The use of machine learning to train a neural network to achieve OCR technology

*Chenin Xu*

*Penn State University Park*

*Email:* [*cjx5021@psu.edu*](mailto:cjx5021@psu.edu)

Optical Character Recognition (OCR) is a technology that turns the characters printed on the picture into editable text by recognizing the light, shadow and color difference on the image. But the problem is that for the same character, there may be countless different images. For human, we can judge that through the experience deposited in memory, but how can the computer successfully recognize them? Obviously, we cannot create a huge database to store all possible situations. In this paper, we will focus on a technology called machine learning. Compared to other technologies, this is a technology that can continuously evolve to achieve nearly 100% accuracy through increasing input data.

**INTRODUCTION**

Machine learning is mainly divided into three parts, training data, a model for calculation and training methods for modifying the model. In essence, if we can find a suitable model and training method, and give it enough data, then all problems in the world can be solved by machine learning.

The model we use to train our OCR is a model called a neural network, a model built to imitate biological neurons. The neural network can be divided into layers, and in each layer, we will have lots of simple artificial nodes, called neurons. They relate to next layer to form a network structure. The layers in the neural network can be divided into three main categories, input layer, hidden layer, and output layer.

**INPUT LAYER**

The input layer is mainly used to receive processed data, sort them and pass them to the next layer. Here are two important concepts called batch and epoch. Batch is the amount of data used in the tuning model at a time. When our program has millions of training data, if we pass in the data one by one to train our model, then this will consume a huge amount of time, but if we just pass in all the data at once, then The training result must be far from our expectation. Therefore, it is necessary to find a suitable size of the incoming data. And the epoch refers how much times will we trainings on all training data. Basically, we divide the training data into many batches, and then perform training. After all the data is passed in once, we will disrupt the order of the data and start the second epoch.

**HIDDEN LAYER**

The hidden layer is the most important part of the neural network. The hidden layer is not a single layer, but a combination of many layers. Data enters the neural network from the input layer, goes through a series of calculations in the hidden layer, and finally passes the result to the output layer. Therefore, our main job is to find a calculation method that can get accurate results and convert the calculation results into easy-to-understand form. In order to achieve these two goals, we divide all the layers in the hidden layer into two categories, fully connected layers and activation function layers.

***3.1:Fully Connected Layer***

In machine learning, we will give each fully connected layer two independent parameters, called weight and bias. When our data is entered, we perform the following calculation (Y=output, X = input, W= weight, B=bias).

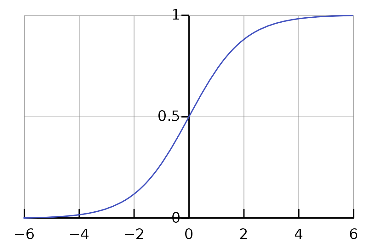
However, we usually directly pass a batch of data into the hidden layer at one time, so here we will use the matrix dot multiplication method to calculate the weight and bias which corresponding to each data.

These two parameters are our main training targets. We will continuously update them at this layer, and we will explain it later in paper.

***3.2 Activation Function Laye*r**

After each full connectivity layer, generally an activation function layer will follow. The activation function layer has two main purposes: to transform our output data into easy-to-understand data, and to add nonlinear factors to solve problems that cannot be solved by linear models.

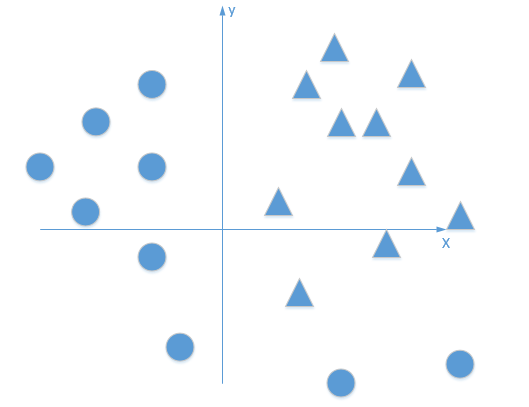
In the last layer of the hidden layer, we will use an activation function called sigmoid. Its function and image are shown below:

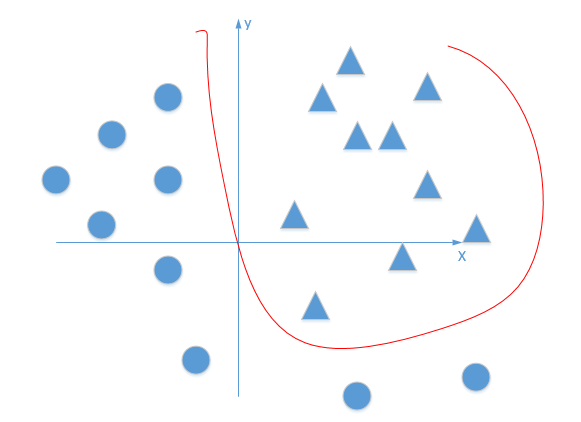


And the derivative of sigmoid function is:

Obviously, sigmoid is a function whose output range is between 0 and 1, so when our output result passes through the sigmoid layer, it will become a series of numbers in the range 0-1, which can be regarded as a percentage. Or the probability of the corresponding result.

And for the activation function layer between the two fully connected layers, their main role is to add non-linear factors. Let’s review the calculation method of the fully connected layer mentioned earlier. If you use an image to express y = w\*x +b, then it must be a straight line, but in fact many times our problems may have many nonlinear factors. For example, you cannot find a straight line to separate the circle and triangle in following picture, but by adding activation function, you can draw the curve and so the problem can be solved:





***3.3 Forward and Backward Propagation***

Forward propagation refers to the process in which the previous layer of the neural network passes the result of the calculation to the next layer. As we mentioned before, each layer in the neural network is closely connected, so the output of each layer will become the input of the next layer. Besides, each layer will store the input value for use in backward propagation.

And remember, our main task is to adjust each weight and bias to improve the accuracy, so the essence of machine learning is to update weight and bias again and again. the method we use to update those data is called gradient descent.

If we imagine all the results obtained by hidden layer as a plane composed of countless points, our task is to find the y we expect, and the precondition in our hands is the y that calculated by present weight and bias. As we adjust the parameters, y will move in all directions, and the direction that moves to the target value at the fastest speed, that is, the direction with the largest directional derivative, is our gradient.

After understanding what a gradient is, let's introduce our method to calculate the gradient vector: back propagation method. Let us first look at the following equation:. This is called the chain rule. The chain rule separates each of our layers, so we don’t need to integrate all the layers together for derivation, but to derivate separately within each layer and pass backward to get our final gradient.

**OUTPUT LAYER**

The output layer is the last layer of the neural network. It receives the calculation results of the hidden layer and gives judgments, and at this layer we will start our back propagation. We have designed two different output layers, one is loss Function layer, the other is the accuracy layer.

***4.1 Loss Function Layer***

The loss function, as the name implies, refers to the digitization of the difference between the calculation result and the correct output. In our program, the final data we get in the hidden layer will be a three-dimensional array, which contains batch size of 26\*1 lists. Each list corresponds to the possibility of inputting pictures for 26 different letters. In addition to accepting the output of the hidden layer, the loss function layer will also accept the correct result by the input layer and converts it into the same 26\*1 list with the corresponding letter position is 1, and the remaining positions are 0. Take the quadratic loss function as an example, this is a relatively common and simple loss function:

Where m is the batch size, h is the calculated result with corresponding and input data X and Y is the correct output. The main purpose of dividing by 2 here is to facilitate the calculation of the derivative during back propagation.

In the program, will be implemented as the sum of list subtraction. This is because there are some special circumstances. For example, when the program predicts that a letter is , the probability of the corresponding position of may be 99%, but the probability of may be as high as 80%. In this case, if we only use Subtract the corresponding index, our loss will become very low, but our program is not so accurate.

The loss function layer is also where we start back propagation. Still take the above quadratic loss function as an example, we can easily obtain the derivative of with respect to :

This will be propagated back as the first ring of the chain rule, and finally we will get our (gradient) in fully connect layer.

***4.2 Accuracy Layer***

In fact, in our training process, the accuracy layer is not necessary, but the accuracy layer can help us confirm the accuracy of our program. Different from the loss function layer, the accuracy layer only gets the most likely one index in the list and compares it with the correct output. In the end, we calculate the proportion of the data successfully predicted among all input data. Generally, the accuracy layer does not participate in the training of the model. We will call it in the main function with different data from the training data to verify whether our model is accurate.

class FullyConnect:  
 def \_\_init\_\_(self, l\_x, l\_y):  
 self.weight = np.random.randn(l\_y,

l\_x) / np.sqrt(l\_x)  
 self.bias = np.random.randn(l\_y, 1)  
 self.lr = 900  
 def forward(self,x):  
 self.x = x  
 self.y = np.array([np.dot(self.weight, xx)

+ self.bias for xx in x])  
 return self.y  
  
 def backward(self, d):  
 ddw = np.array([np.dot(dd, xx.T) for dd,

xx in zip(d, self.x)])  
 self.dw = np.sum(ddw, axis=0)/self.x.shape[0]  
 self.db = np.sum(d, axis=0)/self.x.shape[0]  
 self.dx = np.array([np.dot(self.weight.T, dd)

for dd in d])  
  
 self.weight -= self.lr \* self.dw  
 self.bias -= self.lr \* self.db  
  
 return self.dx

**PARAMETER UPDATE**

After we finally get the gradient, we will go back to the fully connected layer to update weight and bias. Here is another concept called learning rate. We said before that the gradient is a vector, it decides we update our parameters in which direction, but how much will we update the parameters is determined by the learning rate ().

The learning rate will affect our training speed. When the learning rate is too small, it often takes a lot of time to update the parameters again and again to get a result close to the expected value. When it is too large, it may exceed the expected value, thereby causing the total loss rise. Therefore, a proper learning rate is very important.

**PROGRAM DESIGN AND CODE**

***6.1 Data Preprocessing***

After we elaborated on the forward and backward propagation and the characteristics of each layer, and before we began to build our program. There is another function that needs to be introduced, which is called data preprocessing. Its main role is to convert pictures into a list of numbers that the computer can recognize.

First, the pictures we used this time are all pixel pictures composed of black and white like below:

So, in the data preprocessing function, we simply turn all the pictures into a matrix composed of 1 and 0, where 1 represents white and 0 represents black. After the image is converted, this layer will also accept an expected output value and store it in an .npy file for easy reading.

***6.2 Other layers***

Even if we do not need to continue forward propagation in the loss function layer, and in the same way, we do not need to perform back propagation in the original fully connected layer, but in program design, we will give each layer forward propagation And the ability of backpropagation to facilitate the unified call in the main function. Let's take the fully connected layer as an example to show our program design:

The code shows the initial random spreading of points and the changes to each parameter of the full-connected layer. The activation function layer is also the same: in the forward propagation, the data from the previous layer is accepted, and in the back propagation, derivative is obtained and passed to the previous layer.

***6.3 Deep learning***

First, let us build the simplest neural network, which only contains one fully connected layer and one sigmoid layer. When we used it for training, we found that the accuracy rate did not increase when it reached 85. The program reached an accuracy rate of 84 at 20 epochs, but after 100 epochs, it still has an accuracy rate of 85.1.

Here we introduce the concept of deep learning. In this paper, deep learning can be simply understood as a neural network with two or more fully connected layers.

inner\_layers = []  
inner\_layers.append(FullyConnect)  
inner\_layers.append(Sigmoid)  
inner\_layers.append(FullyConnect)   
inner\_layers.append(Sigmoid())  
loss\_layer = QuadraticLoss()

After using the deep learning method, the training effect of our neural network has been significantly improved. After 100 epochs, accuracy rate has risen to around 97.5. However, this is still far from the accuracy rate we are pursuing. By observing the accuracy rate after each epoch, we will find that the increase in accuracy rate is very small. We will explain this problem in the next chapter.

**DEBUGGING AND OPTIMIZATION**

***7.1 Hyperparameter***

Hyperparameters refer to the parameters manually set by the programmer, including the learning rate, batch size, epoch and how much layers in hidden layer. By modifying hyperparameters, we can speed up our training or get higher accuracy. But at the same time, we must consider time and memory costs.

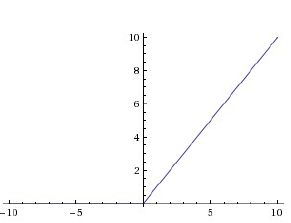
***7.2 Vanishing Gradient***

In the previous deep learning, we mentioned the problem of slow increase in accuracy. The main reason for this problem is the vanishing gradient. Vanishing gradient, or unstable gradient, is the main problem we encounter in the process of debugging the program. When adding more than two fully connected layers and sigmoid layers to the hidden layer, we will find that our accuracy rate rises very slowly. By printing the input and output results of each layer, we found that the gradient propagated back is a very small value.

The main reason for this problem is that when we use sigmoid as the activation function between two fully connected layers, our value will be normalized to between 0-1. Looking back at the image of sigmoid, we will find that for the sigmoid function, when the input value range is between -3 to 3, the function changes greatly, and when the input value is less than -3 or greater than 3, the function becomes insensitive (infinitely close to 0 or 1). In our initial random spotting process, our parameters are quite inaccurate, and this causes our final gradient becomes very small after a series of calculations.

***7.3 Other Activation Function***

One way to solve vanishing gradient is to use other activation function. Here is one sample of other activation function called Relu:



Relu is currently one of the most used activation functions, it is easy to calculate and easy derivation. Its derivative is:

Obviously, if Relu is used, the gradient is either 0 or remains unchanged during back propagation, so the reduction of the gradient is very small. Even if there are many layers in the hidden layer, the convergence speed of the previous layer will not be very slow.

***7.4 Cross-Entropy Loss Function***

Another way to solve the gradient descent problem is to use the cross-entropy loss function instead of the quadratic loss function. The equation of the cross-entropy loss function is as follows:

We can find when y = h = 0, the result is positive infinity, that means our model's prediction at this time is completely wrong. And, when y = h =1, the result is 0, that means the prediction is correct.

The derivative of the cross entropy loss function is as follows：

We will find that the denominator of its derivative is exactly same as the derivative of our sigmoid layer. Therefore, when back propagation is performed, those two values will be reduced, and our vanishing gradient problem is will be solved.

After using the cross-entropy loss function to replace the quadratic loss function, the accuracy rate reached 99 or more after 100 epochs.

**DEMO AND FUTURE WORK**

***8.1 More About Data Preprocessing***

Earlier we have discussed data preprocessing, which is the process of converting pictures into computer-recognizable array. However, in practical applications, pictures are not always in pixel grayscale format, so we need additional function for preprocessing in our main program.

First, for a larger none grayscale image (like 800\*800 pixel RGB image), we need to resize it to a pixel format. Here we use OpenCV to achieve this. Code example are below:

img = cv2.imread(path, 0)  
h, w = img.shape  
count = 0  
while h > 34 and w > 34:  
 h = h / 2  
 w = w / 2  
 h = int(h)  
 w = int(w)  
 img = cv2.resize(img, (h, w))

img = cv2.resize(img, (17, 17))

OpenCV allows the program to open the image in gray scale, and when the picture’s height and width is large, we can half it by while loop. After that, just resize the image to , then we will get an ideal matrix. Attention here is the format of the matrix is not what we can pass to our neural network. We need an additional step to change the matrix to ndarray format, here we use np.asarray.

However, if we just simply resize the image, the graph may be unrecognizable. Because for the whole picture, the proportion of letters is very small, so in the process of shrinking, our subject become ignorable part. There for we need another dilation function to emphasize the character part. Fortunately, in OpenCV, we already have such a function. Its operating principle is to expand the original pixel to the set size. For example, the setting size is , after calling this function, the matrix will change like below:

Through debugging, we found that if we call this function every time, we resize the picture to half the size, the final picture will become very ambiguous, so in the main program, we will perform the dilation after every two reductions.

***8.1 Demo and Future Work***

After completing all the basic functions of the program, we can start the demo. Here I chose two different pictures, the first is a black and white picture, and the second is a black background with blue text.





As shown in the figure, our program correctly recognized the drawn letters. But during the test, I found that the accuracy of the program is not so ideal. Due to the limitation of training data, the recognition of hand-drawn images is not very high. Since the train picture is almost same form with some different interference point. If the hand-draw picture is far with the graph given by the train data, the result will not be correct. This problem can be solved by increasing test data. In future work, we can make our program more generalized by expanding the range of training data.

In addition, there is room for improvement in data preprocessing. Now our data preprocessing function simply do the dilation after every two reductions, but when the font of the input picture itself is large, this operation will still make our subject ambiguous. Here we can alternately use erosion and dilation to make the image have a higher degree of recognition.

And, we can generalize our program by adding a convolutional layer and a pooling layer, so that it is not limited to the recognition of single letters but can recognize a paragraph of letters.

However, the field of neural networks is a very extensive and esoteric field, and there will be no further discussion in this project.