure 5), the current indice in the response matrix will be compared with the threshold multiplied by the maximum response value. This allows us to classify the point as a corner, and then the indice value is compared with four points in the neighbourhood. Finally, only the maximum value from the neighbourhood is selected as a corner and other values are ignored. Hence, redundant corners are removed & only one corner is kept from a set of overlapping corners. If non maximal suppression is not selected, only the threshold is compared & neighbourhood points are not considered.

Converting the points to a coordinate matrix

```
#get the coordinates of the corners, and store them in a NX2 matrix
   coordinates = []

#check the indices where corner matrix has the value one
  for i in range(corners.shape[0]):
     for j in range(corners.shape[1]):
           if (corners[i,j] == 1):
               coordinates.append((i,j))

   coordinates = np.asarray(coordinates)
```

The corners array has a value 1 at the indices which are corners, so I just used a simple loop to append all those indices to an array. Hence, the final result of the function is a Nx2 array of corner coordinates.

Testing my function on the given images

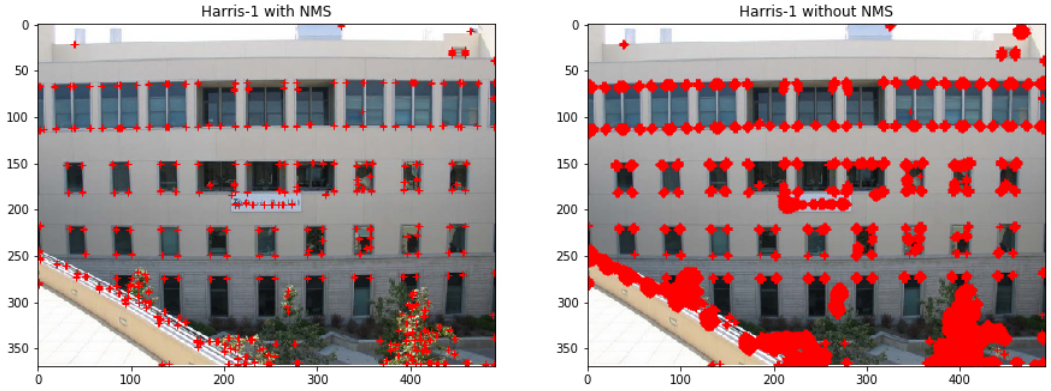

Figure 6 - Harris-1.jpg

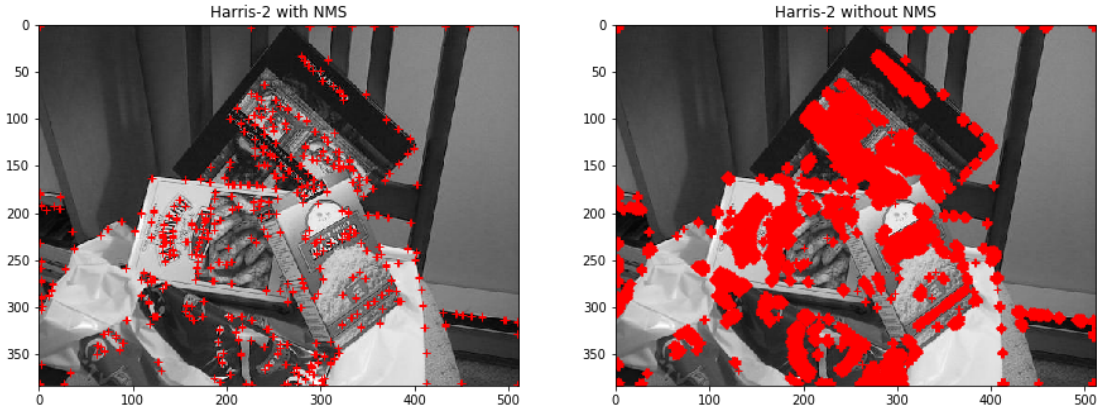

Figure 7 - Harris-2.jpg

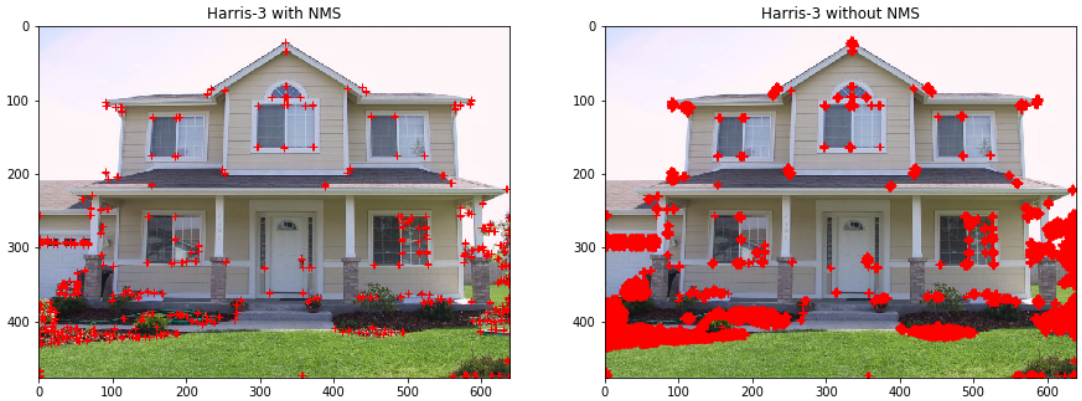

Figure 8 - Harris-3.jpg

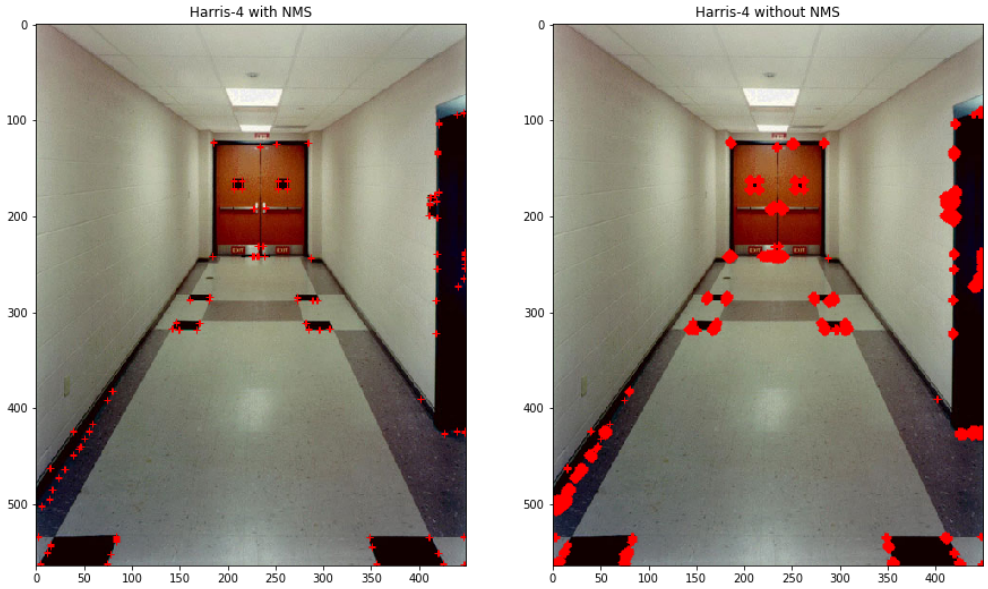

Figure 9 - Harris-4.jpg

Comparing my function to OpenCV

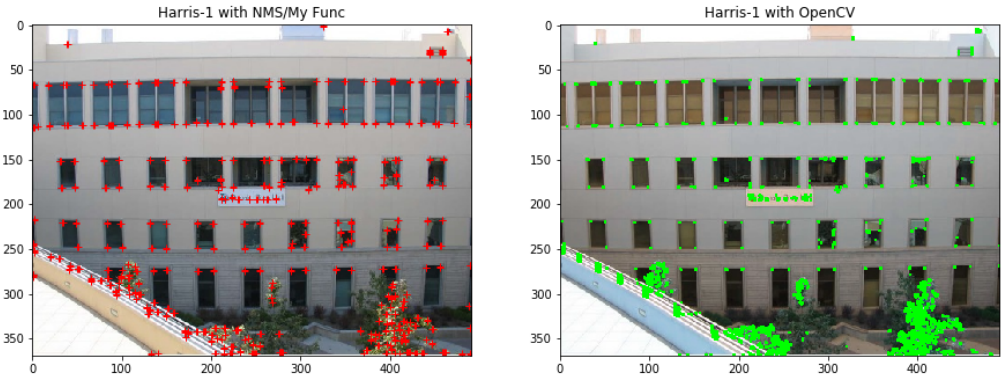

Figure 10 - Comparing Harris-1.jpg with OpenCV function

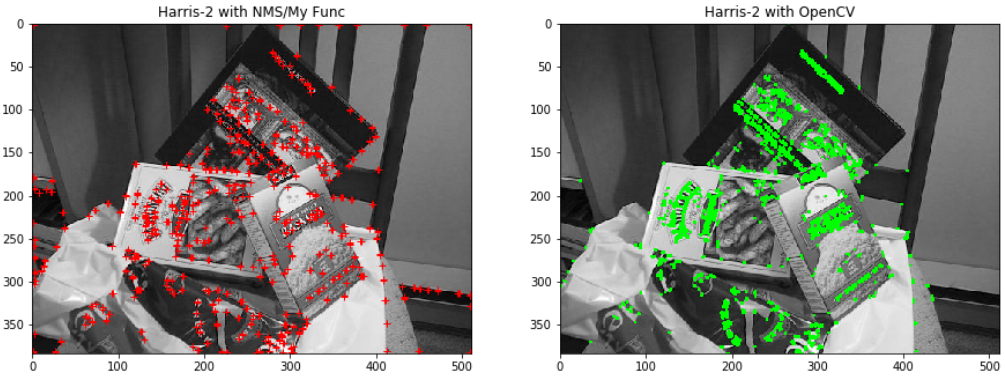

Figure 11 - Comparing Harris-2.jpg with OpenCV function

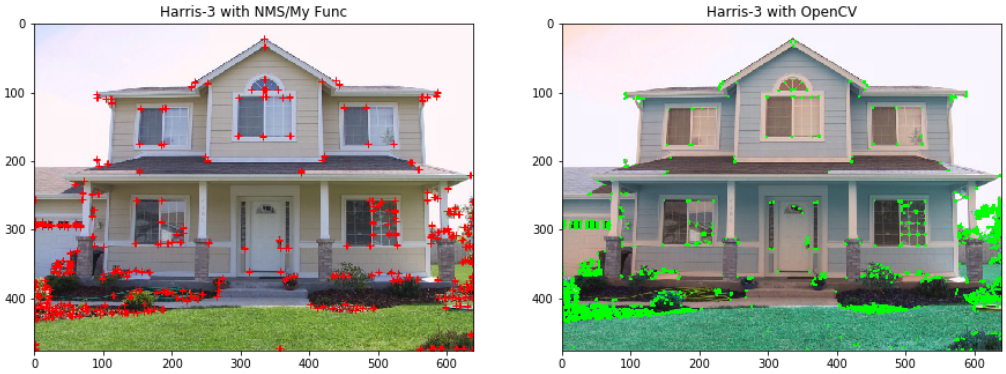

Figure 12 - Comparing Harris-3.jpg with OpenCV function

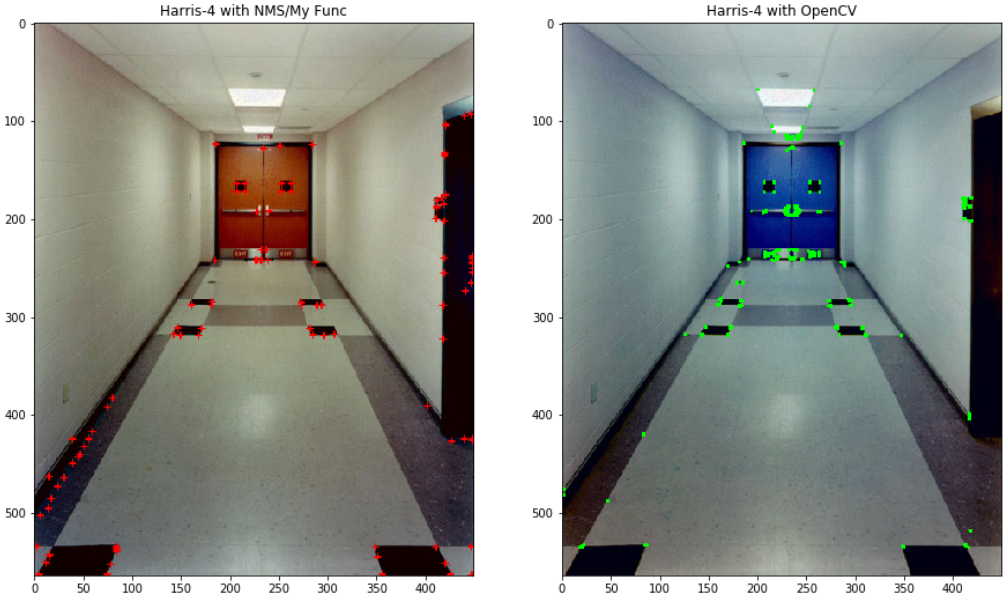

Figure 13 - Comparing Harris-4.jpg with OpenCV function

Performance of my algorithm

The algorithm seems to be doing a good job detecting the major corners and non-maximal suppression helps eliminate the redundant corners. However, still a lot of corners have been misclassified and missed all across the four images. (Figure 14 & Figure 15)

Figure 14 - Misclassifications in Harris-1 & Harris-2 images

Figure 15 - Misclassifications in Harris-3 & Harris-4 images

As an example, here are misclassified corners - where edges have been detected as corners, and one corner has multiple classifications even after non maximal suppression. Therefore, the algorithm does not perform ideally and can be improved. Now comparing my function with OpenCV, we can observe that the latter performs better (Figure 16 & Figure 17).

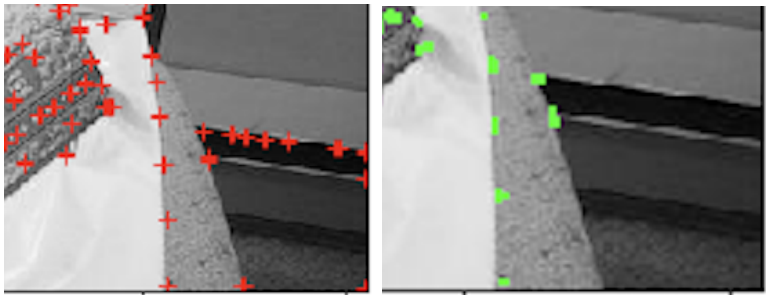

Figure 16 - OpenCV classifications in Harris-2.jpg

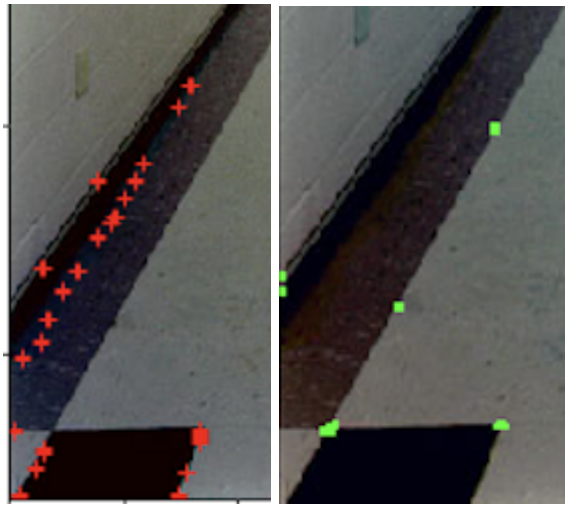

Figure 17 - OpenCV classifications in Harris-4.jpg

As it can be observed, the OpenCV function does not misclassify edges as corners and does a much better job at corner detection. However, even this function is not perfect and ends up classifying a couple of edge points as corners.

Factors that affect performance of Harris Corner detector:

1. Noise - The fundamental concept of harris corner is based on the change in intensities & if the image is noisy, that means there are intensity changes all over the image and not just corner points. That's why, to eliminate the effect of noise, we use a gaussian filter on the image before passing it to the response calculators. However, since gaussian filter is not always effective to eliminate noise - if the image has extreme noise, harris corner detector may only give mediocre results.
2. Scale - Harris corner detection algorithm is not scale invariant. That means if a region has been detected as a corner at one scale value, it might not be classified as a corner if we change the scale of the image. However, the good part is that this algorithm is rotation invariant. To fix the scale invariance, we can use a SIFT descriptor along with the algorithm and achieve correct results at any scale value.

References for Task 1

[1] : https://muthu.co/harris-corner-detector-implementation-in-python/
[2] : https://en.wikipedia.org/wiki/Harris_corner_detector
[3] : https://medium.com/data-breach/introduction-to-harris-corner-detector-32a88850b3f6
[4]:https://stackoverflow.com/questions/65047021/how-to-apply-the-non-maximum-suppression-for-the-corner-detection

Task 2

Setting up the base model (Figure 1) (Code references : [1] [2])

To set up the given model as per instructions given in the framework, I used the PyTorch library. First, I used `transforms.Compose` to sequentially combine all the transform operations, namely - Normalisation between (-1,1), Random horizontal flips, padding with 4 zeros, and random crops with 28x28 dimension.

Then, to load the data I used `datasets.KMNIST` and made the training/validation split. The final pre-processing step was to use `DataLoader` to set up the `batchsize` & `shuffle` parameters for training. I used a `batchsize` of 4 and `shuffle` as `True`. The `shuffle` parameter shuffles the data at the start of every `epoch`.

Then, to build the CNN architecture, I used a new class with `nn.Module` as an argument. I set up the layers as given in the framework in the `__init__` function, and then combined them using a `forward` function. Finally, I made an instance of this new class called `cnn` and the architecture was complete.

For the training process, I set up the loss function as `CrossEntropyLoss` and optimizer as `Adam optimizer`. The training loop uses batches of data from the data loaders to perform training on training data and model evaluation on the validation data. First, for every `epoch`, it takes an image & corresponding label and makes a prediction using CNN model. The loss function calculates a loss value, and then we reset the optimizer to zero gradient to clear any results from previous iteration. `loss.backward()` is used to calculate the new derivate of the loss with respect to supplied parameters and this information is used to take a optimisation step using `optimizer.step()`. Thus, the model is trained. After the training iteration is complete, there is a validation loop that calculates the loss & accuracy of the validation dataset.

Finally, I calculate the accuracy of the CNN model on the testing data using `cnn.eval()`. Along the entire process, I have used `TensorBoard` to write the results of training & validation loss as well as accuracy. The results are shown in Figure 2.

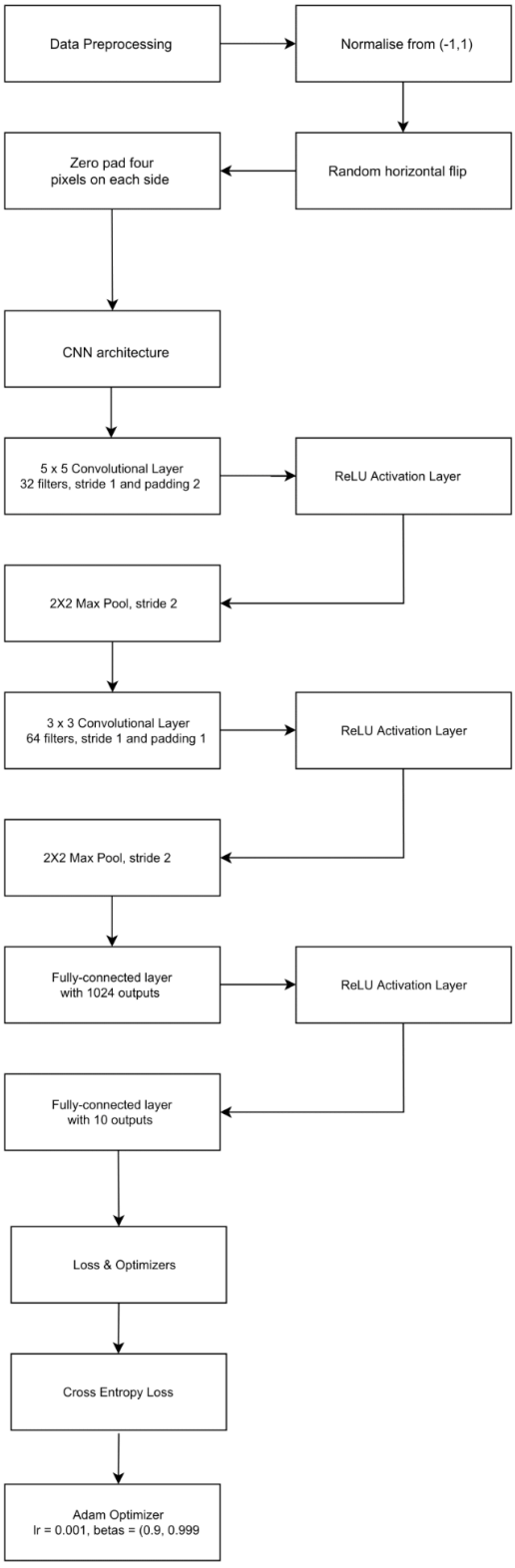

Figure 1 - Base model CNN architecture

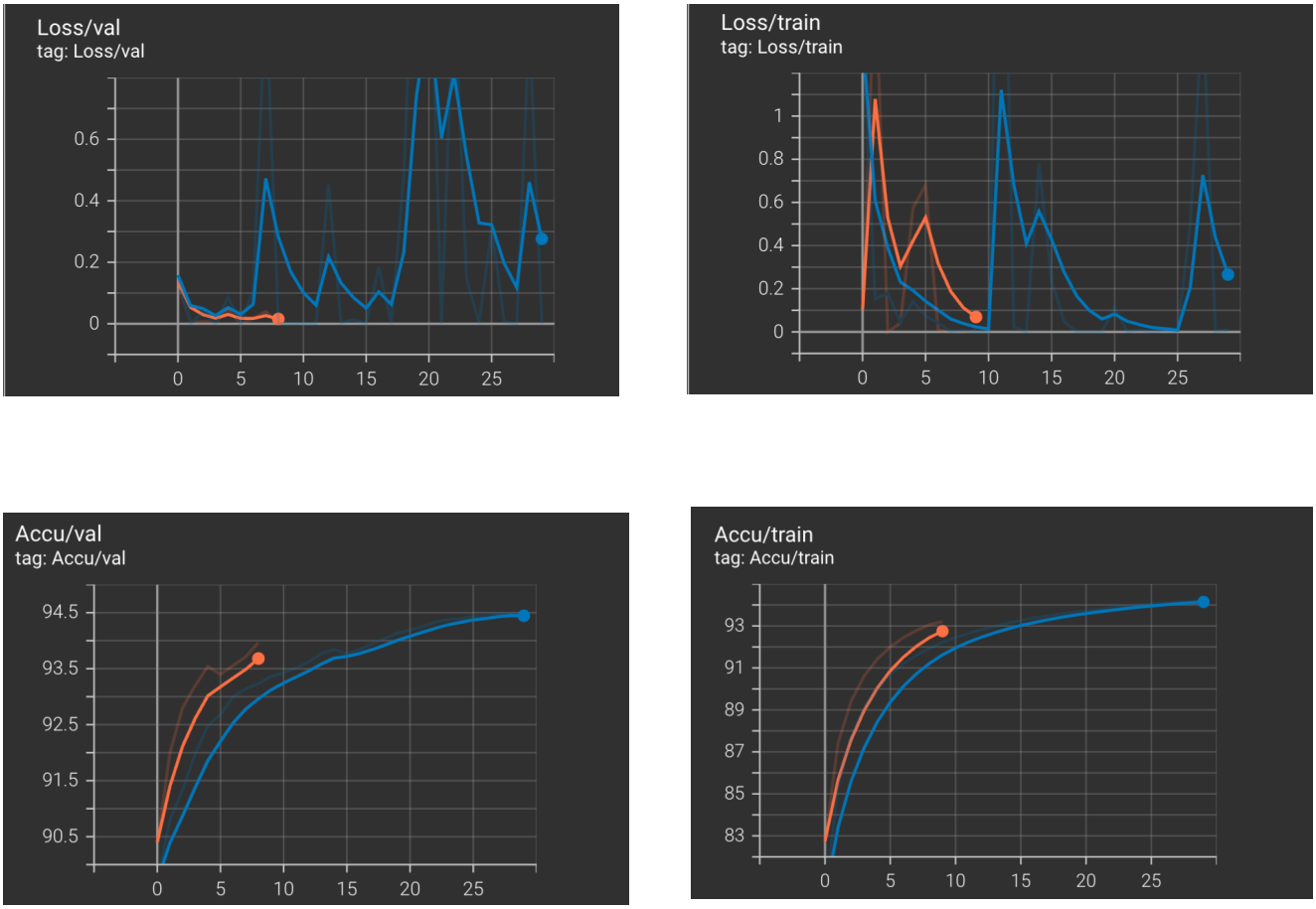

Figure 2 - Training, Validation Loss & Accuracy

Result Analysis

Testing accuracy = 0.75%, Training time for 30 epochs = 140 minutes (Apple M1 CPU)
Observing the loss graphs for validation & train loss, we can observe that there is a significant fluctuation throughout the training process and the numbers are quite high, especially for training. To observe the effect of increasing epochs on the training process, I used 10 epochs (orange line) & 30 epochs (blue line) to train separate models. The loss seemed to have reached a minimum at the end of 10 epochs of training, and started violently fluctuating when the number of epochs was increased. However, the opposite can be observed in the accuracy graphs. The model has pretty high accuracy values, with 10 epochs giving an accuracy of about 94% and 93% for validation and training respectively. With increasing epochs to 30, the accuracy shot up to about 94.5% & 94% for validation & training respectively. In conclusion, for both loss & accuracy, increasing the number of epochs from 10 to 30 did not make much of a difference - which shows that a high epoch number isn't always helpful and there is a saturation point for models. The testing accuracy is about 75%, which is quite mediocre. In the next part, I try to improve the model by changing the architecture.

Improved model (Figure 3) [Architecture reference : 3]
The base model has mediocre testing accuracy & an enormous training time on the CPU. The advantages of the base model are that we use a rigorous transformation object, which has a variety of transformations such as padding & random crops. This results in a generalized model which will have low variance. However, making the architecture better can massively improve training time and still achieve similar or better accuracy - on the trade-off of using a less rigorous transformation object & having a model with high variance.

My architecture [3] is shown in Figure 3. The transformations object I used has only two transformations, namely Normalization from (-1,1) & Random horizontal flip. This makes it easier for the model to learn using the data & should result in lower losses for training.

Then, for the CNN architecture [3], I reduced the number of filters in both conv2d operations, to make the model simpler & less intensive on CPU. The first convolutional layer has 10 output channels using a 5x5 filter size, with stride 1 and padding 0. The next two layers are ReLU & 2x2 Max Pool with stride 2, similar to the base model. Then, there is another convolutional layer with 20 output channels using a 3x3 filter, stride 1 and padding 0. This time, between the ReLU and MaxPool layer - I added a 2D Dropout layer. A dropout layer randomly sets some input neurons to zero in the network, helping solve overfiting and avoiding the model to depend too much on a single neuron in the network. Then, I used a fully connected layer with 50 outputs, followed by a ReLU activation layer and a final fully connected layer with 10 outputs. The number of output channels in all the layers here are significantly lesser than the base model, resulting in a much faster training. Also the base model was missing a dropout layer, which makes it prone to overfitting.

I used CrossEntropyLoss as a loss function, but I changed the optimizer to Stochastic Gradient Descent. I read on a blog that although Adam optimizer converges faster, SGD generalises better and results in improved final performance. The results are shown in figure 4. [3]

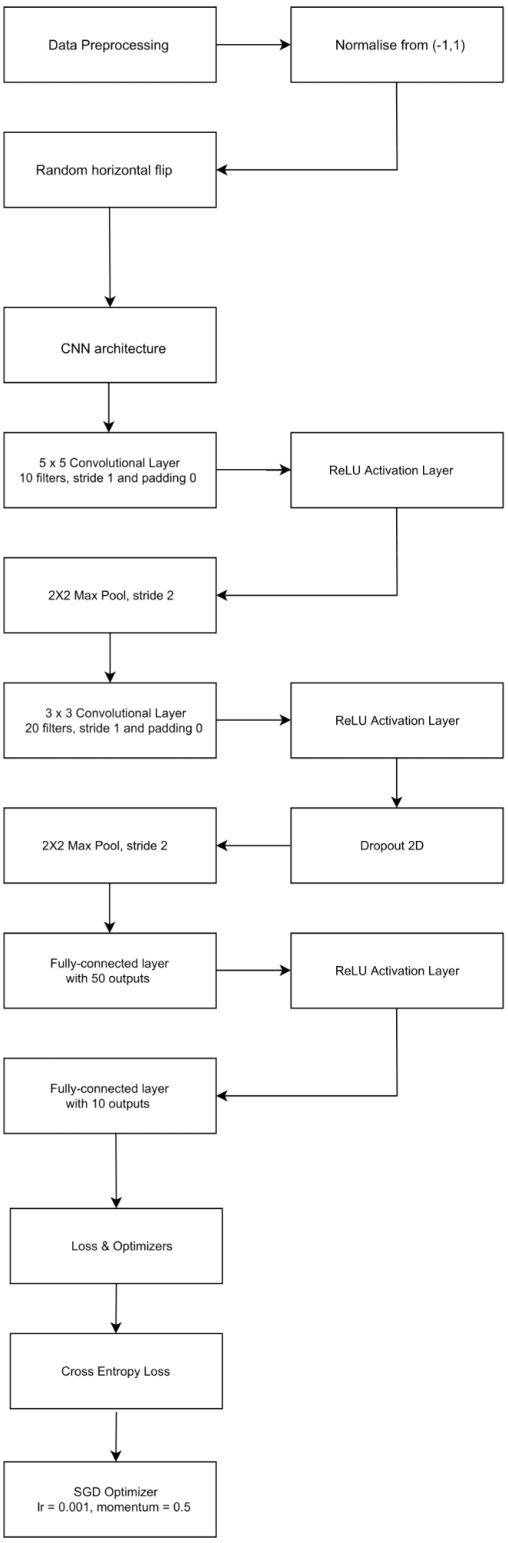

Figure 3 - CNN Architecture for my model [3]

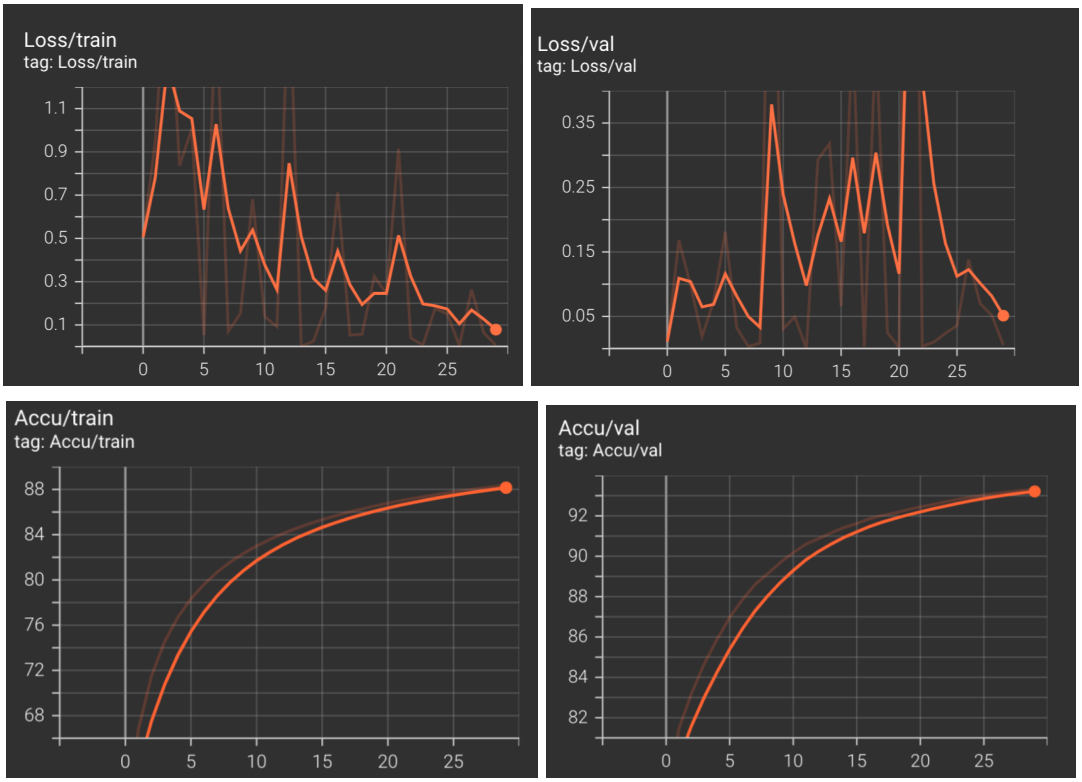

Figure 4 - Training, Validation Loss & Accuracy

Result Analysis

Testing accuracy = 0.75%, Training time for 30 epochs = 30 mins (Apple M1 CPU)

Although the testing accuracy did not improve compared to the base model, there is a significant improvement in the training time. The time taken soared down from 140 minutes to just 30 minutes. Looking at the losses, we can observe that we start with really high loss values for training - however the values decrease at every epoch and we reach a final loss that is much lower compared to the base model. The same goes for validation, although there is a lot of fluctuation and the loss does not decrease at every epoch.

In terms of the accuracy, the base model produces slightly better results than my model, but still - an accuracy of 88% and 93% for training & validation respectively is good enough. My architecture did not majorly improve on all accuracy numbers, but showed a commendable advancement in training time & losses. I would conclude that this architecture is better than the base model, since even though we don't improve on testing accuracy - we would save a lot of time training the model to achieve similar results. On the other hand, the base model is far more generalised & would have lower variance compared to my architecture.

Comparing model to main dataset site [4]
All the models on the main dataset site have a testing accuracy of more than 90%, whereas my model as well as the base model have an accuracy of 75%. Thus, there is a lot of scope of improvement here.

Talking about ResNet18, it may produce better results than standard CNN architecture models since it uses a unique skip connection method to train the model. When we build CNN models, our goal is to learn as many features as possible about the training data set. While increasing the number of layers doesn't always result in a better model, it certainly does improve the model performance to a certain point. Because of the high complexity, training time & resources needed to build extremely deep neural networks, it is quite difficult to keep increasing the number of layers.

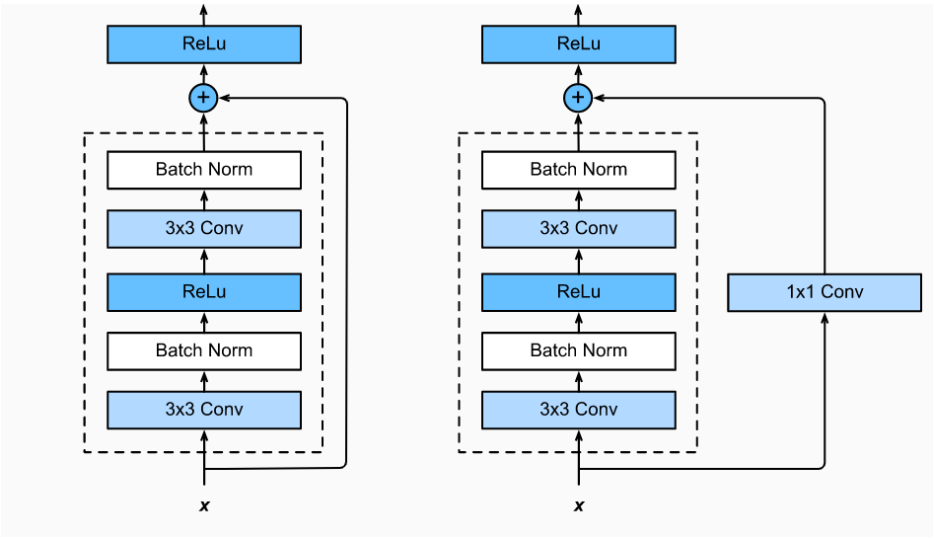

Figure 5 - CNN vs ResNet Architecture [4]

ResNet18 can help solve this issue by introducing the idea of residual blocks (Figure 5). The skipping connection method involves calculating a direct mapping from the input from one residual block to another residual block - significantly reducing the complexity of training deep neural networks & allowing us to keep adding the number of layers inside these blocks. Therefore, the ResNet18 model performs better than the CNN models. [4]

References for Task 2
[1] : https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118
[2] : https://www.geeksforgeeks.org/training-neural-networks-with-validation-using-pytorch/
[3] : https://nextjournal.com/gkoehler/pytorch-mnist
[4] : https://iq.opengenus.org/residual-neural-networks/