Jerry Hsiung
Computer Vision 16-720
Assignment 5
November 28, 2016

# Assignment #5

**Q1.1.1 Theory** [**5 points**] In training deep networks ReLU activation function is generally preferred to sigmoid, comment why?

ReLu function is popular because it is computationally fast without needing to calculate the exponential function. But most importantly, ReLu has a benefit over Sigmoid in avoiding the "vanishing gradient problem". In Sigmoid, If the input $>> 0$, then the output will be saturated to 1. If multiple inputs are large, then their corresponding outputs will all be saturated to 1, and therefore during back propagation, the gradient will be increasingly small. This will run into the vanishing gradient problem. ■

**Q1.1.2 Theory** [**5 points**] All types of deep networks use non-linear activation functions for their hidden layers. Suppose we have a neural network with input dimension $N$ and output dimension $C$ and $T$ hidden layers. Prove that if we have a linear activation function $\mathbf{g}$, then the number of hidden layers has no effect on the actual network.

If we have a linear function $g$, then despite of how many hidden layers we have, all of them can be concatinated to be a single linear function since applying linear functions multiple times will be the same as applying the composite linear function once: $g(x) = y, l(y) = z$ is the same as $f(x) = g(l(x)) = z$, where $f = (g \circ l)$.

Supposed from the given variables, we have input dimension $N$, output dimension $C$, $T$ hidden layers $H$ of dimension $H_1, H_2, \ldots, H_T$. Also supposed our linear activation functions $g_1, g_2, \ldots, g_T, g_C$, weights $W_1, W_2, \ldots, W_T, W_C$, and biases $b_1, b_2, \ldots, b_T, b_C$, then the output can be calculated by forward propagation as follow:

$$\begin{aligned} Output &= g_C(W_C(\ldots(W_2(g_1(W_1 \times input + b_1)) + b_2))\ldots)b_C) \\ &= g_C(W_C(\ldots(W_2(g_1(W_1 \times input) + g_C(W_C(\ldots(W_2(g_1(W_1 \times b_1))b_2))\ldots)b_C) \\ &= f(input) + b_f \end{aligned}$$

This is equal to linearly transform and add to an offset of the input. ■

**Q2.1.1 Theory [5 points]** Why is it not a good idea to initialize a network with all zeros. How about all ones, or some other constant value?

It is a bad idea to have all the initialial values to be zero or a constant. It is a bad idea because there is a high potential of running into vanishing gradient problem. Suppose at some layer all the inputs will be outputted the same, and at the layer the gradient will be the same and therefore the updates will be the same during parameters update. This runs into a repetition and in effect the layer is deactivated since it simply passes through the parameters with all the same values. ■

**Q2.1.3 Writeup [5 points]** Describe the initialization you implemented in Q2.1.2 and any reasoning behind why you chose that strategy.

In Q2.1.2, I have followed the suggestion of "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification by He et al." which is to initialize the weight by a random value devided by $\sqrt{(2/n)}$, where $n$ is the number of nodes in that particular layer. For biases, I initialize them to be 0, since multiple resources online have suggested that 0 can work as well. ■

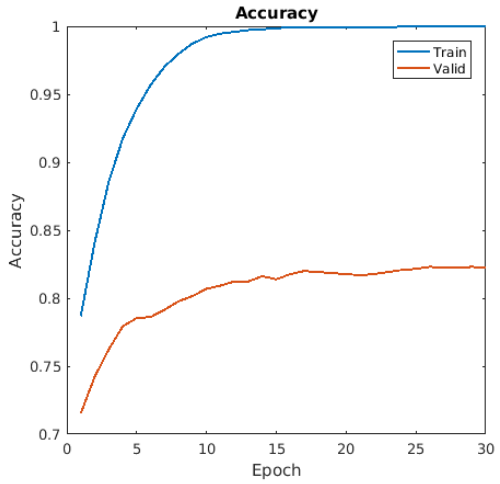
**Q2.4.1 Theory [5 points]** Give pros and cons for both stochastic and batch gradient descent. In general, which one is faster to train in terms of number of epochs? Which one is faster in terms of number of iterations?

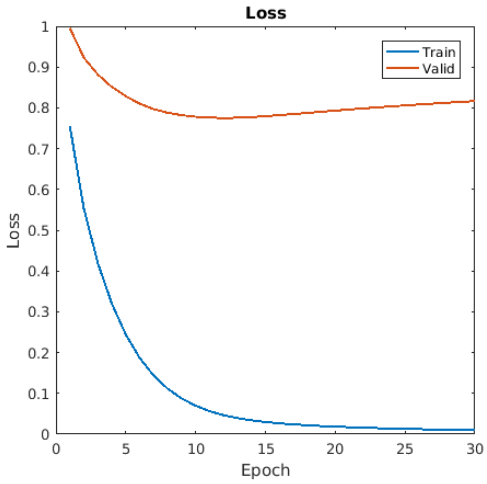
Stochastic gradient descent runs parameter update after every single sample in an epoch, and therefore, it requires much more computation time than batch gradient descent, which updates once after all the samples have been run through. However, stochastic gradient descent achieves higher accuracy, requires less computation power than batch gradient descent, and requires less amount of epochs to train since the parameters have been udpated more frequenctly. There is also mini-batch update which updates after a small batch of samples, (usually a few hundred samples). Mini-batch compromises the pros and cons of stochastic and batch gradient descent.

I implemented stochastic gradient descent update. ■

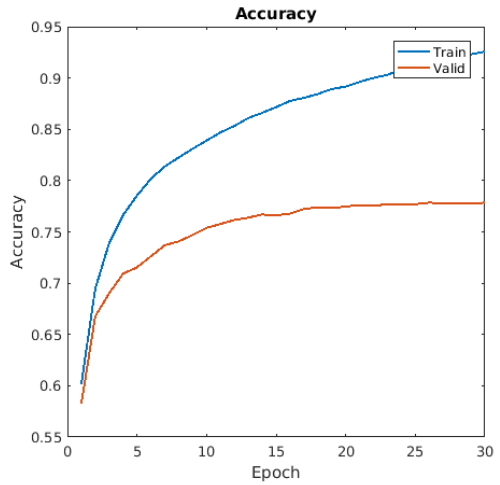
**Q3.1.2 Writeup [5 points]** Use your modified training script to train two networks, one with learning rate 0.01, and another with learning rate 0.001. Include all 4 plots in your writeup. Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set.

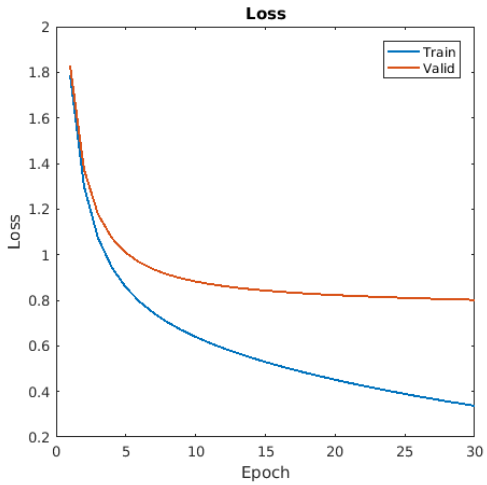
The results are shown below:
leanring rate 0.01:




learning rate 0.001:



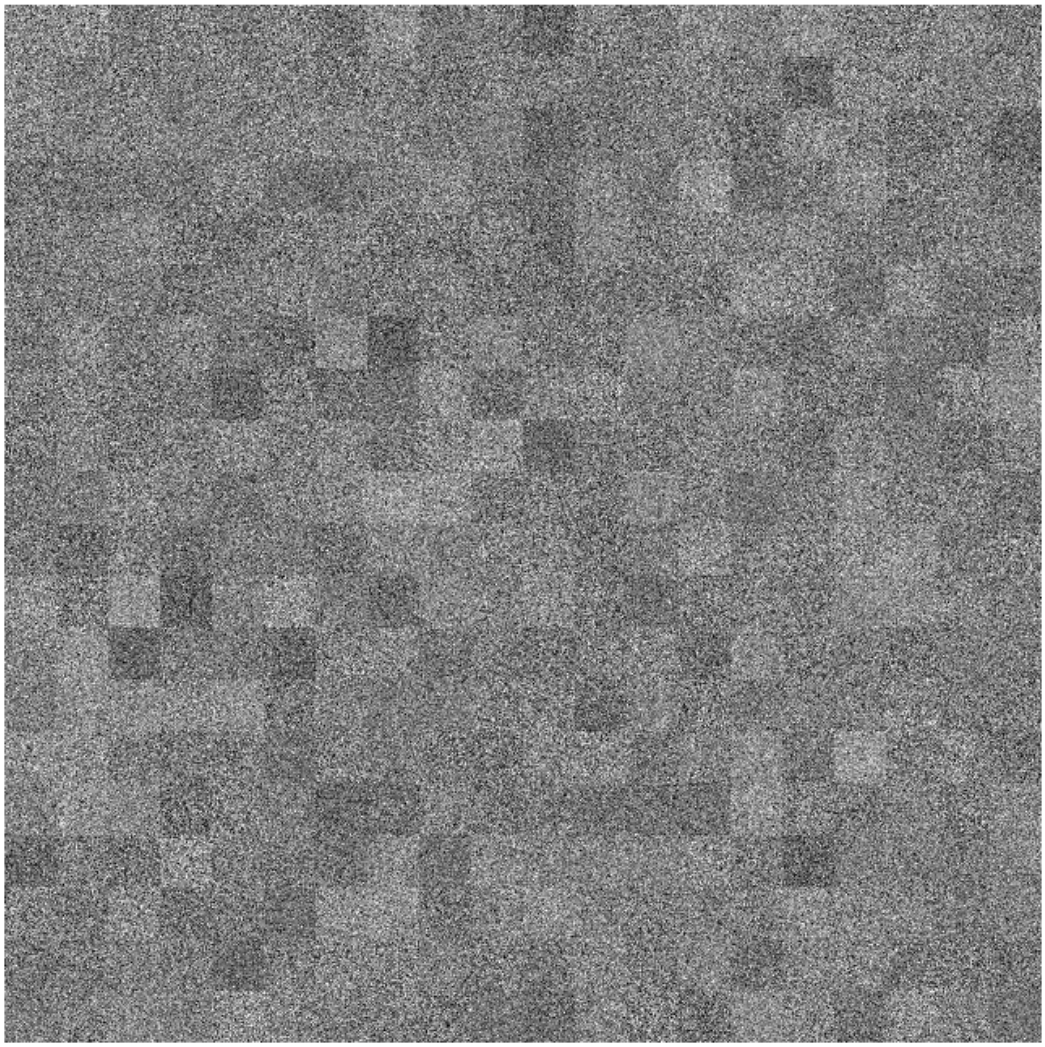

From above graphs, we see that the learning rate has sigificantly affect the training: the higher learning rate, the faster the training goes, and vice versa. However in my implementation, the traning is too fast such that it overfits the training data. We can tell from the increase in loss of validation data after epoch 12 or so, and the 100% accuracy of training data. Using the parameters of learning rate 0.01, we achieve the accuracy of to test set.

The accuracy from learning rate 0.01 is 82.31%, while

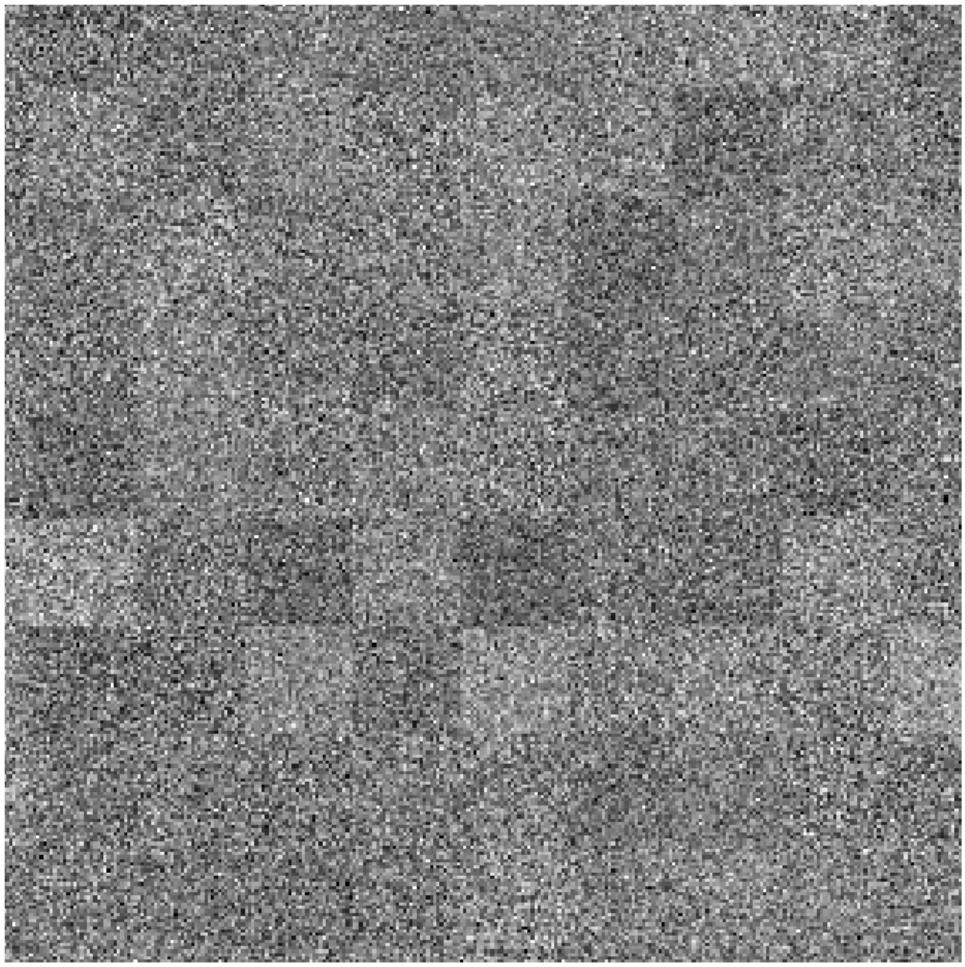
■

**Q3.1.3 Writeup [5 points]** Using the best network from the previous question, report the accuracy and cross-entropy loss on the test set, and visualize the first layer weights that your network learned (using `reshape` and `montage`). Compare these to the network weights immediately after initialization. Include both visualizations in your writeup. Comment on the learned weights. Do you notice any patterns?

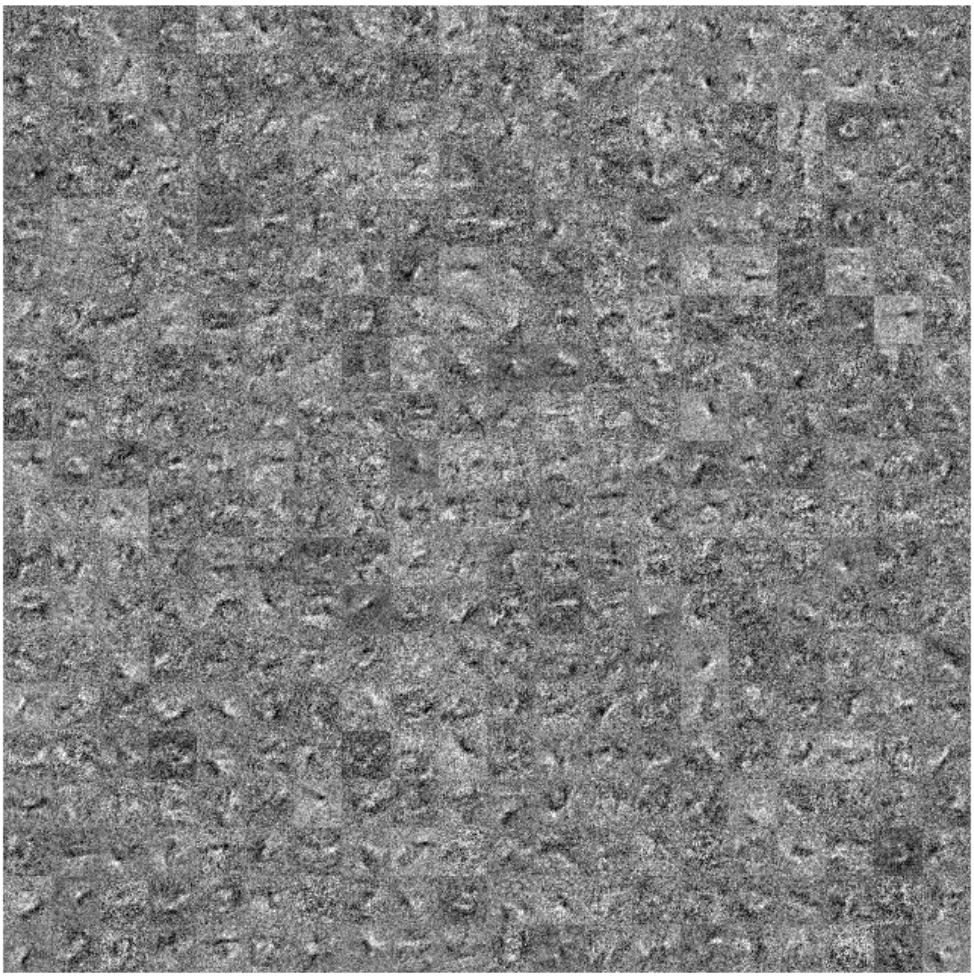
The first layer weight immediately after initialization:



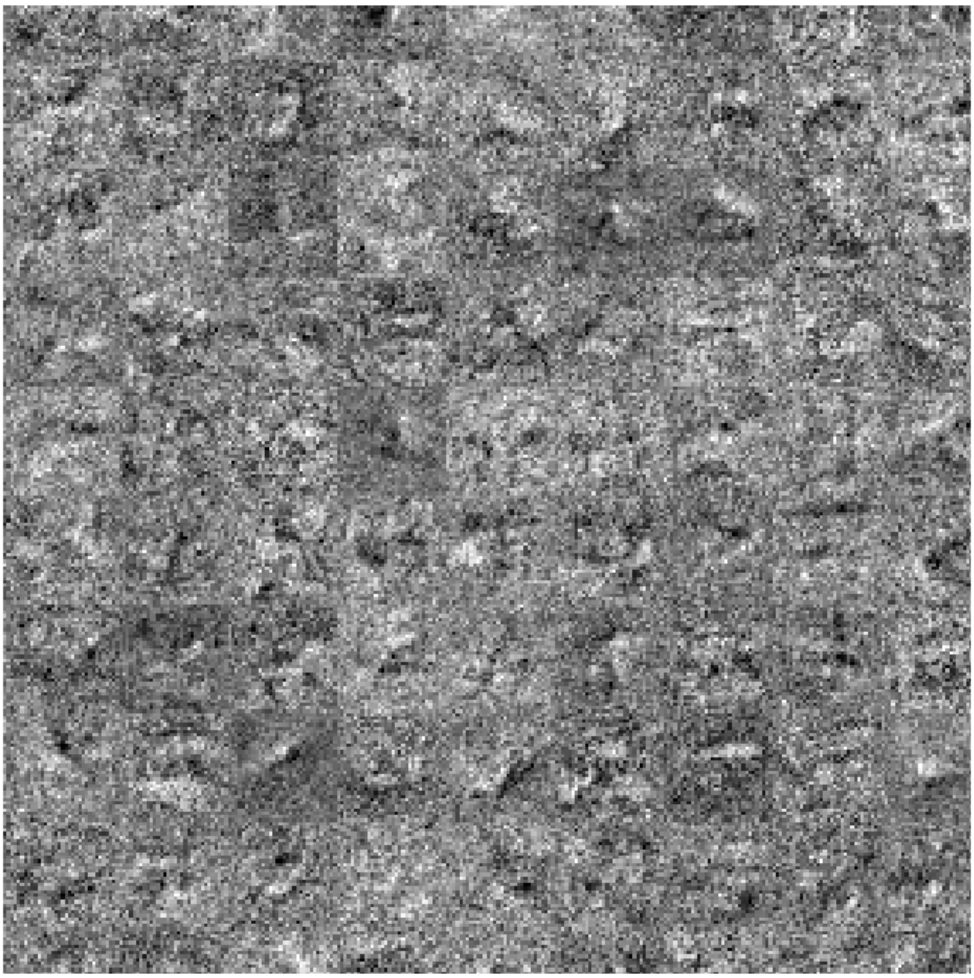
Zooming in:



The first layer weight after 30 epochs:



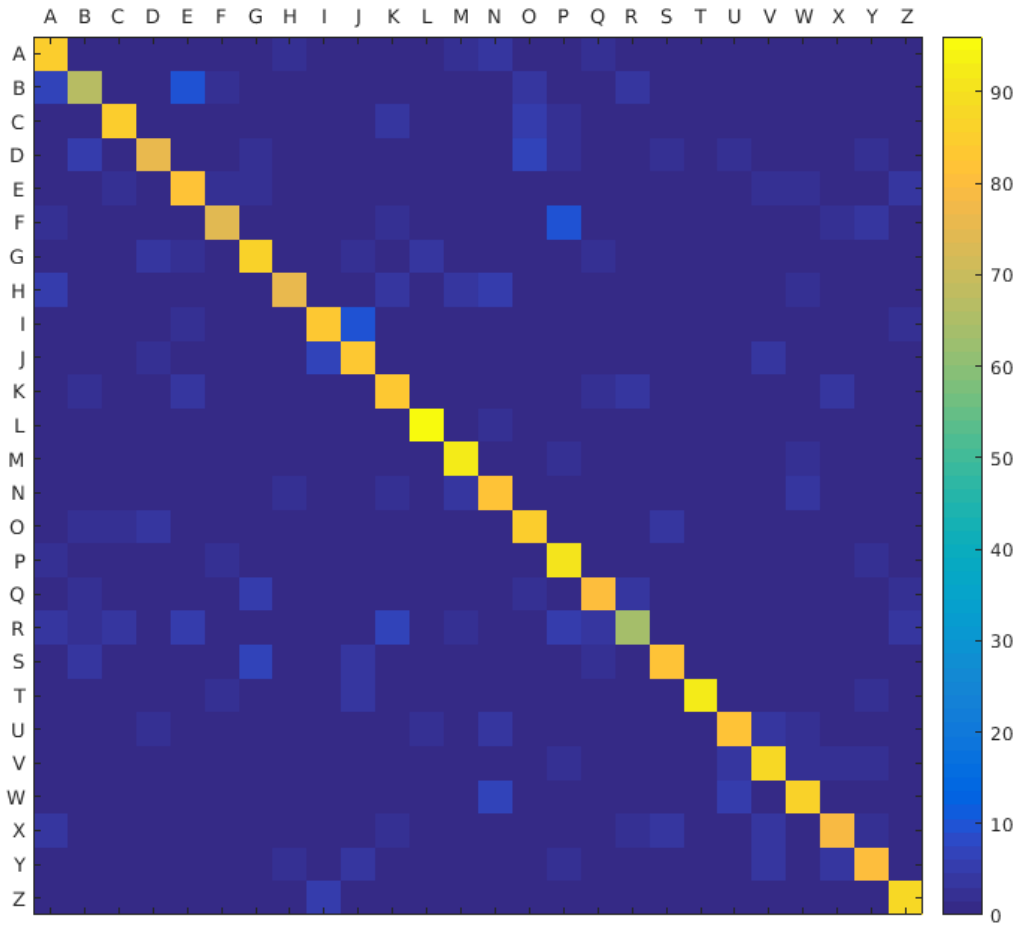
Zooming in:



As we can see, the weight after initialization doesn't appear to be having any structures, which makes sense because we randomly initialize the weight. After 30 epochs of training, we can see that the weight at each node appears to have some structures, such as edge detector, blob detector, or laplacian function. This shows that during the training, the network has pick up significant features of the training data, and use them to modify the weights. ■

**Q3.1.4 Writeup [5 points]** Visualize the confusion matrix for your best model as a $26 \times 26$ image (upscale the image so we can actually see it). Comment on the top two pairs of classes that are most commonly confused.

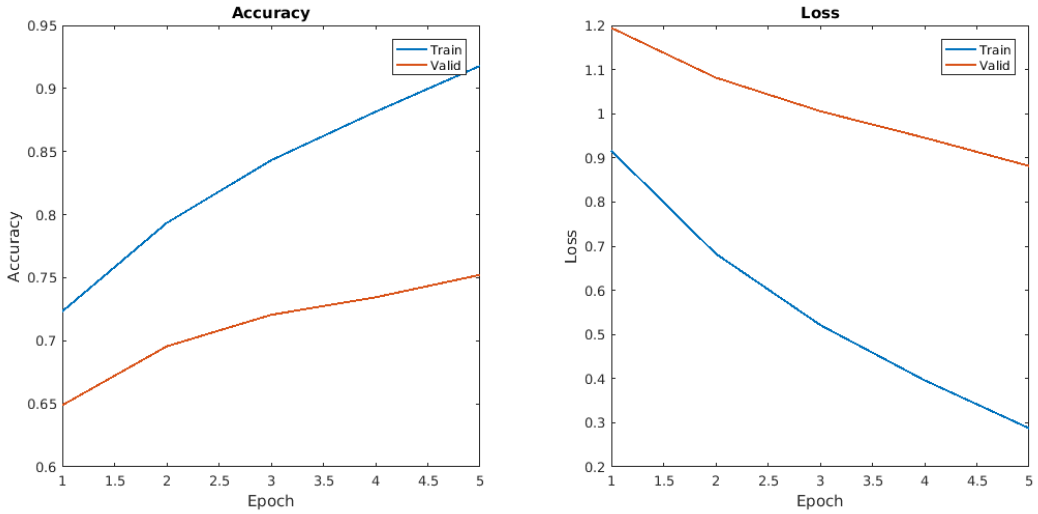
The confusion matrix is shown below:



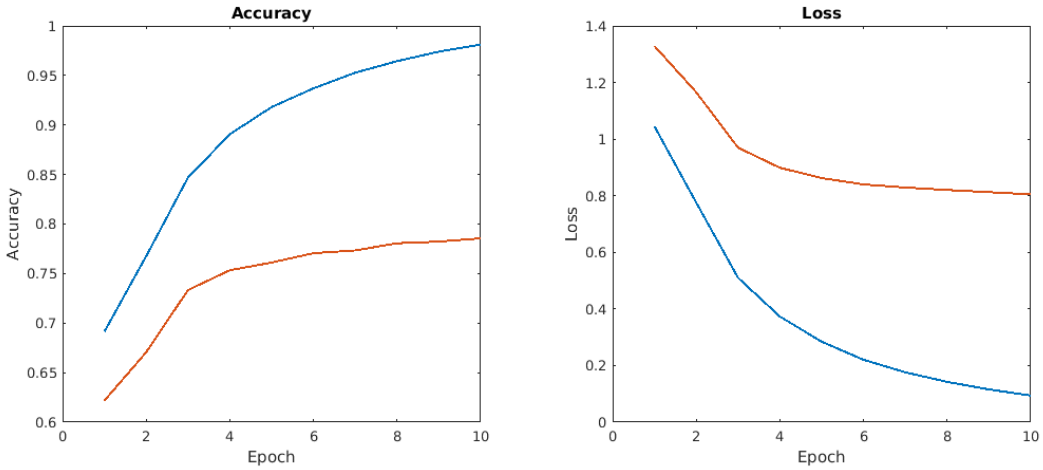

As shown, *I* and *J* are easily confused because their structure is similar. Other comfusing cases are *F* and *P*, *D* and *O*, *B* and *E*, which are casese where the structures are also similar, and therefore the network gets confused about them. ■

**Q3.2.1 Code/Writeup [10 points]** Make a copy of `train26.m` and name it `finetune36.m`. Modify this script to load the data from `nist36_*.mat`, and train a network to classify both written letters and numbers. Finetune (train) this network for 5 epochs with learning rate 0.01, and include plots of the accuracy and cross-entropy loss in your writeup.

Loss and accuracy after 5 epochs:



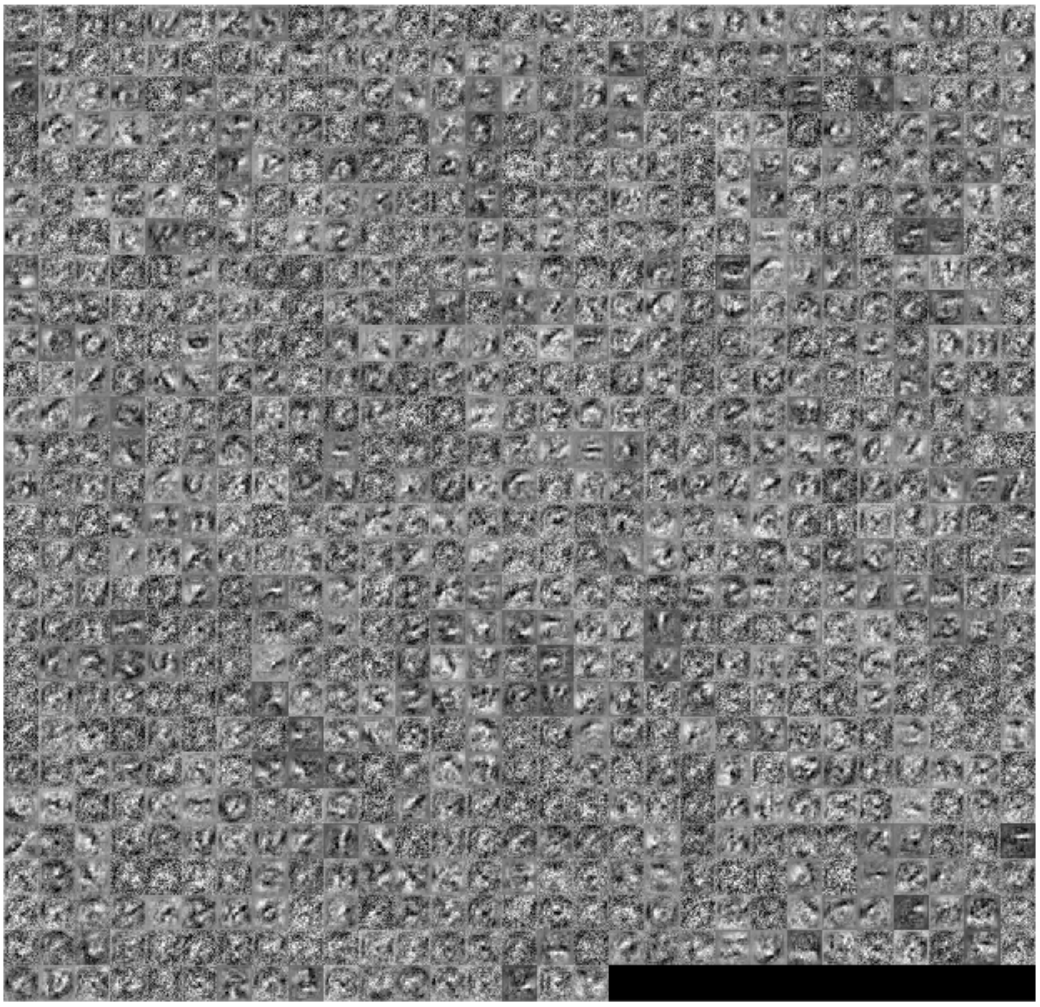

Loss and accuracy after 10 epochs:



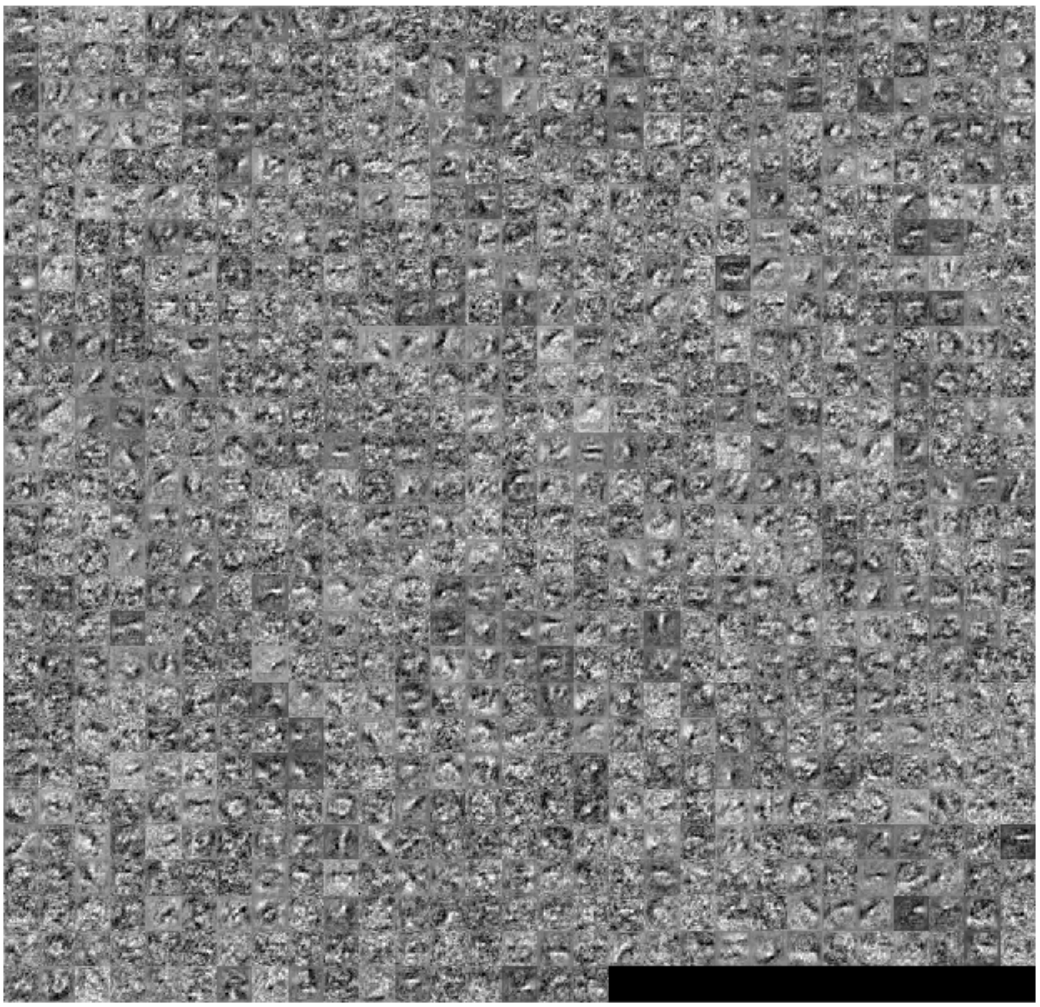

■

**Q3.2.2 Writeup [5 points]** Once again, visualize the network's first layer weights before and after training. Comment on the differences you see. Also report the network's accuracy and loss on the test set.

The first layer weight immediately after initialization:



The first layer weight after 5 epochs:



We can see that both weights show structures, but after the training, there are less ”noise” in the sense that the strutures become much more apparent. This is because after training, the network has enhanced those pre-trained structures. ■

**Q3.2.3 Writeup [5 points]** Visualize the confusion matrix for your best model as a $36 \times 36$ image (upscale the image so we can actually see it). Comment on the top two pairs

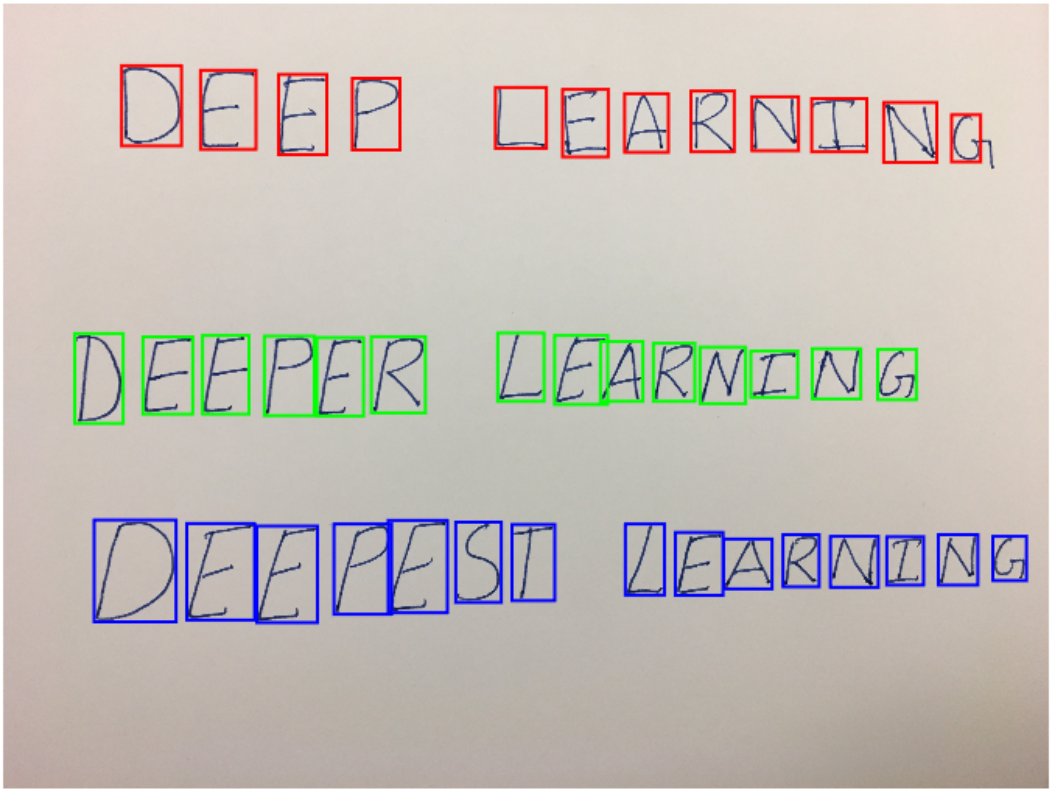
■

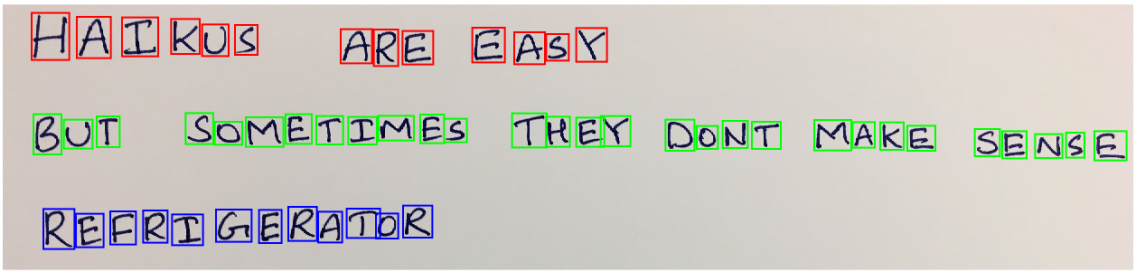
**Q4.1 Theory [5 points]** The method outlined above is pretty simplistic, and makes several assumptions. What are two big assumptions that the sample method makes. In your writeup, include two example images where you expect the character detection to fail (either miss valid letters, or respond to non-letters).

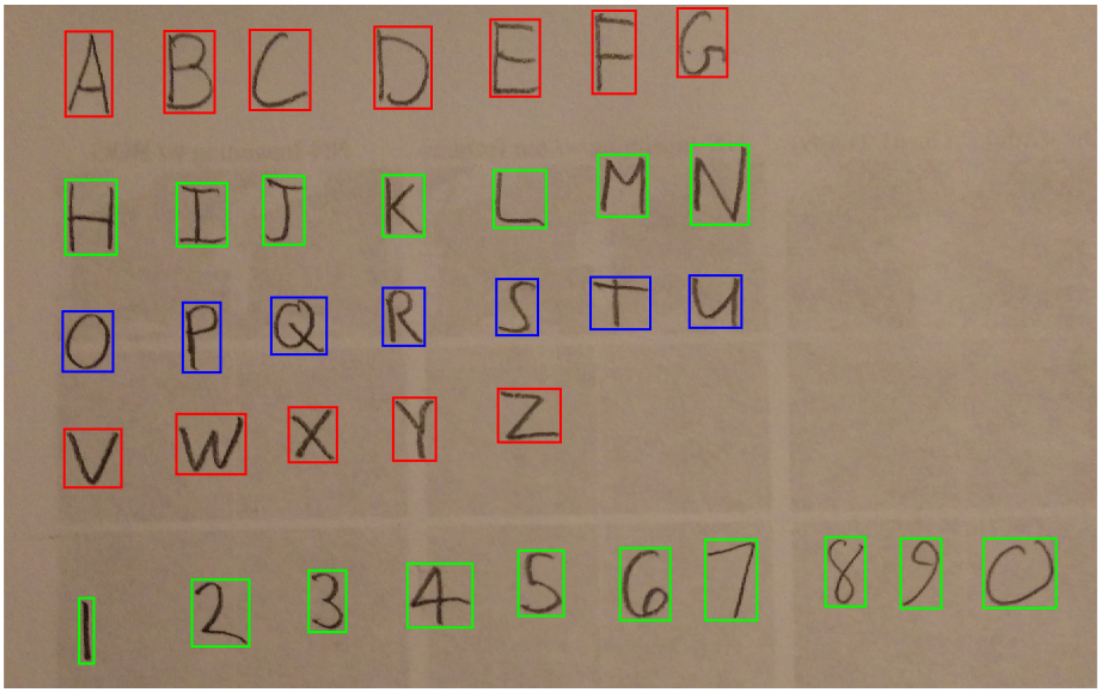
The algorithm assumes that each letter is spatially apart, and therefore each letter can be extracted by itself. It also assumes that the letter has a significant difference in intensity than the background, so the algorithm can tell the letters from the background using a simple threshold value method. Lastly, the algorithm assumes that each line of text is close to horizontal, so it can tell which word is on which line. ■

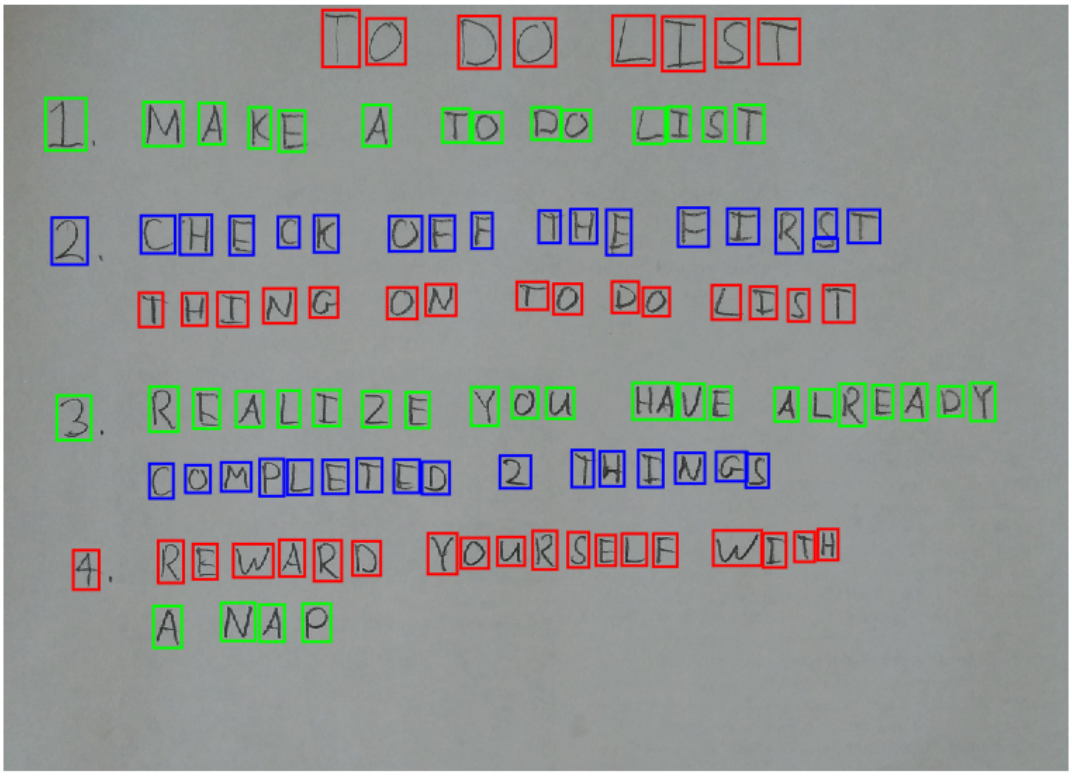
**Q4.3 Writeup [5 points]** Run `findLetters(..)` on all of the provided sample images in `images/`. Plot all of the located boxes on top of the image to show the accuracy of your `findLetters(..)` function. Include all the result images in your writeup.

The results are shown below, which each color represents different line:




A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
1 2 3 4 5 6 7 8 9 0


TO DO LIST
1. MAKE A TO DO LIST
2. CHECK OFF THE FIRST THING ON TO DO LIST
3. REALIZE YOU HAVE ALREADY COMPLETED 2 THINGS
4. REWARD YOURSELF WITH A NAP

**Q4.5 Writeup [5 points]** Run your `extractImageText(..)` on all of the provided sample images in `images/`. Include the extracted text in your writeup.

haha ■