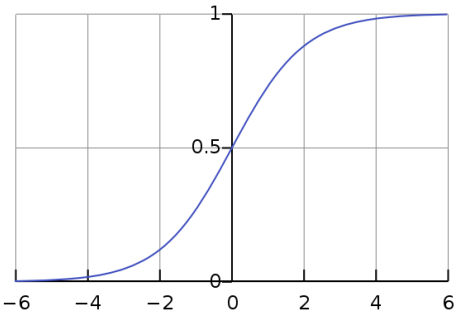
16720 HW5

Q1.1.1

In my own opinion, there are several reasons that make people prefer Relu more than other activation function such as tanh or sigmoid.

1. The calculation of Relu is much more simple than sigmoid, it just require to judge the input is bigger than 0 or not. On the contrary, no matter what value the input has, it has to so some exponential calculations, the difference will not be obvious when the number of the neurons is small, but in reality, the number of both layers and the neurons are really big, under this situation, the difference will be absolutely obvious.
2. The Relu function is safer and simple when we do the backpropagation, the derivative of it can only be 0 or 1, it looks pretty simple, but people has proved that it is also useful. As for sigmoid function, the figure of Relu looks like that:



It can be regarded as linear when the value of input is closed to 0, but when the magnitude of the input becomes bigger, its slope can be more and more flat, which means when we do the backpropagation, the derivative of it can be closed to 0, and we know that this situation can easily cause gradient vanishing.

3. There are many 0 in the output of Relu function, and it can cause the sparsity of the output function, this can not only simplify the calculation, but can also, in another way, reduce the complexity of the neural network thus effectively prevent the overfitting.

Q1.1.2

If all the activation function is linear, there will be a problem that the depth of the network makes no sense. The reason for this is that all the hidden layers can be transported to just one layer, all the weights and bias matrix can be multiplied to a total matrix and can totally represent the whole network, which means there will be no difference between the depth of the neural network.

Q2.1.1

In my opinion, here are some reasons why we shouldn't initialize all the weights as 0 or some other same constants:

1. In forward propagation, we know that the rules for each hidden layer should be:

$$\begin{cases} z_n = W_n a_{n-1} + b_n \\ a_n = g\left(z_n\right) \end{cases}$$

Where a_{n-1} is the output of last layer, especially, for the first hidden layer, the a_{n-1} should be

the output of input layer. $g(\ )$ is the activations function. Imagine if we initialize all of the weights as 0 or other same constants, the result is that all the components in W_n are the same, thus from the beginning of hidden layer, all of the linear output z_n in the same layer will be the same, thus the activated output will also be the same. That means we just input just one input sample into the neural network and train it, and of course the final result is that we cannot find the global optimization for the training samples.

2. In the backpropagation, the problem is just the same. In our homework, we use sigmoid function as our activation function, and the derivative for it is like this:

$$\frac{\partial a_n}{\partial z_n} = a_n(1-a_n)$$

If the activation output a_n are the same in each layer, the derivative of them would also be the same, so the conclusion is that, if we initialize all the weights as 0 or some other same constants, we can hardly improve the neural network.

Q2.1.3

Here are some methods I took to initialize my network:

1. I use the function randn() to initialize the weights and bias, because in the beginning we don't have any information about the training samples, it's a good idea to make the weights object to Gaussian distribution.
2. And I also do some normalization in the initialization step. I tried to normalize each rows of the weights, this can constraint the magnitude of the weights, because in the beginning I didn't use this method, and during the training, the linear output of some layers become bigger and bigger, and considering that we use sigmoid as activation function, so all of the activation output become 1 in the end, and that will cause the same problem as we initialize all the components in weights as the same constants. In order to prevent this from happening, we should constraint the initializing magnitude of the weights.
3. And I also so some normalization for the inputs, I make all the features of inputs object to Gaussian distribution. Actually, I did this in $train26.m$ but not in the initialization function.

Q2.4.1

The batch gradient descent(BGD) means we use all of the training samples at the same time, calculates the average gradients, and then update the weights and bias. And the stochastic gradient descent(SGD) means we should use one sample every time to calculate the gradients and update the weights and bias. So we can easily find out the different cons and pros of these two updating methods.

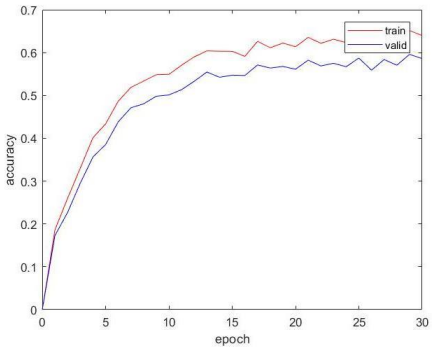
As for batch gradient descent, it can find the global optimal solution and effectively avoid local trapped in local optimal solution. But this method requires to store and calculate all of the

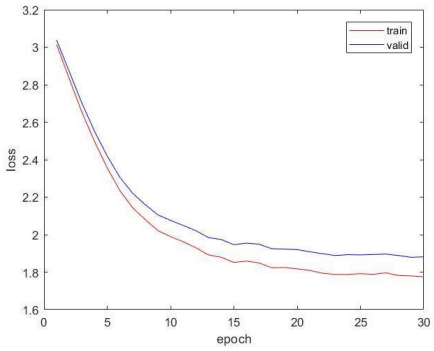
samples at one time, which means it requires more storage and more calculating power. On the contrary, stochastic gradient descent only take in one sample and update the weights and bias at one time, which means in every step, it requires less storage and computation power, but the disadvantage is also obvious, through the definition of it we can easily point out that what this method do is to find the local optimal solution, and it imagine in some high dimensional situation, the local optimal solution will have a high probability to be the global optimal solution, so in this method, there will be more noises and the path through which to find the final solution is more complicated.

So we can easily make the conclusion that in terms of epoch, which means go through the whole training sample set, the BGD is more faster cause it just calculate the whole gradients and update the weights for one time, but in terms of iteration, the SGD will be much faster, the reason is that it only operate one sample for one time, but the BGD has to deal with the whole training sample set, for BGD, one epoch just has the same meaning of one iteration.

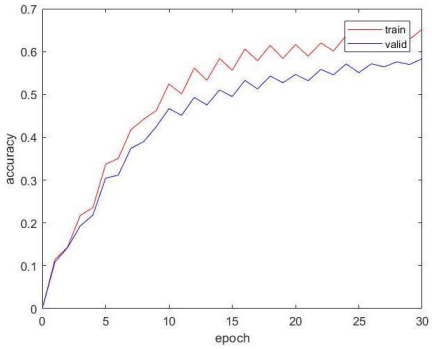
Q3.1.2

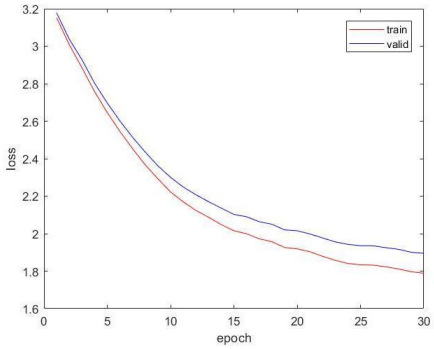
Here are the accucracy and loss when learning rate equals 0.01:





And here are the same figures when learning rate equals 0.001:

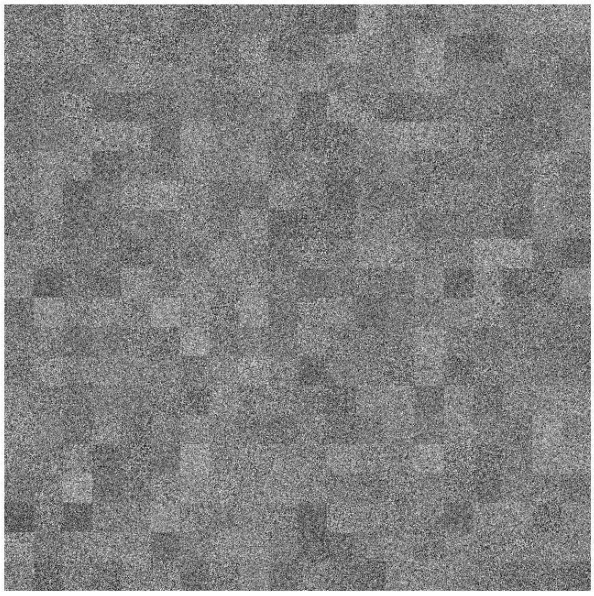


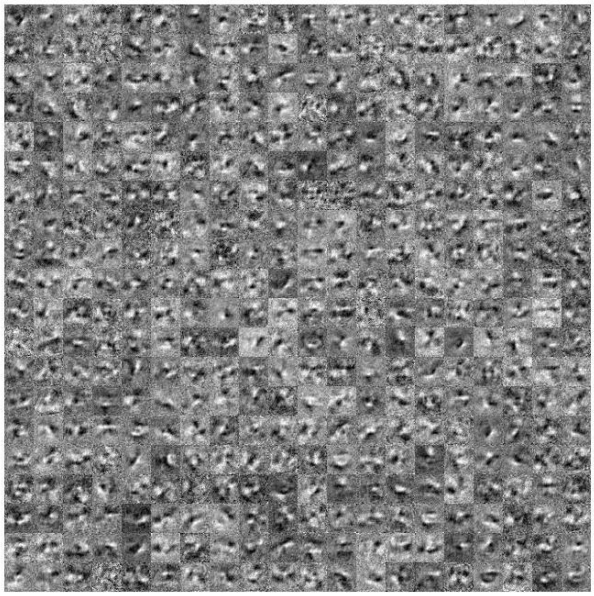


We can find from the images that when the training begins, bigger learning rate can help the network learn faster, so the slope in the first image is larger than that in the second image which represents the learning procedure for learning rate 0.001. But later on, too big learning rate may cause stop of the increase for accuracy. So during my best network, after the first 30 epochs training, I turn the learning rate smaller in order to make it learns better.

Q3.1.3

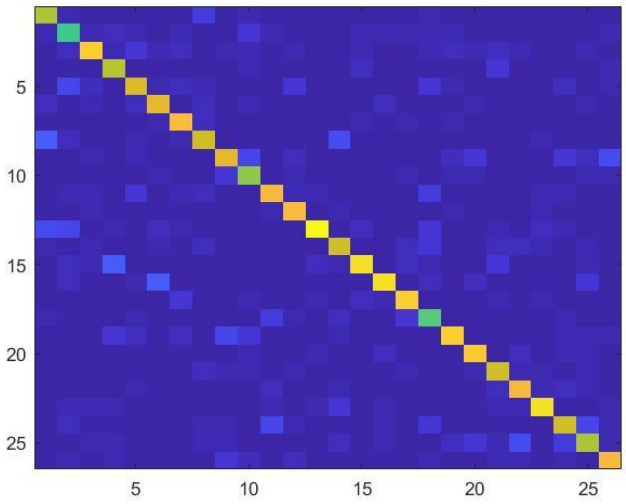
Here are the images for the weights of first layer, the first one is the weights just after initialization, and the second one is the weights that has been trained. We can easily find from the images that the weights before training is actually random and irregular, because we use Gaussian distribution to initialize the weights matrix. But after training, the weights become much more regular.



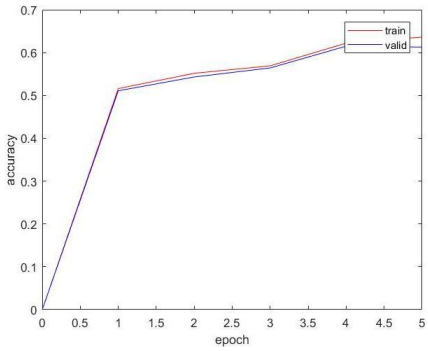


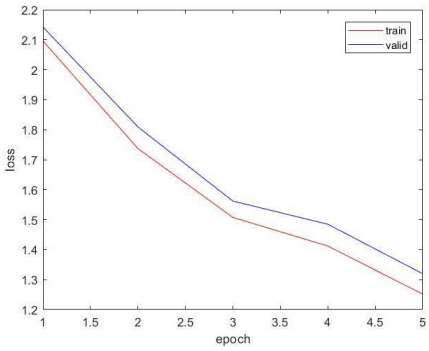
Q3.1.4

Here is the image of the confusion matrix:



Q3.2.1

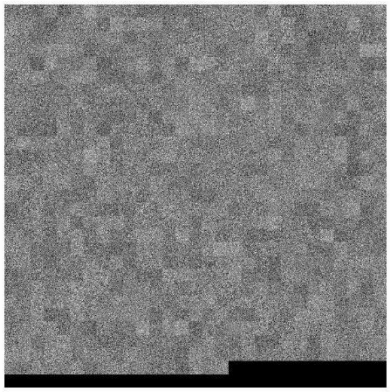


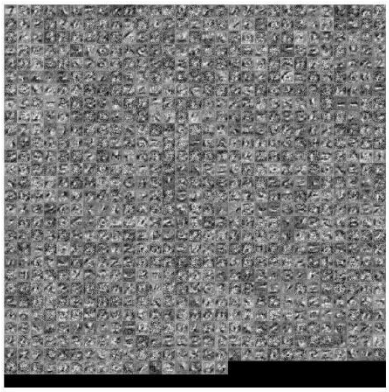


Here are the figures of accuracies and losses for training set and loss set.

We can easily find out that when we use the parameters of nist26 as the initial weights and bias for the new network nist36, it can obviously increase the accuracy for both training and validation from the beginning.

Q3.2.2

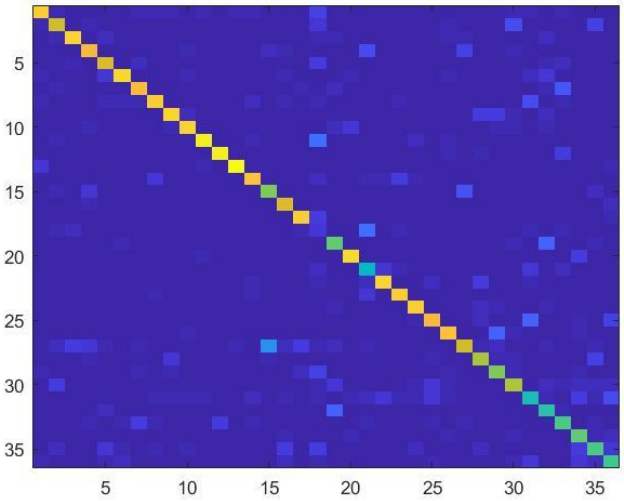




Once again, here are the weights matrix before and after training.

Q3.2.3

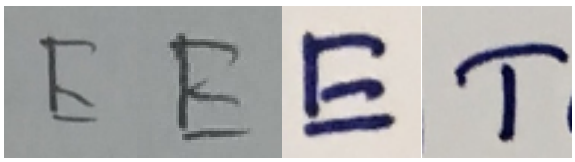
Here is the image of the confusion matrix:



Q4.1

In my opinion, there are two big assumptions that are too simpler and ignore some much more complicated situation in the reality.

The first one is that it assumes that one letter is a connected component, which means it requires all the part of a letter should be connected but actually the reality is not like this. For example, the letter ‘E’ and ‘T’ usually become a bug:



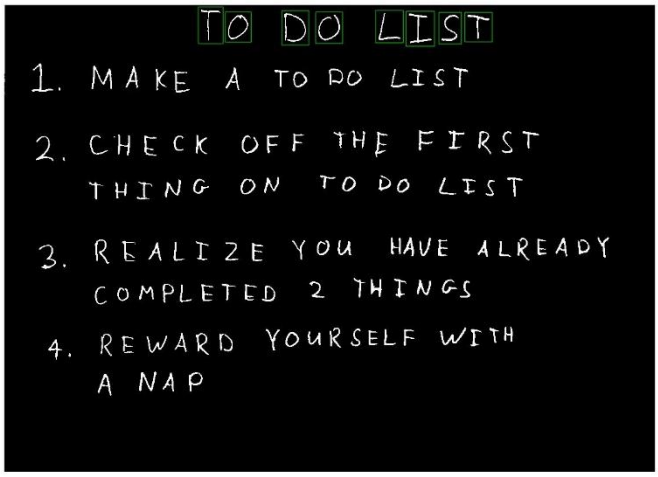
These letters are usually be extracted as two or more different parts respectively. And of course it assumes any two letters should be separated, so it can have a big problem when it comes to a

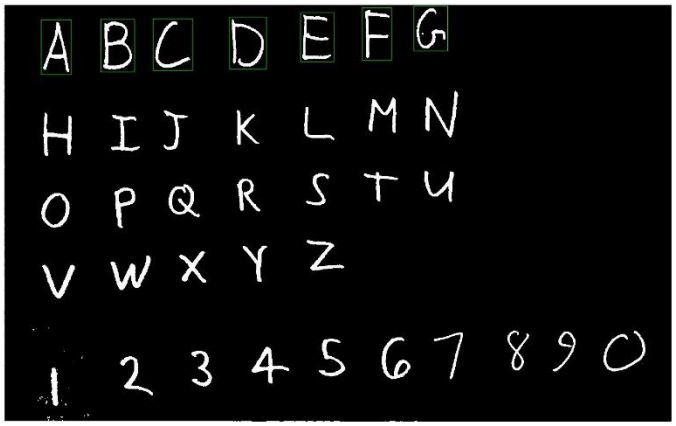
scribbled handwriting.

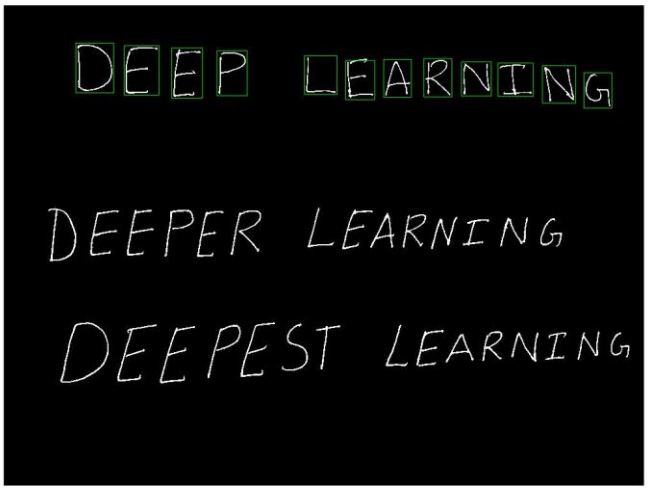
One other assumption that has some problem is that it assumes that all the letters should have almost the same size, or in other word, at least all the letters' size will not have a big difference. So in reality, some letters that is written much more smaller or bigger than others can be easily regarded as the noises or something else and throw away from the result.

Q4.3

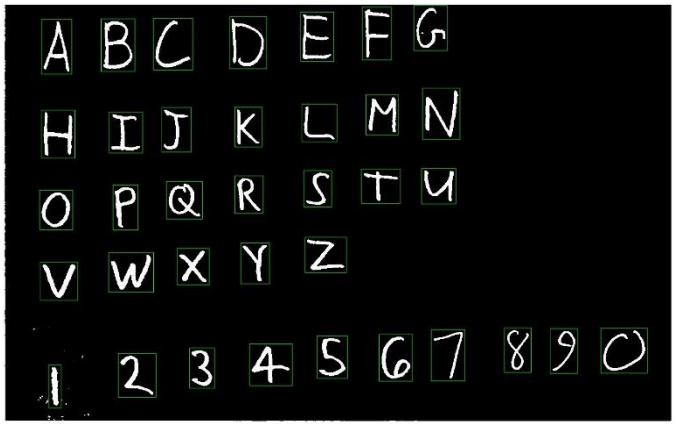
Here are the first line of every images:

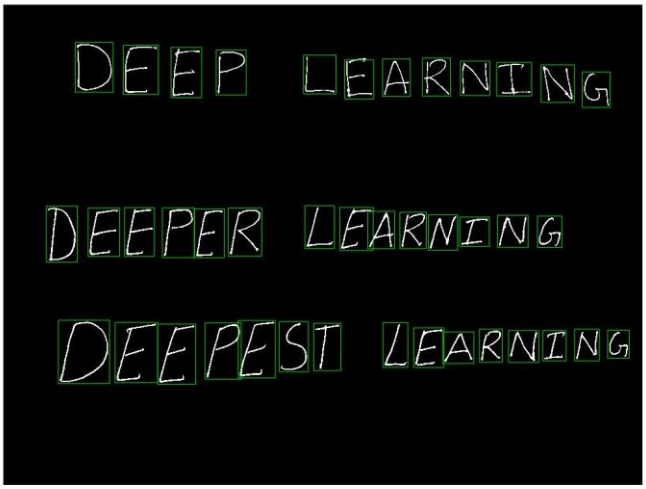






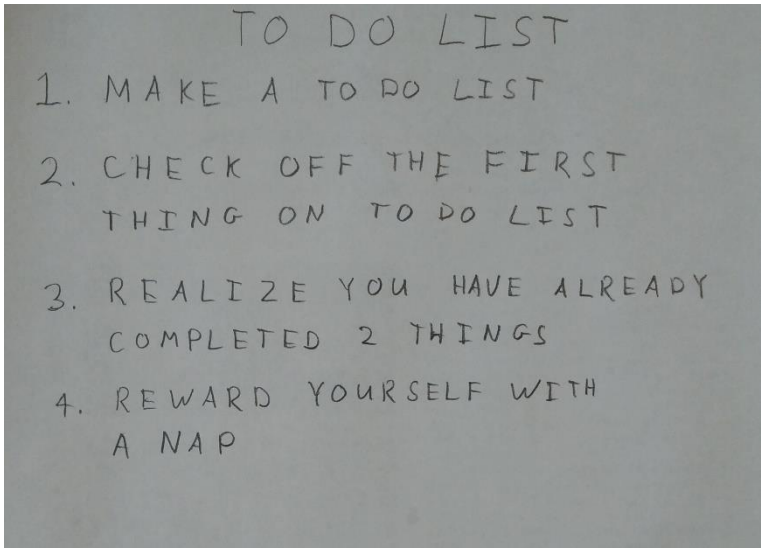
And here are the results of all the letters and numbers extracted from these 4 images:





Q4.5

Here are the texts extracted from the 4 images, I also put the original pictures here so it will be easier to compare them:

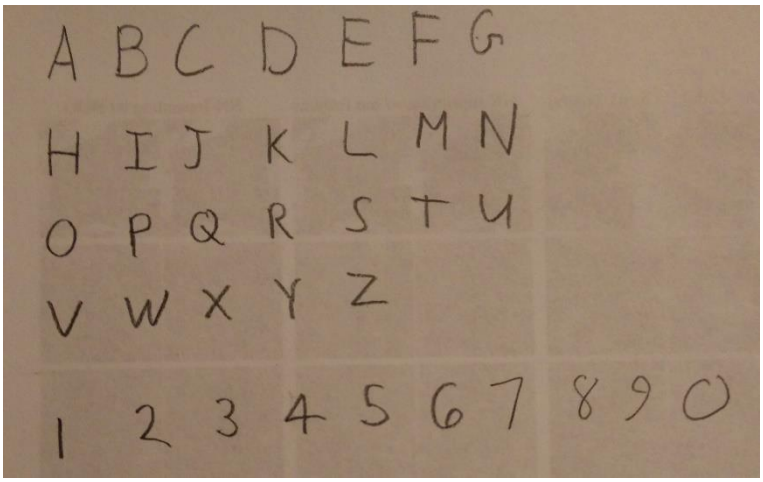


The extracted text is :

‘10d0list luakeatd20list 0check0fethefikst thing0nt0d0list 3u5alizey0unavealk6adt c0mpletcdzthingi 9rfwa2dy0ukselfwith anap’

```
text =

    '10d0list luakeatd20list 0check0fethefikst thing0nt0d0list 3u5alizey0unavealk6adt c0mpletcdzthingi 9rfwa2dy0ukselfwith anap'
```

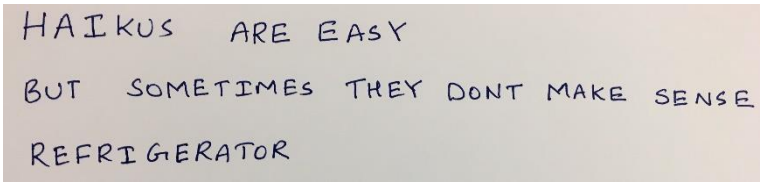


The extracted text is:

‘abrdgfg mijklmw qpqkstu vwxyz 8z3gsg3k83’

```
text =

    'abrdgfg mijklmw qpqkstu vwxyz 8z3gsg3k83'
```

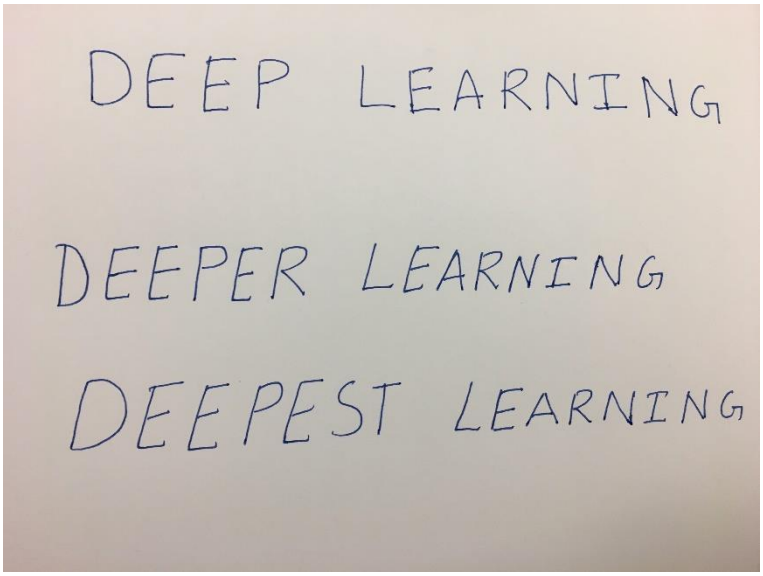


The extracted text is:

‘haiwu5aheeasx butsdme7imestheyddntmakes7ngg xefqigekatok’

```
text =

    'haiwu5aheeasx butsdme7imestheyddntmakes7ngg xefqigekatok'
```



The extracted text is:
‘dcccllldkrlrg dce1ck4faknirg dccfcs11fakring’

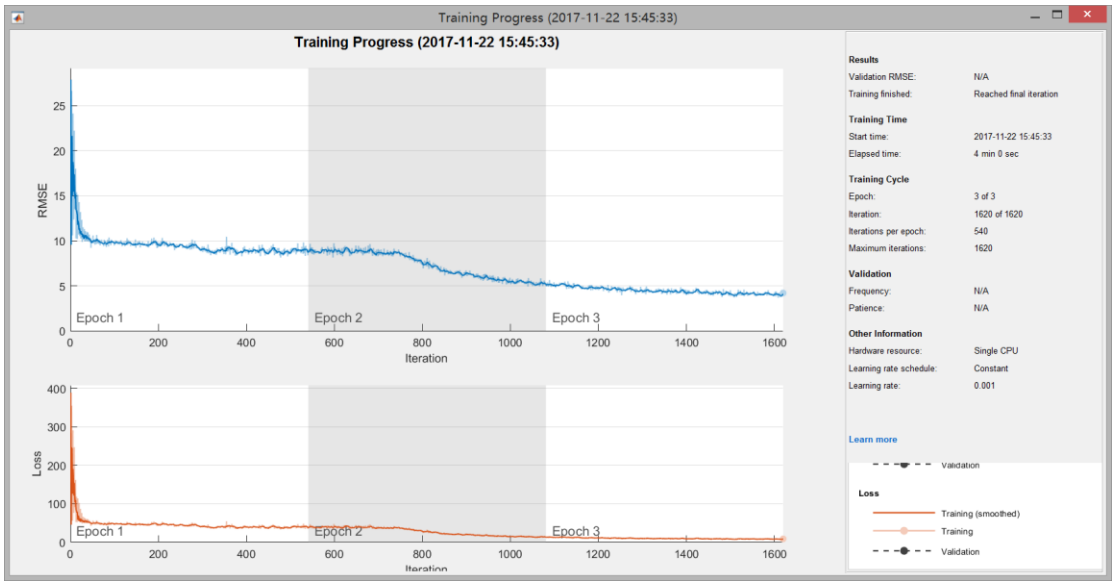
```
text =

    'dccc1ldkrlrg dce1ck4faknirg dccfcs11fakring'
```

Q5.1

The dimension of some convolution layer have some problem, they make some mistakes in the number of filters in each layer.

Q5.2.1



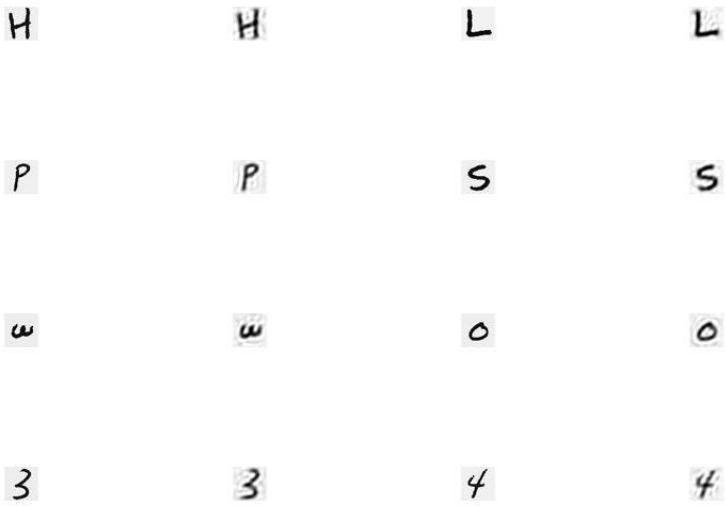
From the data we can easily find that the mini-batch loss decreases to about 7 after 3 epochs. And during the process of training, the speed of loss' decreasing will be slower and slower. In the beginning, I found that the mini-batch loss will become NaN just after few steps of training, so I changed the initial learning rate to 0.001 and the problem has been solved but it also caused the decrease of the training efficiency.

Q5.2.2

In order to increase the training performance, I tried several methods, and here are some of them:

1. I tried to set the LearnRateDropFactor as 0.2;
2. I set the LearnRateDropPeriod as 5

Q5.3.1



Q5.3.2

The average PSNR of the whole test set is 17.9927.

```
answer =

   17.9927
```

Q5.4.1

The projection matrix means the singular value matrix for the original matrix, in MATLAB, the singular value will be sort from small to big on the diagonal line, when we do PCA, we just need to take the front few singular value and we can almost reconstruct the original data, cause the bigger the singular value is, the more important it is.

In this question, the size of the singular matrix is 10800x1024, and we are requested to take the first 64 singular value in it, which means we should set all the parameters but the 64 components in the new projection matrix to be zeros, and then reconstruct the original matrix according to the formula that:

$$A = U \cdot S \cdot V'$$

The original singular matrix's rank is 1024 and the new one extracted by us is 64, because all the other members in the new matrix is 0.

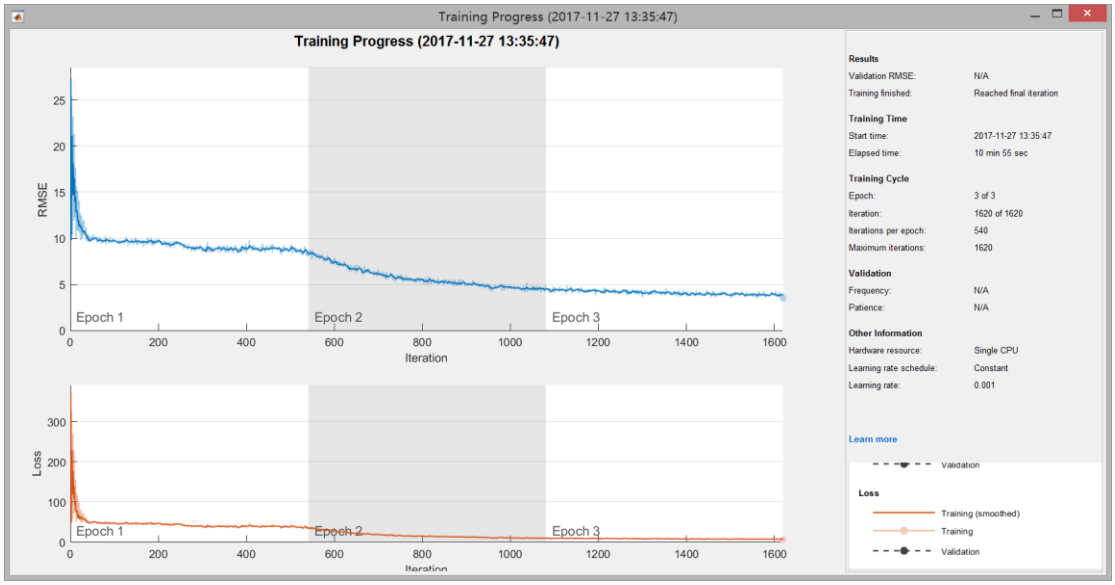
Q5.4.2

Here are the two of the corresponding images in test data. The left column are the original images, and the right column are the two compressed images by PCA.

Q5.4.3

The average PSNR for the test set through PCA is 18.8525. I found that the PNSR of PCA is higher than my own autoencoders, the reason I guess for this is that the PCA method will give up some secondary information in the images, and this kind of neglect may damage the quality of the images.

Q5.4.4



Here is the procedure I train the new network without any corrections.

And here are the same pictures I choose from last question:

The first column are the original images, and the second column are the PCA images, and the last column are the images from the training autoencoder. We can find that both of these two methods can save the basic information of the original images, but there are still some differences between them, for example, this autoencoder can just compress the images contained letters and numbers, but the svd method can almost be used to manipulate all kinds of images.

Q5.4.5

In the PCA method, according to the request of the question, we keep 64 parameters in the singular matrix, and there are 1800x1800 + 64 + 1024x1024 = 4,288,640. And there are 33,920 parameters in the autoencoders.

Q5.5

Here is the average PSNR for all the test:

```
ans =

   51.2334
```

We can figure out from the result that the PSNR of the JPEG2000 method is much higher than that of autoencoder method.

The reason why people still use JPEG rather than autoencoder now is that, we need to train the autoencoder every time before we want to compress different kinds of items, and the procedure of training will cost us lots of time. On the contrast, even though the JPEG will take more memory and need more parameters, it is more convenient to use it in different situations.