Computer Vision - 16720
Homework 5

[redacted]
Andrew id:[redacted]

Start date: Mar 31st
Due date: Apr 18th

Overview

Q1.1.1:

The rectified linear units (ReLU) activation function is:

$$\phi(x) = max(0, x)$$

Reason:
1. Sigmoid function contains an exp function which lead to more computation while ReLU function does not.
2. ReLU would cause the output of some units to be zero, such that guarantee the sparsity of the network as well as diminish the correlation among parameters which in turn diminish the possibility of over-fitting the training dataset.
3. The gradient of sigmoid function are close to zero at the area near saturation which will cause the "disappear" of gradient and lose of information in back propagation. Use ReLU we got either 0 or 1 for gradient.

Q1.1.2:

If we are using linear activation function, we could assume the function is $G(x) = x$ in which there's no change of dimension. Since the input x has dimension C, the first hidden layer pre-activation $a^{(1)}(x)$ is given by:

$$a^{(1)}(x) = W^{(1)}x + b^{(1)}$$

Since the activation function is $G(x) = x$, so the post-activation values of the first hidden layer is:

$$h^{(1)}(x) = G(a^{(1)}(x)) = G(W^{(1)}x + b^{(1)}) = W^{(1)}x + b^{(1)}$$

We can see this output is a linear combination of the input. And for each hidden layer, the post-activation value are :

$$h^{(t)}(x) = G(a^{(t)}(x)) = W^{(t)}h^{(t-1)} + b^{(t)}$$

So the output of each hidden layer are just the linear combination of the input, so no matter how many hidden layers there are, the output are just gonna be the linear combination of the original input which has dimension of C. In this case, the hidden layer has no effect of the representation capability of the network, the network becomes a "big" perceptron, which output is just a linear combination of input.

2 Implement a Fully Connected Network

2.1 Network Initialization

Q2.1.1:

Ans: If we initialize a neural network's weight with all zeros then the network will not be able to learn valuable information from training. If all the weight are zeros, or some constants, then all hidden units in this network are computing the same output, and in back propagation will yield same gradients and furthermore do the same updates to parameters. In this way the network is symmetric.
The weight cannot be initialized to be either too big or too small. Consider the plot of sigmoid function, its curve is nearly flat with too big or too small input, and in both case the gradient will be close to zero in back propagation, which is bad for our network learning. However it's ok to initialize bias as all zeros since with randomly initialize the weights we already broke the symmetry.

Q2.1.2:

Please see the code submitted on Blackboard.

Q2.1.3:

First I randomly initialize the weight in Gaussian distribution with zero mean and sigma equal to 1. Since we cannot initialize the weight with zeros as discussed before, but if the value of weight are too big the signal will be amplify during passing through each layer and finally caused the network's learning failed. Then I tried Xavier initialization. The intuition behind it is to make the variance remain same with passing each layer, so that our signal will not likely to expand to a very large value or diminish to a too small value, which both lead to the failure of learning. In Xavier initialization, it's actually another Gaussian distribution with zero mean but with variance $\frac{1}{N}$ where N is the number of input neurons.

2.2 Forward Propagation

Q2.2.1:

Please see the code submitted on Blackboard.

Q2.2.2:

Please see the code submitted on Blackboard.

Q2.2.3:

Please see the code submitted on Blackboard.

2.3 Backwards Propagation

Q2.3.1:

Please see the code submitted on Blackboard.

Q2.3.2:

Please see the code submitted on Blackboard.

2.4 Training Loop

Q2.4.1:

For gradient descent(GD), each update go through an epoch while for stochastic gradient descent(SGD), each update go through $\frac{1}{D}$ epoch where D is the number of samples. In other words, GD followed the optimal direction of descending gradient while SGD does not. In GD, one epoch equals to one iteration and in SGD, D iteration equals to a epoch. So the GD is more faster than SGD in terms of number of iterations since it always pick the direction that descends the most but requires a lot of computation when dealing with large dataset. GD goes firmly and slowly to the optical solution.
However, SGD is faster in terms of number of epochs because in SGD we calculate one training example per update so after D iteration is an epoch and in GD we calculate D training examples per update so each iteration is an epoch. SGD may get close to optical solution fast but the closer it get the slower the converge will be, and if the learning rate has not been chose wisely it may oscillating around the optical solution.

Q2.4.2:

Please see the code submitted on Blackboard.

Gradient Checker

Q2.5.1:

Please see the code submitted on Blackboard.

Training Models

3.1: From Scratch

Q3.1.1:

I set the number of epoch to be 40 and trained the model. Since I randomly shuffled the data and the parameter were randomly generated as well, the start point of the accuracy and loss might be different each time. However, it will always converge. I also trained the model for 240 epochs with different learning rate to see the difference.

Q3.1.2:

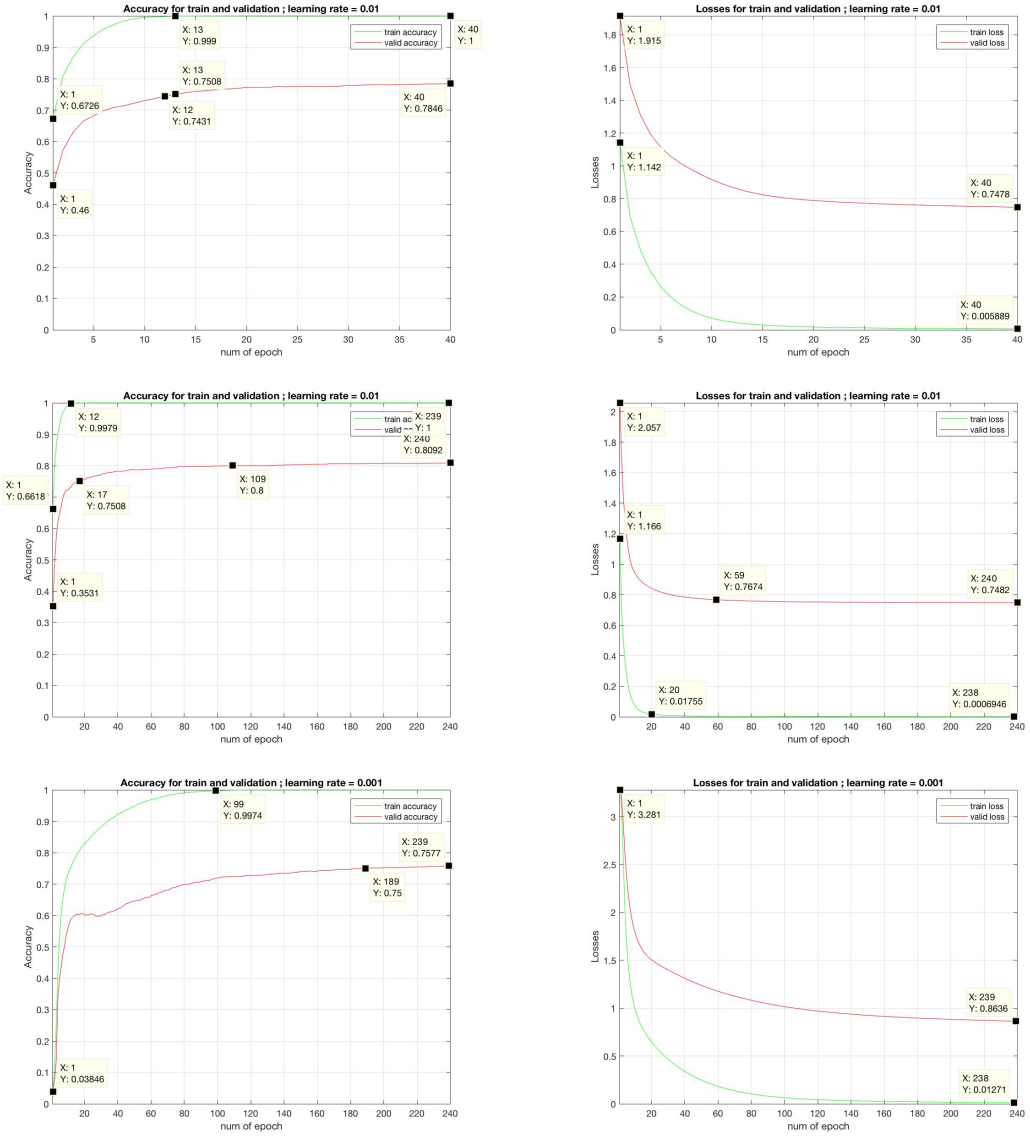


Figure 1: Accuracies and losses for different learning rate

From the graphs above we can see with learning rate equals 0.01 the accuracy increases faster than learning rate equals 0.001, and the loss decreases faster as well. As shown in the figure, it takes much longer the time for 0.001 learning rate to achieve 75% accuracy on validation set. In gradient descent, learning rate affects how much difference the gradient cause on the updated parameters. The bigger the learning rate is, the faster the model learned. But in practice we cannot set the learning rate to be arbitrarily big, since if it is too big we might just "skip" the optimal point and start hovering around it. And if the learning rate is too small it might take a long time to converge.

The accuracy and loss for learning rate equals to 0.01 on test set are: $Acr_{0.01} = 78.46\%$, and $Loss_{0.01} = 0.7478$ and it was trained for 40 epochs. The accuracy and loss for learning rate equals to 0.001 on test set are: $Acr_{0.001} = 75.77\%$, and $Loss_{0.001} = 0.8636$ and it was trained for 240 epochs. In this case, although the 0.001 learning rate was less likely to stuck in local minimum, it caused too much computation. And the model with 0.01 learning rate achieved a higher validation accuracy with less number of epochs. So the model with learning rate equals to 0.01 is better.

So the final accuracy of the best network on the test set is : $Acr_{0.01} = 79.23\%$ (40 epochs training) and $Acr_{0.01} = 81.15\%$ (240 epochs training).

Q3.1.3:

The best network from the previous question is with learning rate equals to 0.01 (240 epochs), and the accuracy and cross-entropy loss on the test set are: $Acr_{0.01} = 81.15\%$, and $Loss_{0.01} = 0.6851$. Below are two graphs about visualizing the weight matrices. I did normalize the matrix to make the pattern clearer. From the graph on the left side we could see that the model has learned patterns from training samples since each hidden units has formed up some specific pattern. And the initial parameters are just randomly distributed and contained no pattern inside.

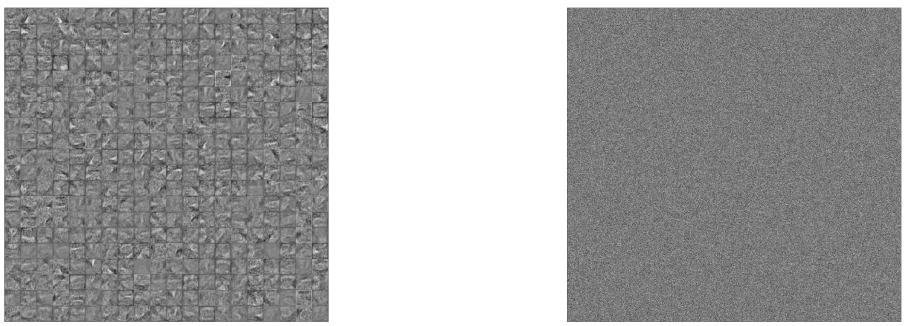
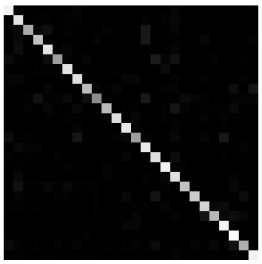
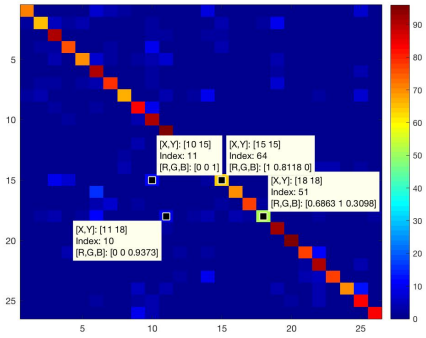


Figure 2: Visualization of two weight matrices

Q3.1.4:

To visualize the confusion matrix, I used 3 ways. The first image was the output of simply multiply predicted labels and real label matrix, and after normalization and upscaling the output was a 260*260 matrix with large value diagonally and small values otherwise. From this image we could see the grayer a pixel is, the more samples were wrong classified on this class. To have a more specific visualization, I used MATLAB function confusion to plot the confusion matrix and with a colorbar to indicate the relative value of pixels.
From this and the third image we could see that the class number of top two pairs of classes that are most commonly confused are 15 and 18 which were indicated on the image with data cursor.
The 15th class was mostly misclassified as the 10th class, i.e, O was mostly misclassified as J. The 18th class was mostly misclassified as the 11th class, i.e, R was mostly misclassified as K. It is reasonable since they do share similar patterns so after the network's learning they might be classified as the other one . Like for O and J, they both have vertical lines and curves. For R and K, they shared diagonal lines and vertical lines and they do looks similar to each other with human eyes.
I also tried using MATLAB function plotconfusion which gives a more detailed output of number and percentage wrongly and correctly classified for each class.





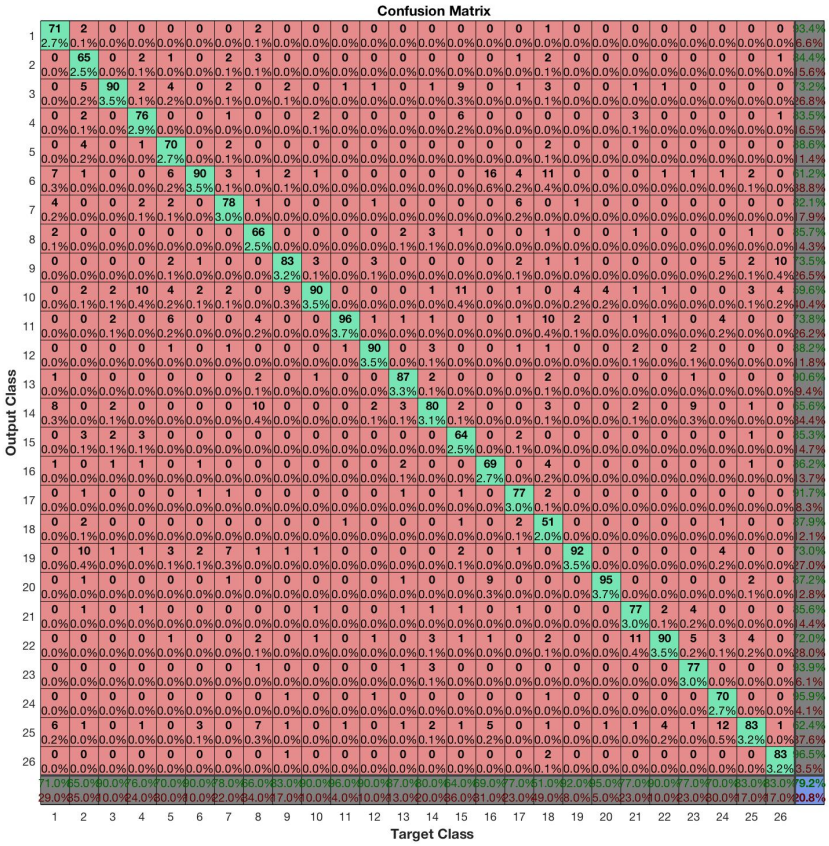


Figure 4: Confusion matrix visualization

3.2: Fine Tuning

Q3.2.1:

In this section I used the trained parameters for 26 letters in $nist26_model_60iters.mat$ for the first layer and randomly initialized the parameter for the second layer. The reason for doing this is that most real images share common properties, so with using parameters from other trained model we could avoid doing repeat work for those common properties. The model was trained for 5 epochs with learning rate equal to 0.01. Below are graphs for accuracy and loss. From the images below we could see that the start point for both accuracies are higher, indicating the pre-trained shared features were used in our model and worked well. With the same learning rate, this model achieve 94.57% accuracy on training set and 72.17% accuracy on validation set after 5 epochs.

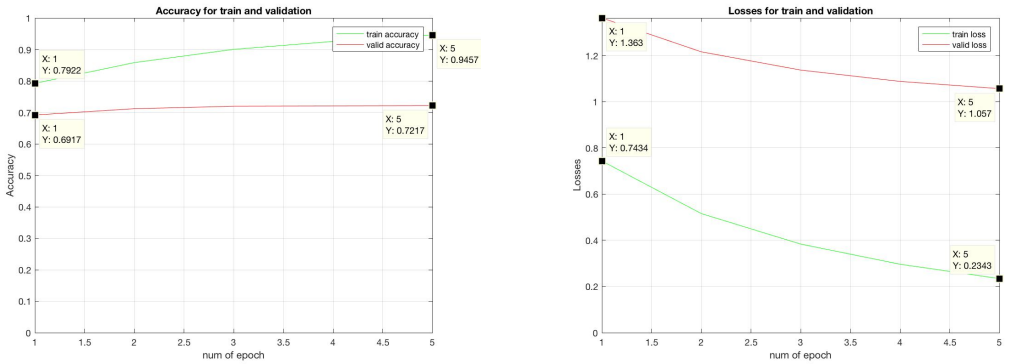
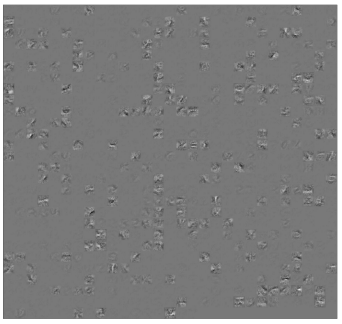


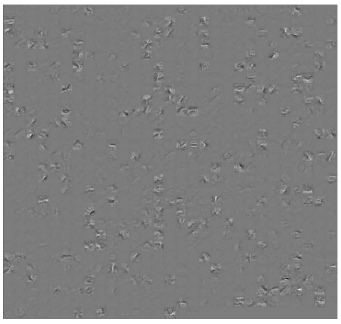
Figure 5: Accuracies and losses for fine tuning model

Q3.2.2: Visualize weight matrix

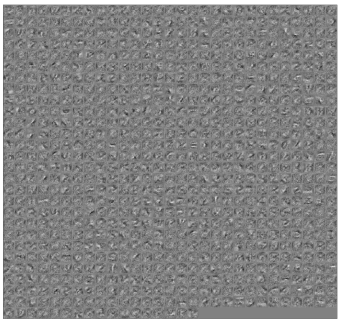
Image on the left side are the visualization right after initialization and the lower one is the output after normalization. Image on the right side are visualization of the trained model's weight matrix and again the lower one is the output after normalization. From the images we could see there are only some slight different between two images, the clear dominate patterns are the patterns they shared and after training the model learned some new pattern on this basis.
The Accuracy and loss on test set are: $Acr = 68.75\%$ and $Loss = 1.1211$.



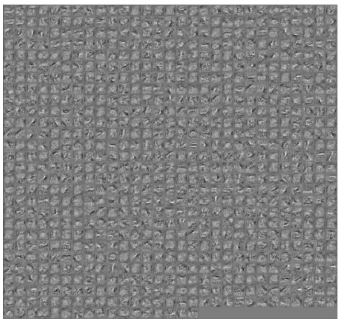
(a) Initialization, no normalization



(b) Trained, no normalization



(a) Initialization, with normalization



(b) Trained, with normalization

Figure 7: Accuracies and losses for fine tuning model

Q3.2.3: Visualize confusion matrix

For this part , since the output of MATLAB function confusion was the most vivid and the output of plotconfusion was too big, I just show the output of confusion. Again, with using imagesc and colorbar. From the image below we could see the class number of top two pairs of classes that are most commonly confused were 28 and 4. For class 28(number 1), it is mostly misclassified as class 9(letter I) and class 4 (letter D) is mostly misclassified as class 15 (letter O). It's different from the output before and again it is reasonable because they share similar patterns and they look alike in bare eyes. From the image of confusion matrix we could see with introducing numbers the behavior of letters' recognition become worse under its effect (more letters were misclassified can be seen from the color change). Since with introducing more labels, more possibility of making mistakes was added.

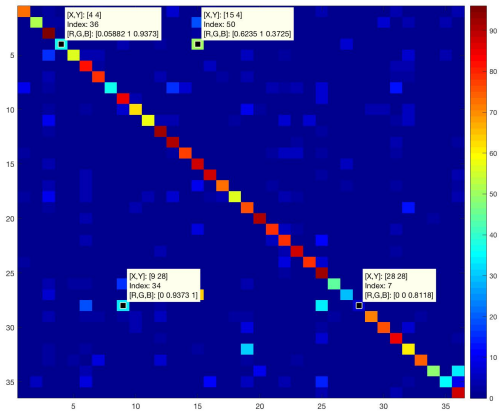


Figure 8: Confusion matrix visualization

4: Extract Text from Images

Q4.1:

First, since this method based on the extraction of connected area, so if two letters were connected this method won't be able to classify them separately.
Second, this method assume all texts were written horizontally and formed in lines. So this method is not rotation invariant.
Third, before text recognition we did morphology processing to the input image to remove noise. In this way if the text are not of the same size, say some of them are too small, the might be eliminate as noise.
In my code, I tested these two images below which I expected to failed in extracting the text. The first image I made was intentionally to make connections between letters and the second one was used to test rotation invariant.

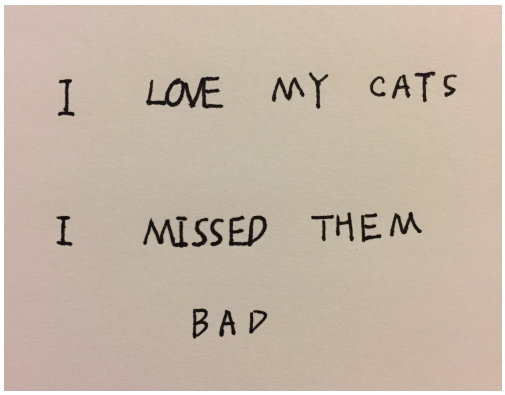


Figure 9: Test for connection between images

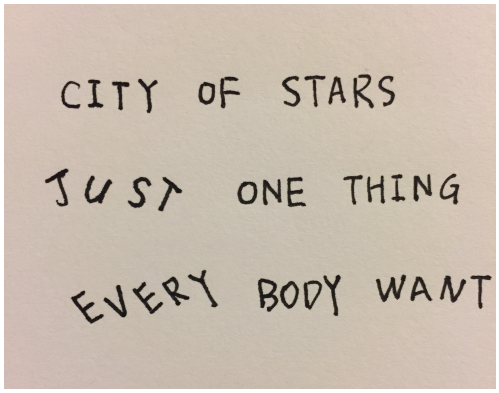


Figure 10: Test for letter rotation

Q4.2:

Please see the code submitted on Blackboard.

Q4.3:

Below are images with located boxes on top of the image. For four provided images, the accuracy was 100% since there's no connection between letters or numbers. For my test image testcats, this method is not able to put an boundingbox for each letter separately since there were connections between letters. For the test image star, letters were correctly detected and bounded as well.

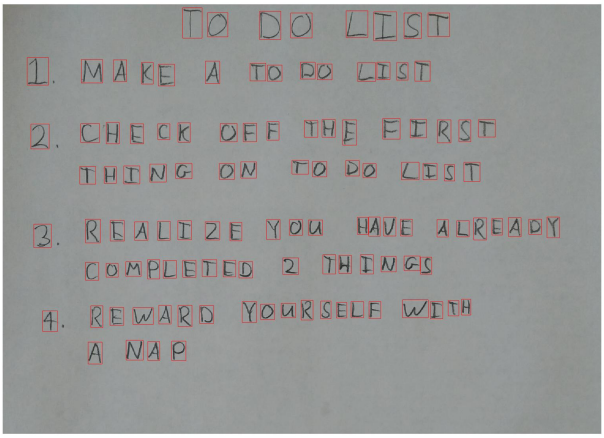


Figure 11: Located boxes for list image

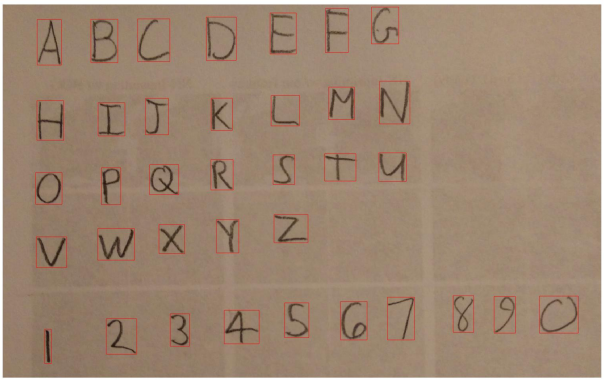


Figure 12: Located boxes for letters image

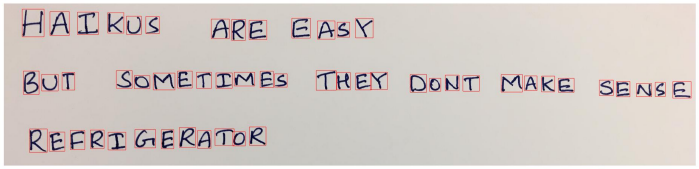


Figure 13: Located boxes for haiku image

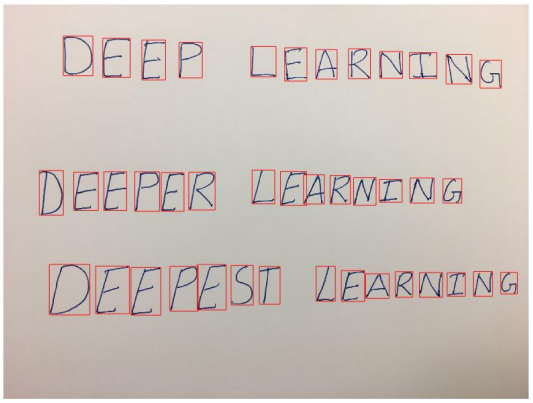


Figure 14: Located boxes for deep image

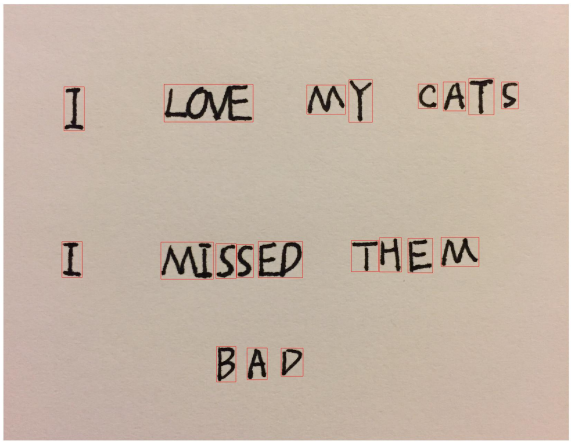


Figure 15: Located boxes for testcats image

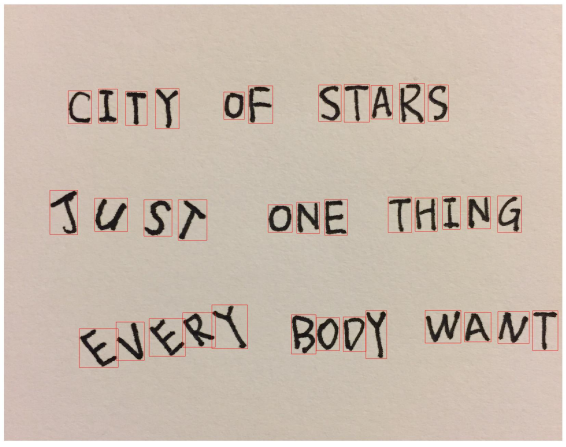


Figure 16: Located boxes for star image

Q4.4:

Please see the code submitted on Blackboard.

Q4.5:

In computing the accuracy I did not include the two images I created since they were created for the case this algorithm may failed. And the accuracy was computed without space and enter. Below are the extracted text and the accuracy. The overall accuracy over these four images is: $187/246 \approx 0.76$.
From the extracted text we could see most letters and numbers were correctly classified.

```
--------------------------------------------------
Extracted Text from image 1

FO OO LIST
I MAXE A TO2Q LI5T
2 LH6CK OFFTHE FIRST
THING ON TODO LIST
3 X6ALI2E YOU HAUE ALR6ADT
EOMPLETED 2 THINGS
4 REWARD YOURSELF WITB
A NAP

Accuracy for this test image is: 0.852174
--------------------------------------------------

--------------------------------------------------
Extracted Text from image 2

IBLDEFG
HIIKLMN
QPQKSTU
VWXYZ
IL34S67YYS

Accuracy for this test image is: 0.694444
--------------------------------------------------
```

```
--------------------------------------------------
Extracted Text from image 3

HAIKU5 A8E EA5Y
BLT SQMETIMES TREY OOWT MAKG SEMGE
LEGRIGERATOR

Accuracy for this test image is: 0.759259
--------------------------------------------------

--------------------------------------------------
Extracted Text from image 4

J4EY LEAKMING
OEFYFK L5AKNING
O FCFFSI LEARNING

Accuracy for this test image is: 0.560976
--------------------------------------------------
```

Extra Credit

Q 5.1: Optimization Algorithms

I implement the Adagrad method to automatically update a learning rate. This method may have a faster converge but since the learning rate keep shrinking in each update it may finally be too small and cause the network stop learning.
From the image below we can see training accuracy increase to nearly 90% fast but validation accuracy keep oscillating below 10%. It seems that the model has over fitted the training dataset.

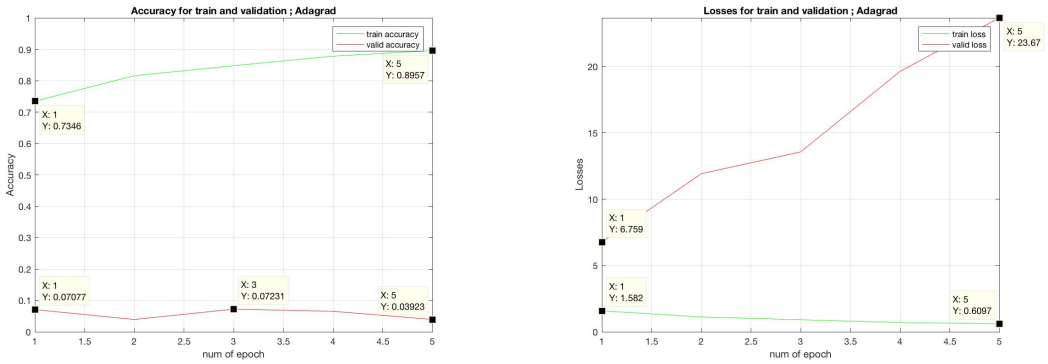


Figure 17: Accuracy and loss with Adagrad

Q 5.2: Convolutional Neural Network

I used MATLAB deep learning tool box to train this model. I used a 10 layer network. The structure of the network is: Input image(32*32*1)→ Convolutional layer(5*5 kernel, 20 in a group)→ ReLU layer → Max pooling layer(2*2 window, stride = 2) → Convolutional layer(5*5 kernel, 40 in a group)→ ReLU layer → Max pooling layer(2*2 window, stride = 2)→ Fully connected layer(n = 26) → softmax layer → Classification layer. The accuracy curve with respect to each iteration was plotted during training, the lost and accuracy curve with respect to each epoch was plotted after training.

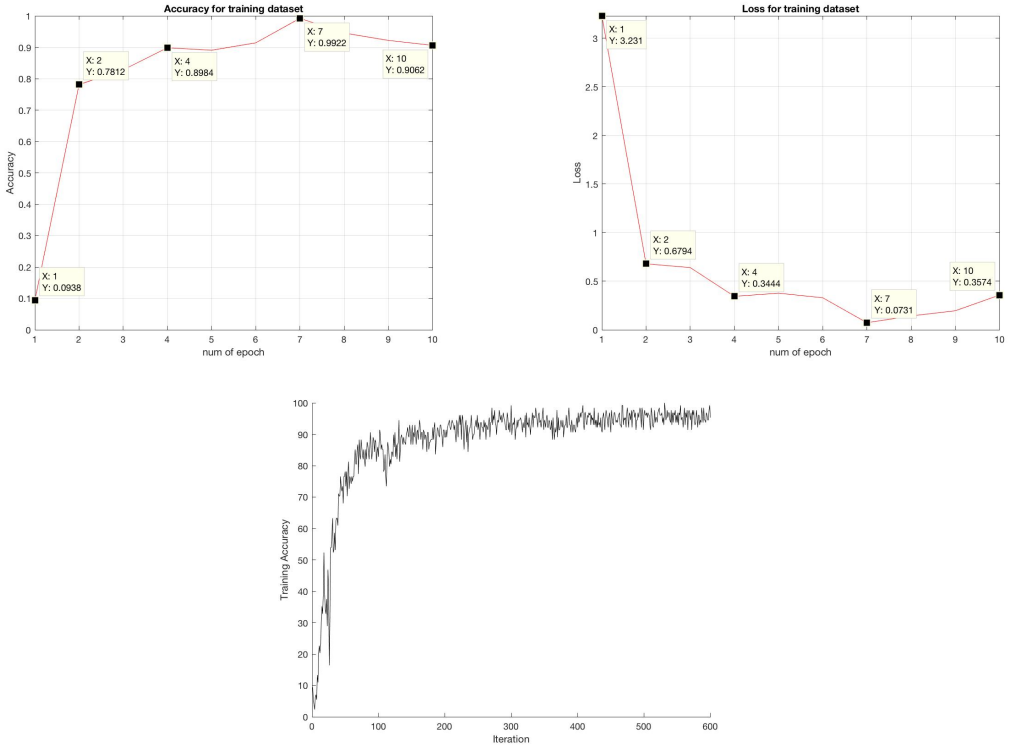


Figure 18: Accuracy and loss with CNN

Q 5.3: Better OCR

Q 5.4: Dropout

In this section I re-implement Forward and Backward function as well as the Train function. In forward with 0.5 dropout means we randomly let 50% hidden units not working by setting their output to zero. To maintain the same "sum" of the output I rescaled the left output by multiply them with 2. And in back propagation those hidden units who are not working we do not change their gradient. Theoretically by doing this we may get a lower accuracy of training dataset but a higher accuracy on validation dataset and test dataset, because we could prevent our model from over fitting the training dataset by randomly "ignore" half hidden units.
Below are figure of accuracy and loss after training for 40 epoch. From the image pf accuracy we could see the training accuracy never get to 1 in 40 epoch, but without dropout at the 14th epoch the training accuracy is 1. However, for the validation accuracy, without dropout the final accuracy after 40 epoch is 78.46% and with dropout the final accuracy is 83.54% which is relatively higher. It turned out we fit the training data less well but got a more general model with dropout, which is exactly the type of model we need.
For test dataset, like mentioned in the former section, without dropout, the test accuracy with parameters trained for 40 epochs are: $Acr_{former} = 79.23\%$ and with dropout the test accuracy with parameters trained for 40 epochs are: $Acr_{dropout} = 83.81\%$. The model without dropout didn't get such test accuracy even with 240 epochs' training. From these results we could see we got a more general model with method of dropout.

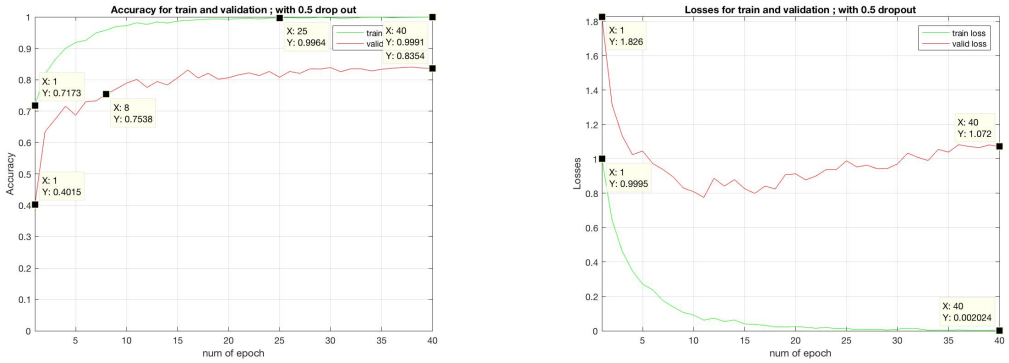


Figure 19: Accuracy and loss with dropout

Q 5.5: Count the number of parameters

For VGG Network it proved with deeper network we could improve the performance of network. In VGG network, it used small convolutional kernel(3*3) so that the number of parameters of one kernel is 9, and with larger kernel requires more parameters. For VGG-16 network, its depth is 16 and as shown in the paper its number of parameters are about 138 millions. There are 3 fully connected layers with output dimension of 4096 (64*64), 4096 and 1000. After all convolution layer and maxpooling layer, the output dimension for each input is 7*7, so the percentage of parameters of fully connected layers are:

$$(7 * 7 * 4096 + 4096 * 4096 + 4096 * 1000)./138million = 15.3\%$$

For GoogleNet its depth is 22 which is relatively deeper than VGG-16 Network but has less parameters. Because in Googlenet 1*1 convolution kernel and 3*3 convolution kernel were used, and 1*1 kernel reduced the number of parameters. The using of 1*1 convolution kernel called bottleneck. It is used to diminish the number of features input to next layer. The reason for doing this is that there are correlation between input features, so we could eliminate the redundancy by passing them through a 1*1 convolution kernel to deliver less feature th next layer.
Besides, googlenet used Inception, which used sparse connection between layers instead of fully connection. In this way they could diminish the computation and the number of parameters, since the fully connected layers requires large amount of parameters.