Clab-2 Report

ENGN4528

Yixi Rao

U6826541

Task 1 Harris Corner Detector. (5 marks)

1. Read and understand the below corner detection code (Fig. 2).
The block #1 imports the NumPy, block #2 defines a convolution function which takes an image and a filter, and returns the processed image. Block #3 defines a fspecial function which is used to generate a specific Gaussian kernel given a shape and sigma. Block #4 initializes some parameters such as sigma, threshold and derivative masks, and then it computes the x and y derivatives of the image. Block#5 creates a gaussian kernel with sigma = 2, kernel size = 13. Block#6 defines the component of the matrix M such as $Ix^2, Iy^2, Ix * Iy$.

2. Complete the missing parts, rewrite them to 'harris.py' as a python script, and design appropriate function signature (1 mark).
In order to compute the Harris cornerness, I define a Harris_Cornerness function, which uses $Ix^2, Iy^2, Ix * Iy$ to compute the corner response matrix R. The corner response is $R = det\, M - k\,(trace\, M)^2$ where $\det M = Ix^2 * Iy^2 - Ixy^2$ and $Trace\, M = Ix^2 + Iy^2$. Note that Ix^2, Iy^2, Ixy are all matrix, so we can perform matrix operation to compute the R matrix. The R matrix is the same size as original image except the values are corner responses rather than grey scale value.

To perform the non-maximum suppression and thresholding. My thought is to go through all the R value in the R matrix, if the R value is greater than $thresh * R_{max}$ and this specific R value is greater than all the local neighbour's points, then we store the coordinate of that R value. Note that the neighbour's window size is a parameter that we can change.

3. Comment on block #5 (corresponding to line #13 in "harris.m") and every line of your solution in block #7 (0.5 mark) in your report. Specifically, you need to provide short comments on your code, which should make your code readable.
Block#5

```
#! BLOCK #5

# compute a normalised Gaussian kernel g using the fspecial function, where the kernel size is

related to the sigma, namely, kernal_size = 2 * floor(3 * sigma) + 1

g   = fspecial((max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma)
```

Block#7

```
#! BLOCK #7

def Harris_Cornerness(Ix2 : np.array, Iy2 : np.array, Ixy : np.array, k : float = 0.04) -> np.array:

    '''Compute the Harris Cornerness, using the fact:

       1. det(M)   = Ix2 * Iy2 - Ixy ** 2

       2. Trace(M) = (Ix2 + Iy2)

       3. R        = det(m) - (k * (trace(m)) ^ 2)

        Args:

            Ix2 (np.array): Ix**2
```

```
            Iy2 (np.array): Iy**2
            Ixy (np.array): Ix * Iy
            k (float)     : Harris detector free parameter in the equation.

        Returns:
            np.array: R matrix
    '''
    return (Ix2 * Iy2 - Ixy ** 2) - k * ((Ix2 + Iy2) ** 2)
def find_local_max(x : int, y : int, matirx : np.array, size : int = 1) -> tuple:
    '''using a neighbour's window size to find all the adjacent points of (x,y)

        Args:
            x (int): centre point x
            y (int): centre point y
            matrix (np.array): what we perform on
            size (int, optional): neighbour window size. Defaults to 1.

        Returns:
            tuple: this tuple contains three values, this first is the R value, second and third are
the coordinates of the neighbour point
    '''
    xmax, ymax = matirx.shape # boundary of x and y
    neighbours = [] # list of tuples: (R value, X, Y)
    for xi in range(-1 * size, size + 1):
        if x + xi >= 0 and x + xi <= xmax - 1: # all neighbour points should satisfy 0 <= x + xi <=
xmax - 1
            for yi in range(-1 * size, size + 1):
                if y + yi >= 0 and y + yi <= ymax - 1: # all neighbour points should satisfy 0 <= y +
yi <= ymax - 1
                    neighbours.append((matirx[x + xi][y + yi], x + xi, y + yi))

    return max(neighbours) # finding the maximum neighbour point

def detect_Harris_Corner(R : np.array, thresh : float = 0.01, neighbour_size : int = 1) -> np.array:
    '''detect all the Corners by using a thresh and non-maximum suppression

        Args:
            R (np.array): R value matrix
```

```
            thresh (float, optional): R thresh. Defaults to 0.01.
            neighbour_size (int, optional): used to decide the neighbour point's range in NMS.
Defaults to 1.

        Returns:
            np.array: an Nx2 matrix of all the Corners
    '''
    rows, cols    = R.shape
    corner_points = []  # contains all the Corners' coordinates
    R_max         = np.max(R) # maximum Response in R matrix
    for row in range(rows):
        for col in range(cols):
            if R[row][col] >= thresh * R_max: # if this point is a corner then R must be greater than
thresh * R_max
                _, xmax, ymax = find_local_max(row, col, R, neighbour_size) # all the adjacent points
of (row, col)
                if (xmax, ymax) == (row, col): # if (row, col) is the local maxima of its all
adjacent points then add this to the corners list
                    corner_points.append((row, col))
    return np.array(corner_points)


def Harris_Corner(img : np.array, dx : np.array, dy : np.array, k : float = 0.04) -> np.array:
    '''overall function, integrate the compute_M, Harris_Cornerness, detect_Harris_Corner functions
into one function,
       to calculate tbe corners

        Args:
            img (np.array): orignal image
            dx (np.array): Derivative masks
            dy (np.array): Derivative masks

        Returns:
            np.array: an Nx2 matrix of all the Corners
    '''
    # calculate the Iy^2, Ix^2, Ix*Iy
    Iy2, Ix2, Ixy = compute_M(img, dx, dy)
    # calculate the R matrix and collect all the corners
    R       = Harris_Cornerness(Ix2, Iy2, Ixy, k)
```

```
    corners = detect_Harris_Corner(R, thresh, size)

    return corners
```

4. Test this function on the provided four test images (Harris-[1,2,3,4].jpg, they can be downloaded from Wattle). Display your results by marking the detected corners on the input images

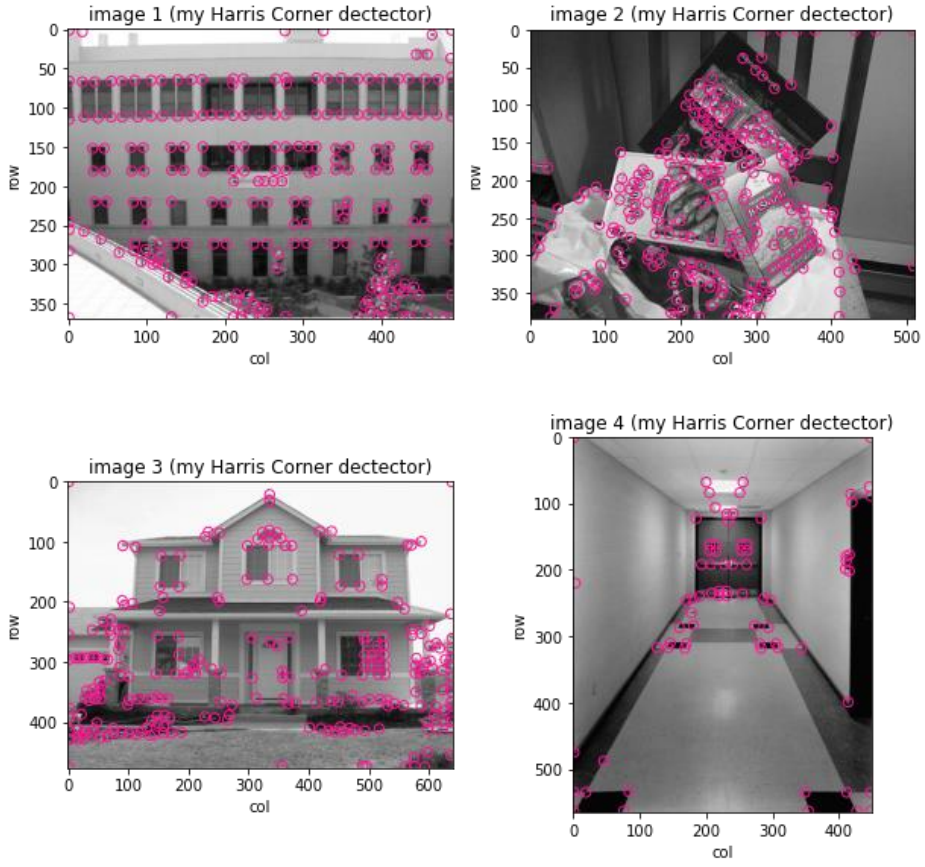

Figure 1.4 using the Harris detector on four images

The figure 1.4 shows the result of after performing the Harris corner detector, we can observe that nearly all the corner points are detected and marked by circle, and we can see that there is nearly no overlap among the different regions of circle. For image 1, we can observe that nearly all the windows are detected except the third row of the window, some corners are missed. For image 2, my detector works well, it only misses some corner on the dark area of the book. For image 3, my detector can detect the corners of left-hand side of the upstairs'' s window but miss the right-hand size corners. I think this is because left-hand side's window has black background whereas the right-hand size's window has white background that is hard to be distinguished with the grey wall. For image 4, we can see that all the corners are detected.

5. Compare your results with that from python's built-in function cv2.cornerHarris() (0.5 mark), and discuss the factors that affect the performance of Harris corner detection (1 mark).

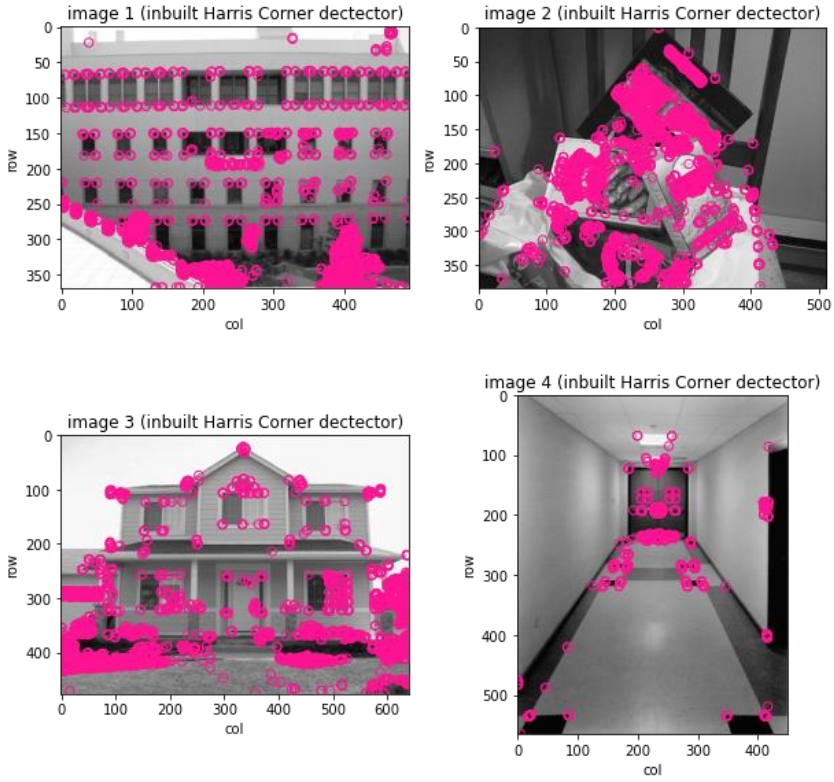

figure 1.5 using the built-in Harris detector on four images

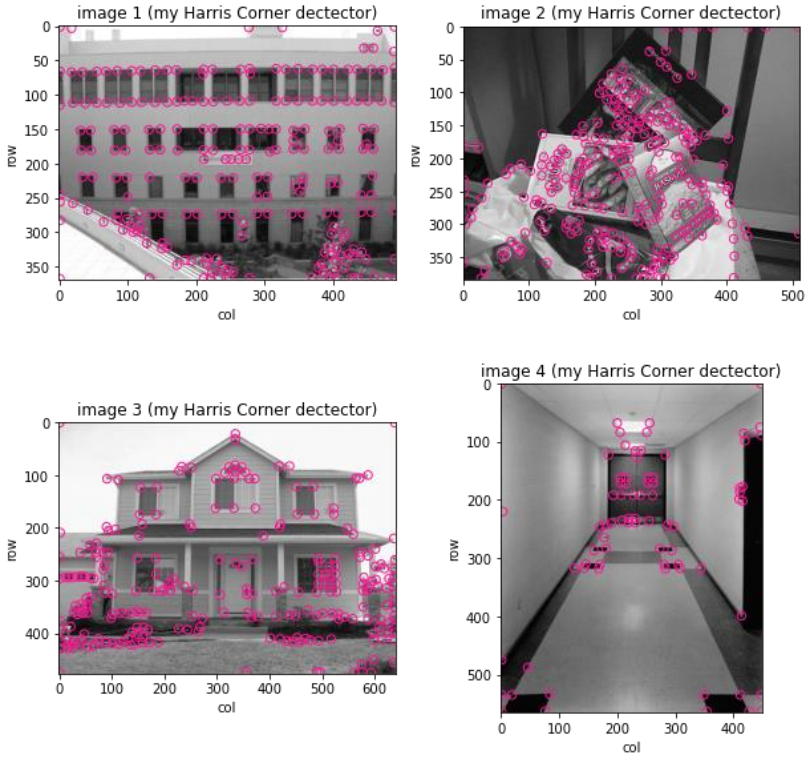

figure 1.6 my harris corner detector (for comparison)

The figure 1.5 shows the result of after performing the built-in Harris corner detector. And figure 1.6 shows the result of after performing my Harris corner detector. From a holistic perspective, we can conclude that built-in Harris corner detector and my Harris corner detector can detect the corners correctly, the only difference is that the result of using built-in Harris corner detector is denser than the result of using my Harris corner detector, this might because the built-in Harris corner detector does not use the non-maximum suppression.

Compare the built-in detector with my detector on image 1, the built-in detector can detect more window's corners than my detector. The reason might be the threshold setting is not very suitable to my detector. For the image 2, the result of using my detector and result of using the built-in detector are nearly identical if we do not consider the overlap. And for image 3, built-in detector can detect all the missed corners mentioned above, and everything else is the same. For image 4, my detector and built-in detector can detect the corners on the ceiling, on the ground tiles, and on the doors.

There are four factors that affect the performance of Harris corner detector, which is the neighbour's window size of the NMS algorithm, the R threshold value, the K value, which is the Harris detector free parameters in the equation, and the kernel of the derivative mask. It is obvious that the neighbour's window size of the NMS algorithm will affect the performance because if we do not use the NMS, there will be a lot of overlap such as one corner will be detected many time and marked many time, however this does not means that bigger window size will perform better, a bigger window size will eliminate some potential corners. The R threshold value and the K value affect the performance in the same way because they affect the R value, for example, setting bigger threshold value will lose some corners, while setting smaller threshold value will misjudge some flat regions as corners. Likewise, bigger K value will lead to bigger R value and vice versa. For the kernel of the derivative mask, this factor will affect performance because the padding value, if we set the padding value as 0, the detector probably marks the corners of the whole image as a corner in the image.

Task 2 - Deep Learning Classification (10 Marks)

1. Download the Kuzushiji-MNIST dataset from google drive

I define a custom torch dataset class to store the Kuzushiji-MNIST dataset. The default torch.utils.data.Dataset's __init__ function requires us to use the path name as inputs (such as annotations_file, img_dir), and then loads the images and labels using the np.load(path) to load the images and labels. I found that this kind of behaviors have a drawback, which is the data iteration will be extremely slow when using dataloader. So, I made a modification to the __init__ input structure, which is using the already loaded images and labels as input rather than loading the image and labels

in the initialization function. This kind of modification will decrease the training time significantly.

2. After loading the data using NumPy, normalize the data to the range between (-1, 1). Also perform the following data augmentation when training: randomly flip the image left and right zero-pad 4 pixels on each side of the input image and randomly crop 28x28 as input to the network.

I am using the Compose function to implement the transform, first I normalize the data using transforms.Normalize(0.5, 0.5), then using transforms.RandomHorizontalFlip(p=0.5), to flip the image left and right randomly, afterward, I use transforms.Pad(4, fill=0, padding_mode='constant') to pad the images on each side, finally, using the transforms.RandomCrop(28) to crop a random location of the image.

3. Build a CNN with the following architecture:

This CNN overall function is $y = fc2(relu(fc1(maxpool(relu(conv2(maxpool(relu(conv1(x))))))))$
We first apply a Convolutional Layer with 5x5 filter (stride 1, padding 2), and the output channels will be 32, so the total number of filters of the first convolution layer will be $32 * (1 * 5 * 5)$. After applying the RELU, the output size of the first convolution layer is $32 * 28 * 28$. This is because

$W_{conv1} = H_{conv1} = \frac{W_{input}\ or\ H_{input} - K + 2P}{s} + 1 = 28$, where K = kernel size, P = padding size, s = stride.
This output will be passed to a 2x2 Max Pooling layer with a stride of 2, which means that the output size will be shrank twice, the output size is $32 * 14 * 14$. The second convolution layer is a Convolutional Layer with 3x3 filter (stride 1, padding 1), and output channels will be 64, so the total number of filters of the second convolution layer will be $64 * (32 * 3 * 3)$. After applying the RELU, the output size of the first convolution layer is $64 * 14 * 14$ because

$W_{conv2} = H_{conv2} = \frac{14 - 3 + 2}{1} + 1 = 14.$

This output will be passed to a 2x2 Max Pooling layer with a stride of 2, which means that the output size will be shrank twice, the output size is $64 * 7 * 7$. To pass the three-dimensional data into the fully-connected layer, we should first flatten the data into one dimension, therefore, the input size of the fully-connected layer will be $64 \cdot 7 \cdot 7 = 3{,}136$ and the output size will be 1024. The next fully-connected layer will take a tensor with size 1024 as input and output a tensor with size 10.

4. Set up cross-entropy loss.

Using LossFunction = nn.CrossEntropyLoss() to set up cross-entropy loss.

5. Set up Adam optimizer, with 1e-3 learning rate and betas= (0.9, 0.999).

Using optimizer = optim.Adam(model.parameters(), betas=(0.9, 0.999), lr=lr) to set up Adam optimizer.

6. Train your model. Draw the following plots

I train my model using the standard process. For each epoch, there is a training loop using the training dataloader to do the forward, backward propagation and optimization. And then do the validation loop to print the statistics.

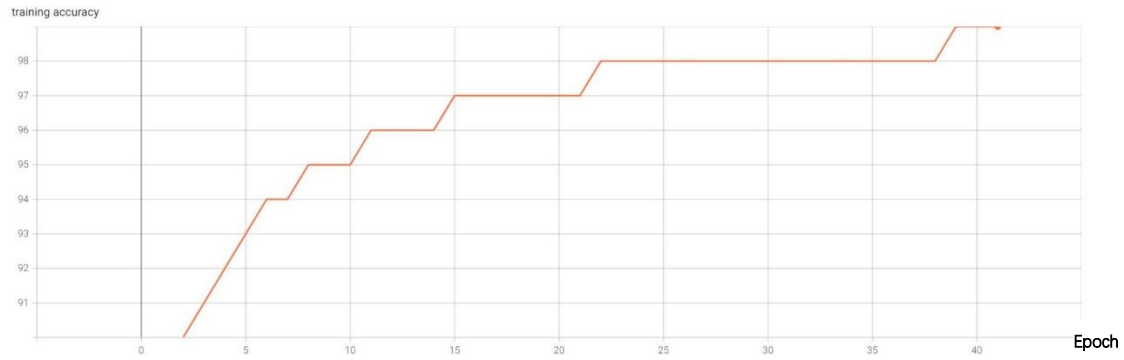

figure 2.1

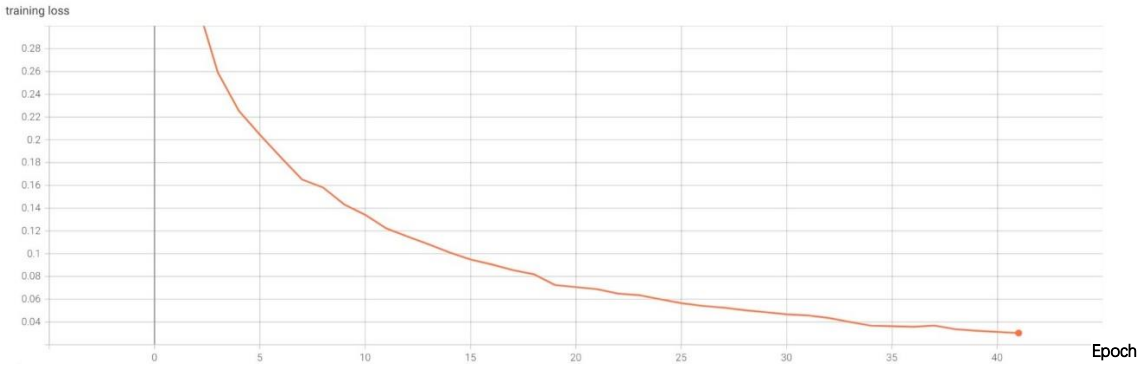

figure 2.2

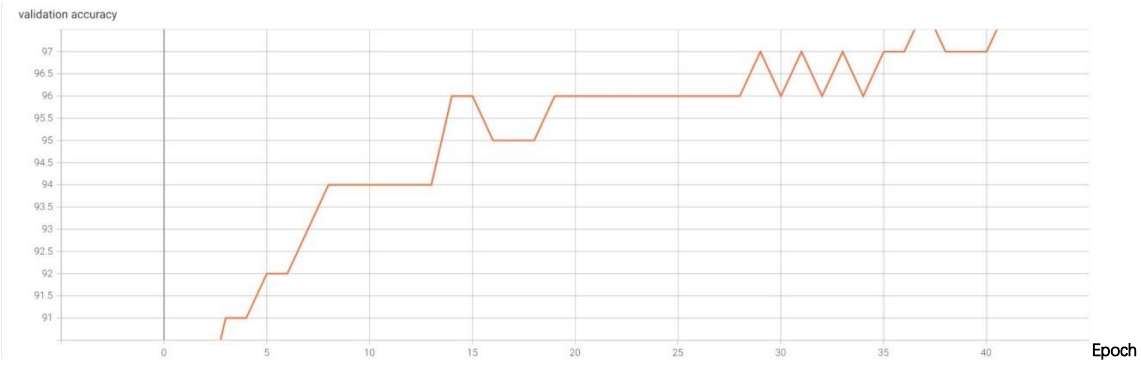

figure 2.3

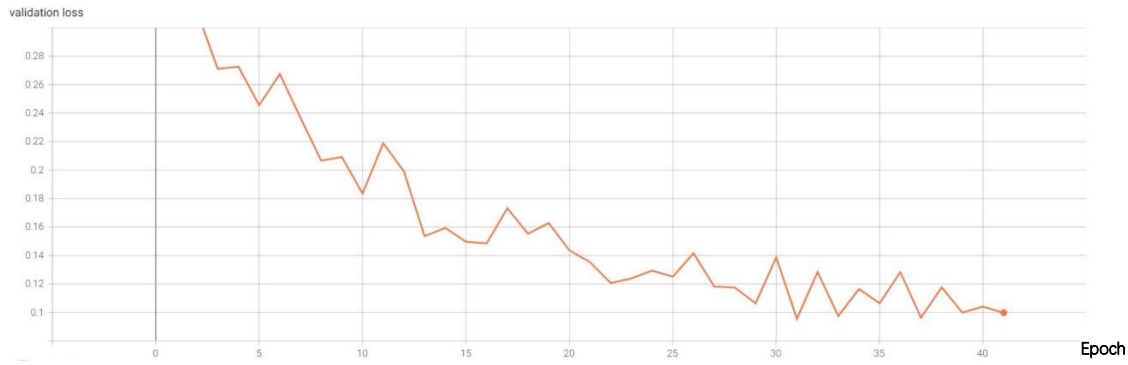

figure 2.4

figure 2.1 shows the training accuracy VS epoch, we can see that the training accuracy remains stable at 98% after epoch 22. Figure 2.2 shows the training loss VS epoch, we can see that the loss decreases tremendously in the first 10 epochs, and then decreases slowly. Figure 2.3 shows the validation accuracy VS epoch, the best accuracy is 97%. Figure 2.4 shows the validation loss VS epoch, the loss fluctuates with the epoch but the loss keeps descending.

7. Train a good model, good design, and your discussion. You need to describe what exactly you did to the base model to improve your results in your report, and your motivation for your approach. Please include plots as above for training and validation loss and accuracy vs. epochs, as well as the final accuracy on the test set.

The methods I use to improve the performance of the CNN are the Early stopping algorithm, learning rate scheduler, and dropout layer. The function of the Early stopping is to prevent the overfitting when training the data, this kind of technique can help us to monitor whether the validation loss will decrease or not, if it detects the validation is increasing that means the model is overfitted, then the training loop will break. However, the loss is fluctuating during training process as we can see in the figure shown above, so, we need to setting a threshold to prevent training stops too early. When the updated model's validation loss is decreased after one epoch, we increase the counter by one, if the counter's value reaches the threshold, we know that it is meaningless to train the model any more, so we need to stop. Another advantage of the early stopping is that we do not need to worry about setting the number of epochs any more, we only need to set up the number of epochs big enough because we know the training process will stop at a specific time, and the model produced is the best model up to this specific epoch.

Learning rate is an important hyperparameter in deep Learning as the Learning rate determines whether the loss function converges to the local minima. A good learning rate can let the loss function converge to the region near the local minima. If we set

the learning rate very small, the convergence process will become very slow, if we set the learning rate too high, the gradient may fluctuate around the local minima's region or even fail to converge. Therefore, it is very important to choose a proper learning rate for the model. The basic idea of Learning rate scheduler is using a high learning rate in the early stage of training, and using small learning rate to fine-turned model in the final stage. The scheduler I selected is the exponential decay, the formula is $lr_{new} = 0.95^{epoch} * lr_{old}$.

Besides, I add a dropout layer after the fully-connected layer, the idea of dropout layer is neurons are temporarily discarded from the network according to certain probability and forces a neuron to work with a random selection of other neurons to achieve good results, which can eliminate and weaken the joint adaptability between neurons and enhanced the generalization ability. So, for a neural network with n neurons, after it has dropout, it can be seen as a collection of 2^n models, and each batch trains a different network because it is discarded randomly. Therefore, dropout layer is a good method to deal with the overfitting and improve the performance of CNN.

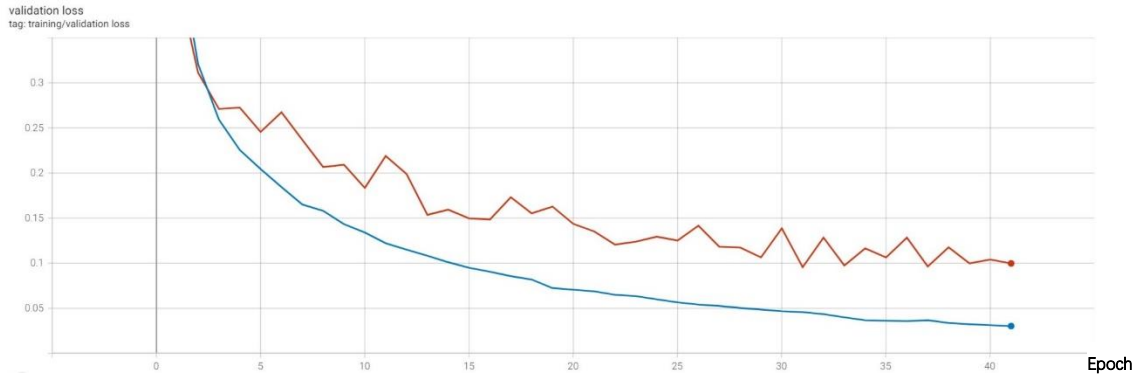

Figure 2.5

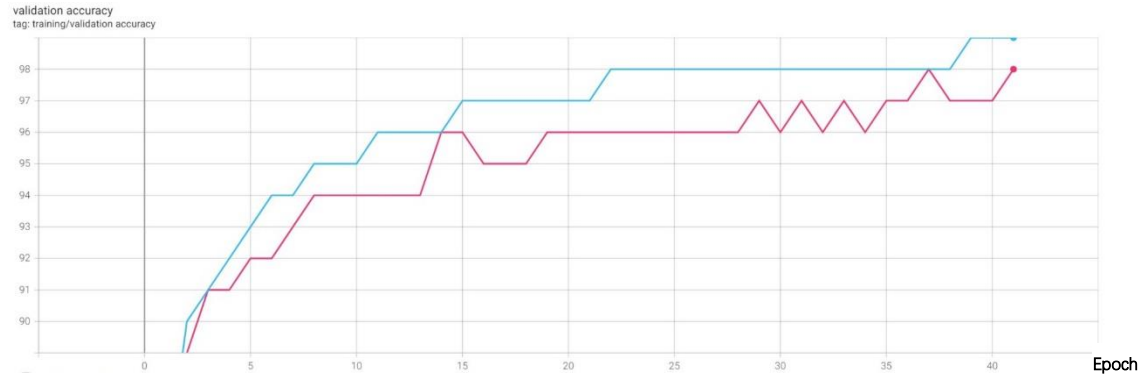

figure 2.6

The figure 2.5 shows the training and validation loss vs. epochs, we can observe that the training and validation loss keep steady after 30 epochs, and if it keep training on more than 41 epochs, the validation loss will increase, so the early stopping algorithm stops at epoch 40. The figure 2.6 shows training and validation accuracy vs. epochs, we can see that the accuracy keeps steady at 25 epochs.

I run the training process multiple time and the final accuracy on the test dataset is in the range of 92%~95%, the average accuracy is 94%.

NOTE: I test my code on the google Colab with GPU acceleration enabled. The training time is in the range of 7min~20min (depends on the GPU Colab allocated to me).

8. The main dataset site on github includes a series of results for other network models (under Benchmark & Results). How does your model compare? Explain why the ResNet18 model may produce better results than yours

My model compares with the Keras Simple CNN Benchmark, Keras Simple CNN's accuracy is 94.63%, which is close to model's accuracy (94%).

My model performs better than 4-Nearest Neighbour Baseline, PCA + 4-kNN, Tuned SVM. The accuracy of these model is 92.10%, 93.98%, 92.82% respectively.

Other models like PreActResNet-18, ResNet18 + VGG, DenseNet-100 (k=12) perform better than my model, they all have the accuracy more than 98%.

[1] ResNet-18 produce better results than my model. This is because it uses the Residual learning, which can help CNN has deeper layers. [3] From experience, we know that the depth of the network is crucial to the performance of the model. The more convolutional layers and pooling layers, the more complete image feature information can be obtained. Therefore, better results can be theoretically obtained when the model is deeper. However, in the actual experiment, it is found that with the network depth increases, network accuracy keeps stable, and even decline. This is because two problems arise.

- Vanishing and Exploding Gradients
- Degradation problem

So, we use the Residual Learning to deal with the problems. [2] When the input is x, the features learned are $H(x)$, the ResNet Function will be $F(x) = H(x) - x$. Now, we are learning $H(x) = F(x) + x$. We found that updating the weight to keep a high testing accuracy will be very difficult when we are in a very deep network layer, so we want the last layer's output x that is the current the best solution does not change to much (for example $H(x) = x$) so we only need to update $F(x)$'s weight a little bit. Then, we can use the ResNet to build deeper network to better performance.

Reference:

[1] https://cv-tricks.com/keras/understand-implement-resnets/
[2] https://blog.csdn.net/sunny_yeah_/article/details/89430124?spm=1001.2101.3001.666

1.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc_relevant_paycolumn_v3&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc_relevant_paycolumn_v3&utm_relevant_index=1

[3] Facial expression recognition via ResNet-50 BinLi, DimasLima