16-720 HW5 Optical Character Recognition using Neural Network

Xiaoyu Bai (xiaoyub)

Q1.1.1

The major advantage of ReLU is to **prevent gradient vanish**. For example, in the function f(x) when x >0 and x has large value, sigmoid tends to have very small gradient while ReLU has constant gradient. The other benefit is **sparsity**. When x<0, ReLU has constant value of f(x)=0 so the system enjoys sparse representation which is more beneficial than dense representation. Also ReLU's **training speed** is faster since the gradient is constant for x>0 and x<0 respectively.

Q1.1.2

If both f(x) and g(x) are linear, then f(g(x)) is also linear.

In this particular case, suppose the input layer is $N = [n_1, n_2, \cdots, n_N, 1]$ which has N+1 dimension (the 1 is for bias). So the input of i_{th} hidden unit will be $w_{ti}N$ where w_{ti} is the weight vector for i_{th} hidden layer which also has N+1 dimension. The output of i_{th} hidden unit will be $t_i = a_{ti}w_{ti}N + b_{ti}$. Here the sub-notation t just mean it is for the hidden layer. If we say $T = [t_1, t_2, \ldots, t_T, 1]$. The input for j_{th} output unit is $w_{cj}T$. And the output of j_{th} output unit is $c_j = a_{cj}w_{cj}T + b_{cj}$. We set the output $C = [c_1, c_2, \cdots, c_C]$. So we see that T is a linear function of N, and C is a linear function of T, which means C is also a linear function of N. Therefore the hidden layer acts just like it does not exist.

Q2.1.1

First, if the neurons start with the same weights, then all the neurons will follow the same gradient. And they will most likely end up doing the same thing as one another.

Second, neural networks tend to end up with local minima. So it will be good to start at different places.

Q2.1.3

I select each element in *W* and *b* randomly from Gaussian distribution with mean value of 0 and sigma of 0.1. In this way, W^TX will vary around 0 value to keep gradient at reasonable magnitude. Make W vary around 0 is to avoid gradient vanishing. If we set W value from 0-1 then W^TX will be all positive given all positive X values, and the sigmoid will be all concentrated at 0.5-1 which may possibly lead to gradient vanishing.

Q2.3.1 *Backward Propagation*

Case 1 (an output node):

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial a}\frac{\partial a}{\partial w} \quad and\ L = -\sum_{i=1}^{c} y_i \cdot \log(h_i)\,, h_i = \frac{e^{a_i}}{e^{a_1} + e^{a_2} + \cdots + e^{a_c}}, z_i = w_i^T \cdot x + b$$

For softmax,

$$\frac{\partial h_i}{\partial a_j} = \begin{cases} h_i(1 - h_i) & if\ i == j \\ -h_i \cdot h_j & if\ i \cong j \end{cases}$$

Through derivation,

$$\frac{\partial L}{\partial h}\frac{\partial h}{\partial a} = [h_1 - y_1, h_2 - y_2, \ldots, h_c - y_c]$$

And

$$\frac{\partial z}{\partial w} = x$$

Case 2 (not output node):

$$\frac{\partial L}{\partial w\_this} = \frac{\partial L}{\partial h\_next} \cdot \frac{\partial h\_next}{\partial a\_next} \cdot \frac{\partial a\_next}{\partial h\_this} \cdot \frac{\partial h\_this}{\partial a\_this} \cdot \frac{\partial a\_this}{\partial w\_this}$$

And

$$\frac{\partial a\_next}{\partial h\_this} = w\_next$$

$$\frac{\partial h\_this}{\partial a\_this} = \sigma(a\_this) \cdot (1 - \sigma(a\_this))$$

$$\frac{\partial a\_this}{\partial w\_this} = h\_prev$$

This means for each hidden layer, we calculate $\frac{\partial L}{\partial a\_this}$, and this value could be used as $\frac{\partial L}{\partial a\_next}$ for the previous layer, meaning for each adding hidden layer, we only need to add $\frac{\partial a\_next}{\partial h\_this} \cdot \frac{\partial h\_this}{\partial a\_this}$.

$$\frac{\partial L}{\partial w\_this} = \boldsymbol{\frac{\partial L}{\partial h\_next} \cdot \frac{\partial h\_next}{\partial a\_next}} \cdot \boldsymbol{\frac{\partial a\_next}{\partial h\_this} \cdot \frac{\partial h\_this}{\partial a\_this}} \cdot \frac{\partial a\_this}{\partial w\_this}$$

The bold part can be used as $\frac{\partial L}{\partial a\_this}$ for the previous layer, and we add the red part for previous layer.

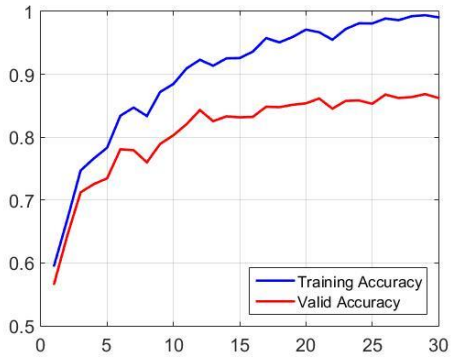
Q2.4.1

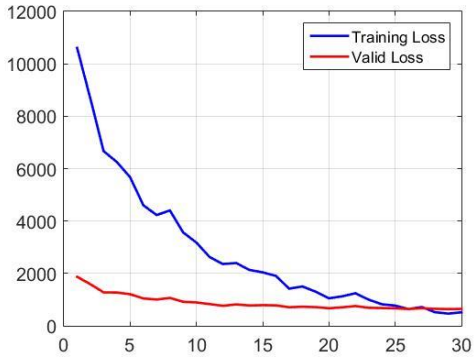
So first to clarify the concept of epoch and iteration, from my understanding, one epoch means one forward pass and one backward pass of *all* the training examples; and one iteration means one pass of a particular batch size of data.

Therefore, full batch GD is faster to train in terms of number of epochs. And SGD is faster to train in terms of number of iterations.

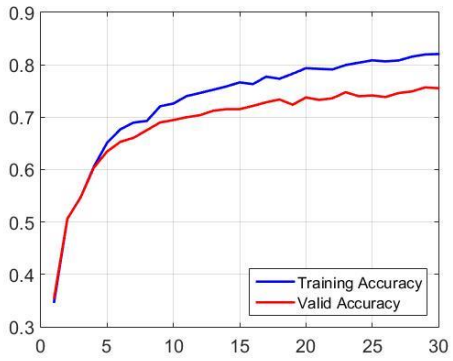
Q3.1.2

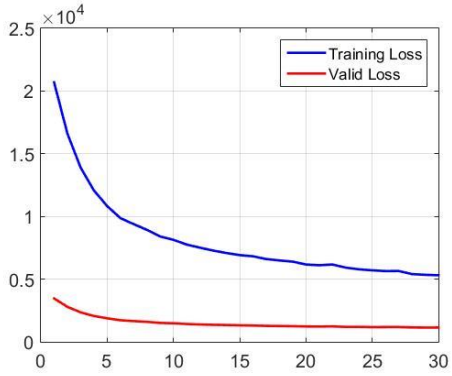
Learning Rate = 0.01





Learning Rate = 0.001



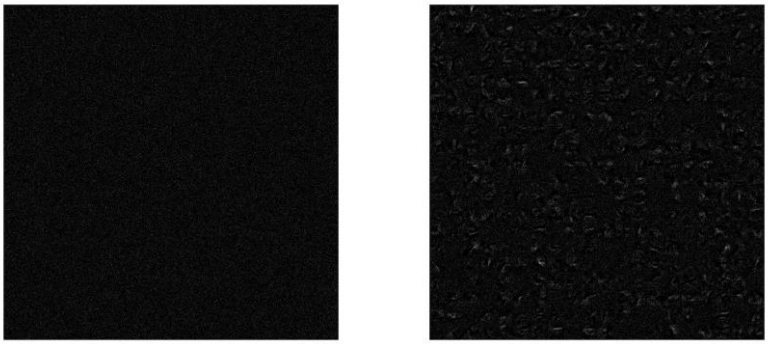


Here I am using stochastic GD. By comparison, small learning rate yields smoother curve due to the smaller step for each iteration. However in current scenario, 0.001 is too small for 30 epochs since the curve has not saturated yet. A better way for doing might be adaptive learning rate which means larger learning rate from beginning and smaller learning rate later.

The best test accuracy I achieve from scratch is using learning rate=0.01 with 30 epochs. The test accuracy is 0.87231.

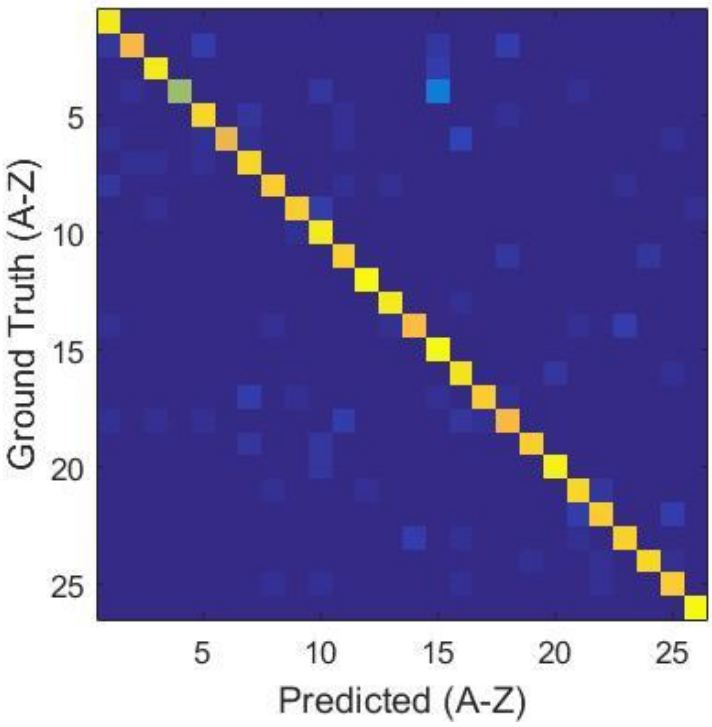
Q3.1.3

The cross entropy loss I got from previous best model is 1231.47853.



The above two figures shows first layer weights of initialization (left) and after training (right). At initialization the weight is nothing but just a mess, while after training it shows some certain pattern but hard to recognize what it is.

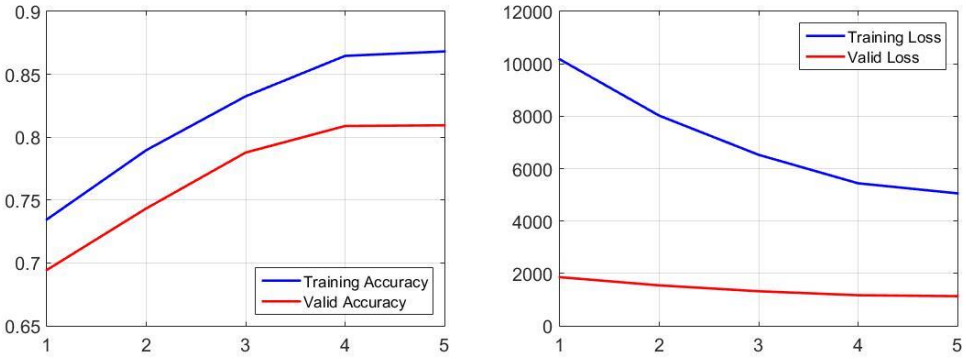
Q3.1.4 *Confusion Matrix*



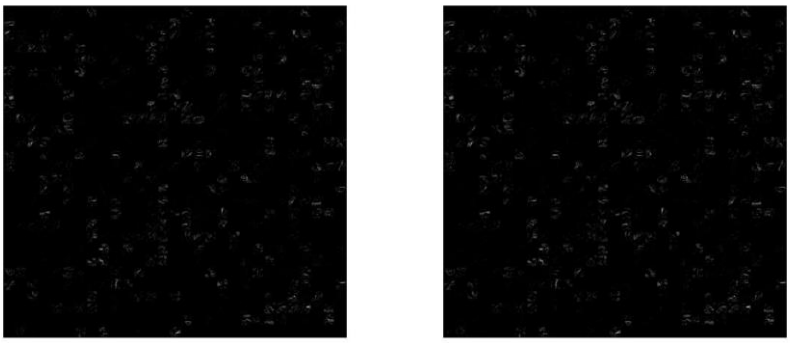
The most distinguished wrong prediction is at [4, 15] which means misrecognizing ‘D’ as ‘O’.

There is a less significant one at [17, 7] which means misrecognize ‘Q’ as ‘G’.

Q3.2.1

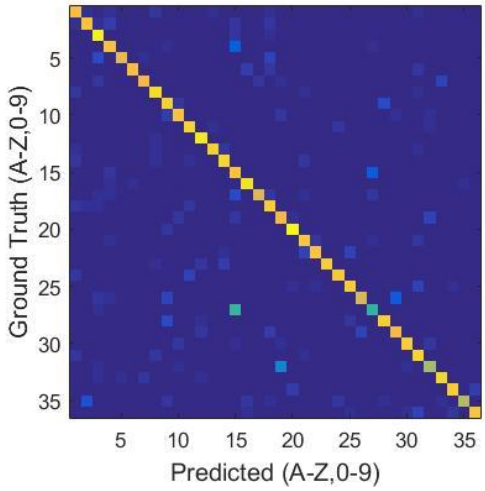


Q3.2.2



With fine tuning, the above two figures shows first layer weights of initialization (left) and after training (right). It can hardly be observed any difference between them. I think this means after 60 epochs, the first layer weight is already well trained, and the further training mostly only deal with weight at further layers.

Q3.2.3 *Confusion Matrix*



One significant shinny patch is at [27, 15] which means misrecognizing 0 as O, which is understandable since the difference is really trivial.

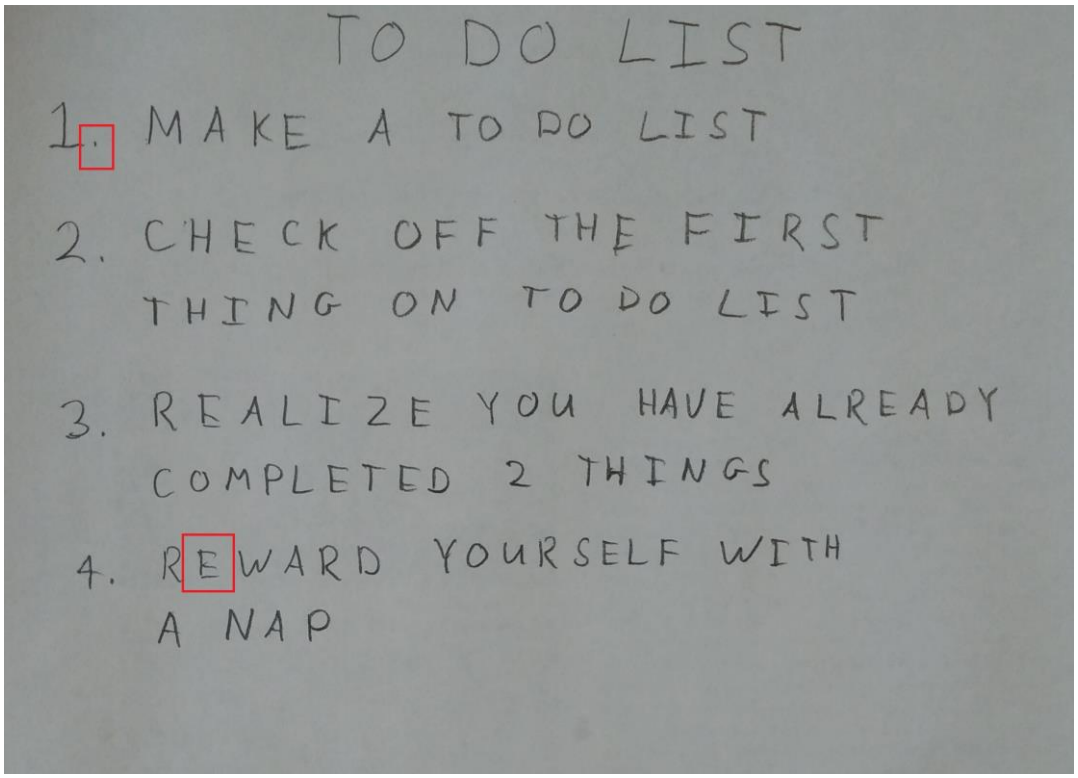
Another one is at [32, 19] which means misrecognizing 5 as S, since the way it curves looks similar.

Q4.1

I think the two assumptions are:

1. All the characters can be clearly distinguished from background, meaning ignoring the illumination of the image.
2. All the characters are well separated from each other, meaning the distance between two adjacent characters cannot be too close.

Two places that I expect the character detection to fail:



First thing is the dot might be recognized as a character by mistake.

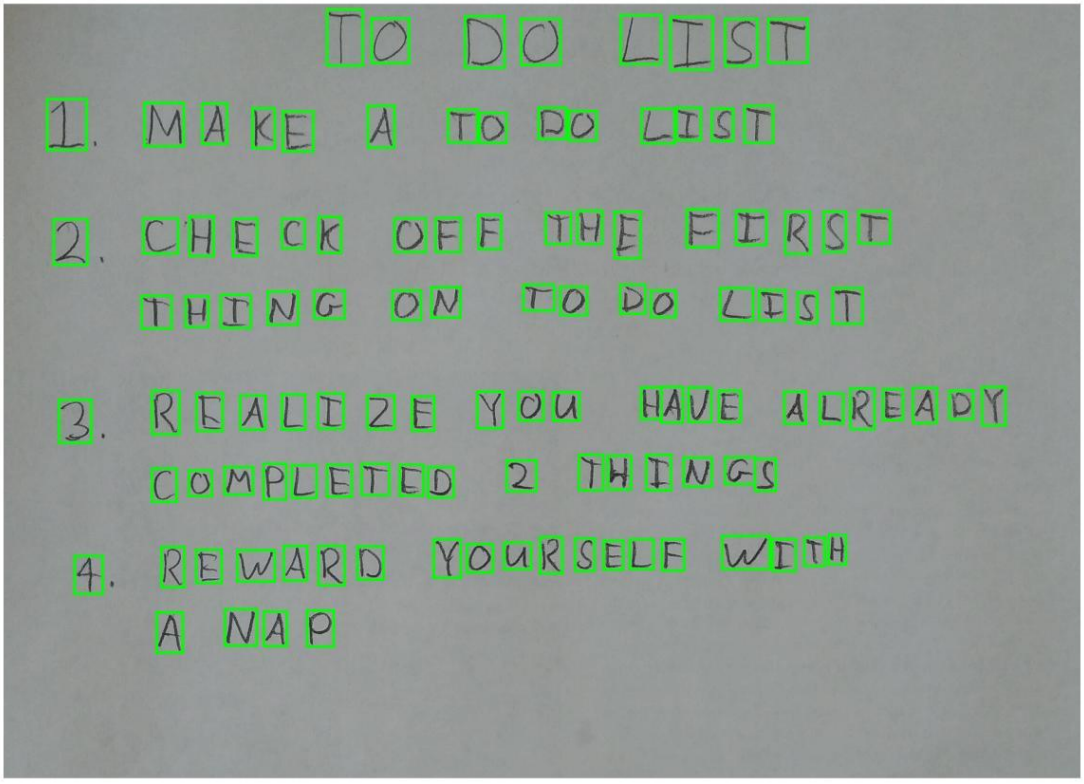
Second is that the letter *E* has been separated into two parts (‘F’ + ‘-’), so it is possible to be misrecognized as two different characters.

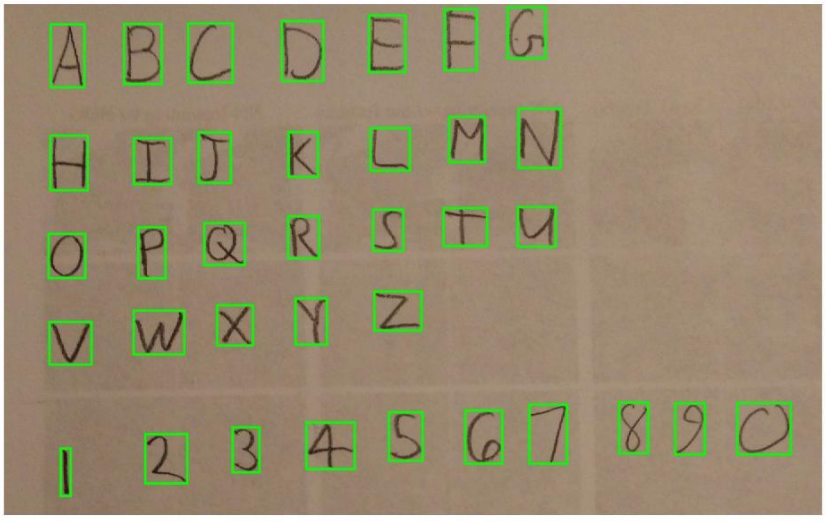
Q4.2 Reference:

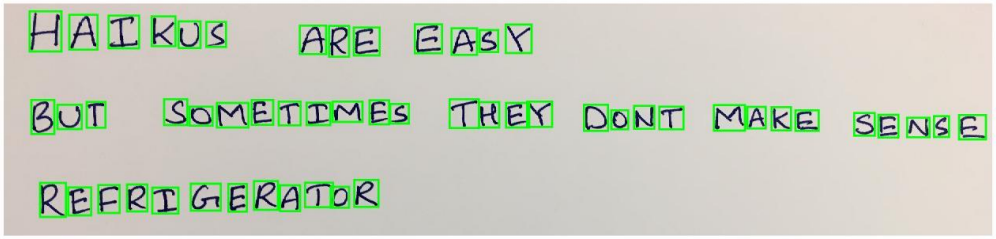
https://www.mathworks.com/matlabcentral/answers/110412-extract-text-from-image

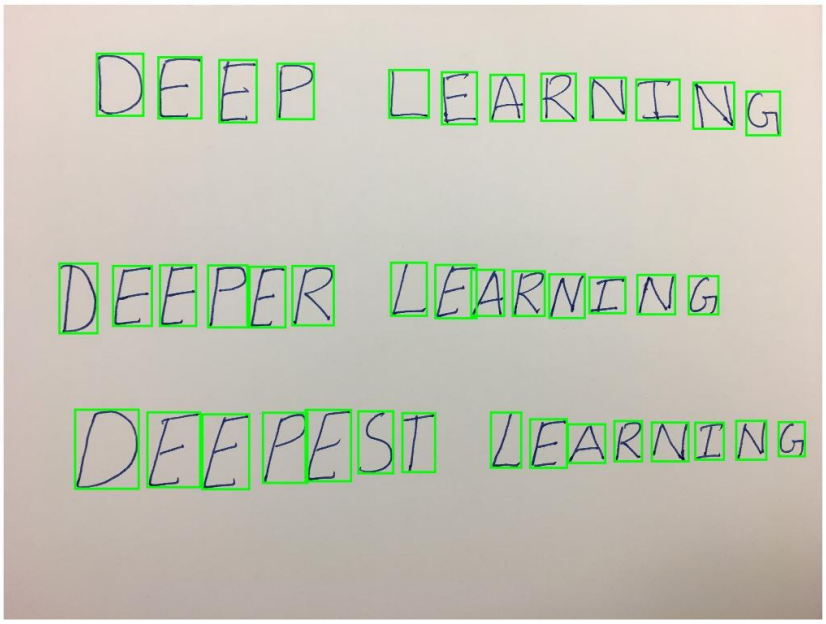
Q4.3

The images with bounding boxes are shown below:







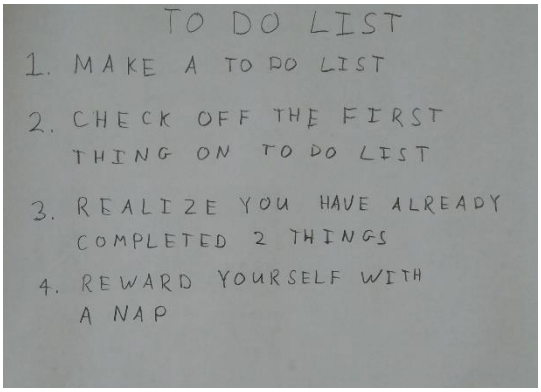


There are a few steps for preprocessing to achieve the results:

1. Convert RGB to gray if necessary, then convert gray scale into binary. The threshold is calculated by MATLAB function ‘graythresh’.
2. Apply Gaussian blur to fix some character which appears spread (some parts are separated which is likely to be treated as another character). Here the sigma for Gaussian blur I use 2.
3. Eliminate those “characters” who have too few pixels. For example, there are a few dots in image 1 which might be categorized as character by mistake. Here I use the threshold of 400 pixels.

Q4.5 *Displayed Text*

Image 1:



TCCDLIST

LMAKEATDDDLIST

ICHLCKDFETHEFIRST

THINGQNTQDOLIST

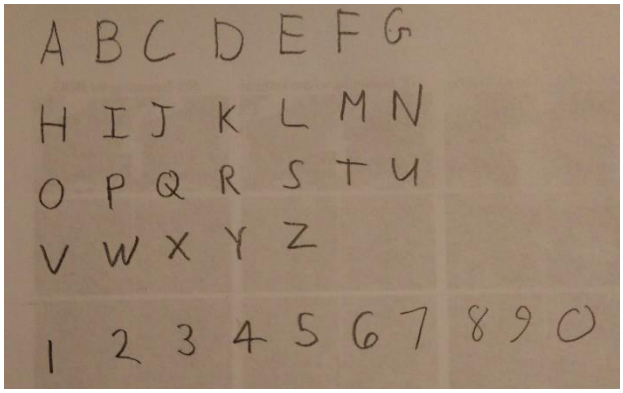
4RKALIZEYOUHAUEALRLADT

COMPLETCDZTHINGS

0REWARDYOURSELFWITH

ANAP

Image 2:



3CCCGFG

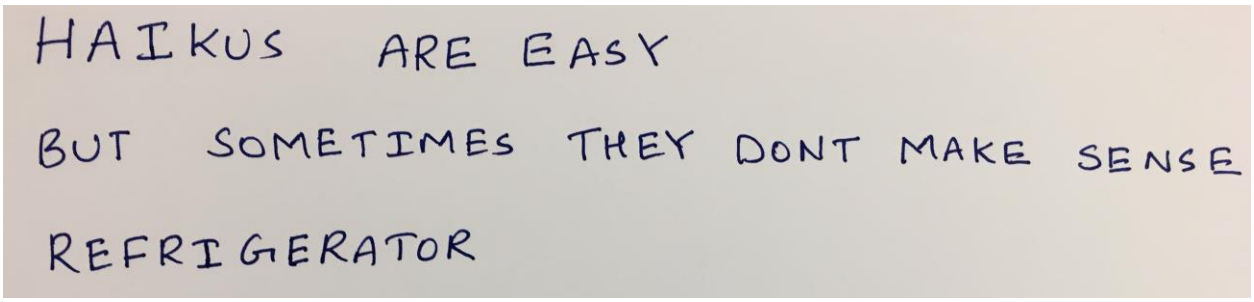
MIIKLMW

CVQKSTU

VWXYZ

BZ4GSGYXDJ

Image 3:

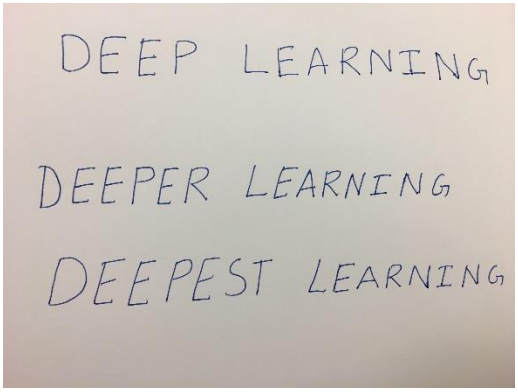


HAIKU6AREEASY

BUTSQMETIMESTHEYDDWTMAKGSENGG

REGRIGERATQR

Image 4:



CCCCCCDKYING

DCCYFKLCAKNING

CCCCCSF2EAKNING