**Recognizing OCR English Alphabet by building Neural Network**

Mingyuan Sun

Vanke Meisha Academy

Yuan Xu

August 30, 2021

**Abstract:**

The present study in Machine learning is being well-developed. This paper is using Stochastic Gradient Descent (SGD) algorithm to build a Neural Network (NN) that could identify Optical Characters with 99% of accuracy. The Neural Network has three inner layers, which are fullyConnect layer, activation layer, loss layer. To show the accuracy of the NN, we create the Activation layer. To train the model, we use 60000 photos with their labels. 40000 photos are used in training the NN, 10000 photos are used in validating the NN, and 10000 photos are used in testing the NN.

*Keywords: Machine learning, Neural network, Stochastic gradient descent, sigmoid, fully connect, python, NumPy*

**Introduction:**

With the development of technology, more jobs could be done using machine learning. In this paper, we build a Neural Network (NN) model to recognize English optical characters with 99% accuracy.

* 1. **Neural Network: a model of machine learning.**

To implement machine learning, we use a typical model: the Neural network (NN). The basic linear model of NN is made by three main layers: which are the input layer (a1, a2, and a3 in the graph), the network layer (b1 and b2 in the graph), and the output layer (h1 and h2 in the layer).

图示

描述已自动生成

Graph [1-1]: a simple illustration of neural network

In Order to get b1 and b2, we need to multiply the weights to corresponding inputs and then add a bias to the result. In math, we could write the process to calculate b1 and b2 as matrix calculation:

示意图

中度可信度描述已自动生成

Graph [1-2]: the matrix calculation

Theoretically, if we could get perfect weight and bias then the b1 and b2 would have a perfect output. So, we could say that the process of achieving the best output is the process of founding the best bias and weight. Theoretically, if we could get perfect weight and bias then the b1 and b2 would have a perfect output. So, we could say that the process of achieving the best output is the process of founding the best bias and weight. Since we have the expected result b1 and b2, we can compare it to the result get from the neural network to get a loss, which is also known as the lost layer, and use the loss to revise our neural network. Due to our result is linear, we need to use an activation layer to make the result become a non-linear one.

* 1. **Loss function**

There are two important roles to follow when training the NN, which are using **a random** number to initialize neural network parameters and using **loss function** to use the initialized neural network.

To start with, the loss function is the function describing the difference between NN's output and the expected output. In this case, we use quadratic loss function as our loss function, which has a math formula:



Graph [1-3]: math formula of the loss function

Theta stands for variables that needed to be tuned insides our NN, X is the input of the NN, and Y is the expected result. By squaring the loss, we could get a greater loss which would be more obvious for people to see how the NN is working.

The smaller the output of the loss function is, the better the NN is, and in a perfect condition, the loss function would have an output of zero. So, our goal is to make the loss function's as small as possible.

* 1. **gradients descend**

To achieve the goal of making the loss function to its lowest value, we could use an algorithm called gradients descend. For people to understand the concept more clearly, we would use an example here. Imagine a person is trying to go to the lowest point from the mountain, and he can’t see where the lowest point is, the only information he could know is that how steep the slope is from point A toward point B. According to the graph [1-4]:



Graph [1-4]: a graph showing that how the purple sphere gets to the lowest point

When the people achieve the minimum point, he won't move anymore, because he could notice that if he takes another step, he is getting upward. However, this algorithm has two problems. The first one is that he may only achieve the local minimum point he would stop there without getting the global minimum point. The second one is how to select the step length while doing gradient descent.

图片包含 折线图

描述已自动生成

Graph [1-5]: a graph showing the result of low learning rate.

图片包含 多边形

描述已自动生成

Graph [1-6]: a graph showing the result of high learning rate

According to the graph [1-6], if his step is too large, he may miss the minimum point. However, if he takes an extremely small step, it will take him much more time to get to the lowest point. This problem could be solved by providing a suitable step length图片包含 图形用户界面

描述已自动生成

Graph [1-7]: the formula used in calculating gradient

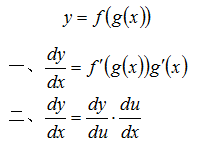
To illustrate, we give an equation, as shown above, DJ/D theta stands for the slope, alpha stands for the step length.

* 1. **backpropagation algorithm**

After knowing how to use gradient descend to minimize the loss function, here

comes another question: how could we calculate D J/ D theta?

According to the chain rule in calculating derivative, we could get the derivative of a complex function by Multiplying the derivatives consecutively,



Graph [1-8]: the graph showing chain rule

To illustrate the idea of back propagation algorithm, let us use S**igmoid function** as the example. The origin function of Sigmoid is . To find the derivative of it, we need to decompose the function. First, we define function

A = x, B = -A, C = , D = 1, E = D + C, F = 1/E = .

According to the chain rule we mentioned above, we could calculate D F / D A by using the chain rule, which would be

The process of decomposing the function is called **Forward** calculation, which starts from A to F; the process of calculating the derivative of the function is called **Backward** calculation**.**

* 1. **epoch and batch**

During the training process, we used 40000 images. We divided them into 40 batches, so each batch contains 100 images. batch size is also an important hyperparameter, for example, if the batch size is too large it would bring a negative effect while using ReLu as the activation layer. If the batch size is too small, the Stochastic Gradient Descent algorithm would not work as intended and the calculation time would also increase. However, 40000 photo is not enough for the neural network to get an accuracy of 99%, so we shuffle those photos again to train the neural network after all batch is finished, this is also known as an **epoch.**

**2.1 data preloading**

In this neural network, we have 60000 photos with a .txt file that include the correct optical character for each photo. We would put them into three groups: train, test, and validate. The training group includes photo 0~39999, the test group include photo 40000~49999, and the last 10000 images are in the validate group. The neural network uses train group data to train and revise the model, then uses test group data to test the trained model’s accuracy. However, with the ongoing training process, the machine would remember the photo with its label but not learning how to recognize the photo because those photos are being used again and again, which is also known as **overfitting.** To determine if the model is overfitting, we could use the validate group to test the model.

For the convenience and speed of the following program, we pack all photos and their correct labels into a numpy file.

def main(src, dst):

with open(src, 'r') as f:

list = f.readlines()

data = []

labels = []

for i in list:

name, label = i.strip('\n').split(' ')

print(name + ' processed')

img = misc.imread(name)

img = img/255

img.resize((img.size, 1))

data.append(img)

labels.append(int(label))

print('write to npy')

np.save(dst, [data, labels])

print('completed')

The code above provides a function called “main”, it takes the input location and the output location the NumPy file would be stored. The code above reads in images (all 17 \* 17 pixels) and resizes them into a 289\*1 matrix, and packs them with the correct label, and stores them into the file.

**2.2 Data layer**

**2.2.1 init()**

After we pack all those photos and their label into a .npy file, we could use them in our code. We would store the photo in an array and its label in another array. According to 1.5, we have a hyperparameter called batch size, so we intake a parameter to assign a value for it. We create an integer that records the length of x and a variable to record the position of the photo are being read.

def \_\_init\_\_(self, name, batch\_size):

with open(name, 'rb') as f:

data = np.load(f, allow\_pickle=True)

self.x = data[0]

self.y = data[1]

self.l = len(self.x)

self.batch\_size = batch\_size

self.pos = 0

**2.2.2 forward()**

According to 1.5, we would shuffle and run those photos again after the last batch of the epoch is ran. We could use the position, the length, and the batch size to determine whether this is the last batch of the epoch. When the current position plus the batch size is larger or equals to the length, we know this would be the last batch of the epoch. For the next epoch, we shuffle photos and their labels by given them a random index.

def forward(self):

pos = self.pos

bat = self.batch\_size

l = self.l

if pos + bat >= l:

ret = (self.x[pos:l], self.y[pos:l])

self.pos = 0

index = range(l)

np.random.shuffle(list(index))

self.x = self.x[index]

self.y = self.y[index]

else:

ret = (self.x[pos:pos + bat], self.y[pos:pos + bat])

self.pos += self.batch\_size

return ret, self.pos

**2.2.3 backward ()**

The data layer should be the first layer, so it does not have a backward function.

**2.3 fully connected layer**

**2.3.1 init()**

In 1.2, we discuss that we need to **randomize** neural network parameters, and in 1.1, we mentioned that the neural network is trying to get the perfect **weights** and **bias.** At the beginning, we assign weights and biases as random numbers in the constructor of class FullyConnected. Meanwhile, we mentioned there is a hyperparameter called alpha which is the step length in 1.3, so the constructor of FullyConnect would be:

def \_\_init\_\_(self, l\_x, l\_y, learningRate):

self.weight = np.random.randn(l\_y, l\_x) / np.sqrt(l\_x)#the reason for np.sqrt(l\_x) would be explain later

self.bias = np.random.randn(l\_y,1)

self.learningRate = learningRate

**2.3.2 forward()**

According to 1.4, FullConnect layer would have a forward () method and a backward() method. To start with, the forward function would have an input as its parameter. Due to fully connected is the layer after Data layer, its input would be a 3d array [batch size][289][1]. To take out each photo from the batch, we use a for loop to iterate the array. The forward function would store the input as a global variable for the backward function. According to 1.1, we have the formula to calculate the output b1and b2, so we rewrite it into code:

def forward(self, input):

self.input = input

output = np.array([np.dot(self.weight, xx)+self.bias for xx in input])

return output

**2.3.3 backward()**

According to 1.4, In the backward function, we need to calculate is the derivative of this layer and then time the input from the backward function, which is the previous layer's derivatives. According to the stochastic gradient descend algorithm, we add all derivatives together and calculate the average of them. To implement the gradient descend algorithm, we need to update weight and bias by minus their derivative.

ddw = [np.dot(dd, input.T) for input, dd in zip(self.input, d)]

dweight = np.sum(ddw, axis = 0) / d.shape[0]

dbias = np.sum(d, axis = 0) / d.shape[0]

dx = np.array([np.dot(self.weight.T, dd) for dd in d])

self.weight = self.weight - self.learningRate \* dweight

self.bias = self.bias - self. learningRate \* dbias

return dx

**2.4 activation layer: sigmoid**

The activation layer here is used to Introduce the nonlinear part into our NN.

**2.4.1 init()**

The activation layer does not involve any private variables.

**2.4.2 sigmoid()**

For the convenience of the following code, we define sigmoid as a function.

According to the mathematical formula, we could write it as:

def sigmoid(self, x):

return 1 / (1 + np.exp(-x))

**2.4.3 forward()**

The forward method is simple here. All we need is to store the input for backward use and put the input into sigmoid and return it.

def forward(self, x):

self.x = x

self.y = self.sigmoid(x)

return self.y

**2.4.4 backward()**

According to formula of sigmoid derivative, we could write the backward as:

def backward(self, d):

sig = self.sigmoid(self.x)

self.dx = d \* sig \* (1 - sig)

return self.dx

**2.5 quadratic loss function**

**2.5.1 init()**

Loss function does not involve any private variables

**2.5.2 forward()**

To calculate the loss, we need to have the final output from the model and the corresponding labels, so it would have two parameters. The input of the forward function would be a 1024\*26\*1’s array, so we need to take out each photo’s output, which is a 26\*1 array. Then we create a new 26\*1 array and assign all numbers in it to zero, except the number with the index of the correct label; we assign it with 1. After we have the array, we use the array to minus the array that we input (which is the output of the neural network) then square it to get the loss and get the average of 1024(batch size) output.

def forward(self, x, label):

self.x = x

self.label = np.zeros\_like(x)

for a,b in zip(self.label, label):

a[b]=1.0

loss = np.sum(np.square(self.x-self.label))/self.x.shape[0]/2

return loss

**2.5.3 backward()**

In the backward function, we get the derivative of the loss function the return it.

def backward(self):

self.dx = (self.x-self.label)/self.x.shape[0]

return self.dx

**2.6 Accuracy Layer**

The neural network's main code already ends; this layer is used to tell us the accuracy of the neural network.

**2.6.1 init()**

There is no private variable need to be used in the accuracy layer.

**2.6.2 forward()**

We use the test.npy to test the average accuracy of the neural network.

def forward(self, x, label):

self.accuracy = np.sum([np.argmax(xx) == ll for xx, ll in zip (x, label)])

self.accuracy = self.accuracy\*1.0/x.shape[0]

return self.accuracy

**2.6.3 backward()**

There is no need for the backward function in this layer.

**2.7 main function**

We have all components to build a neural network, the main layer is to put all puzzle together.

def main():

datalyer1 = Data("C:\\Users\\13794\\Desktop\\python\\train.npy", 1024)

datalyer2 = Data("C:\\Users\\13794\\Desktop\\python\\validate.npy", 10000)

innerlyer=[]

innerlyer.append(FullyConnected(17\*17, 26, 1000))

innerlyer.append(Sigmoid())

losslyer = QuadraticLoss()

accu = Accuracy()

epoch = 150

for i in range(epoch):

print("epoch: ", i)

lossSum=0

iters = 0

while True:

data, pos = datalyer1.forward()

x, label = data

for layer in innerlyer:

x = layer.forward(x)

loss = losslyer.forward(x, label)

lossSum += loss

iters += 1

d = losslyer.backward()

for layer in innerlyer[::-1]:

d = layer.backward(d)

if pos == 0:

data, \_ = datalyer2.forward()

x, label = data

for layer in innerlyer:

x = layer.forward(x)

# print(label)

accur = accu.forward(x, label)

print("accuracy: ", accur\*100, "%")

print("loss: ", lossSum/iters)

break

1. **revising neural network**
   1. **single fully-connect layer neural network**

at first, we try single fully connected network and get 92.67% accuracy in 150 epochs with 0.0425 loss. In the third epoch it achieves 60.20% accuracy. However, to achieve our target, which is 99% accuracy, we need to make some changes

* 1. **multiple fully-connect layer neural network:**

The first action we do is to add one fully-connect layer.

innerlyer.append(FullyConnected(17\*17, 26, 1000))

innerlyer.append(Sigmoid())

innerlyer.append(FullyConnected(26, 26, 1000))

innerlyer.append(Sigmoid())

by adding one fully connect layer, the accuracy after 150 epochs is 97.6%. However, it takes 15 epochs for the neural network to get to 60% accuracy. When we add the third layer, the final accuracy is 97.97 %, which does not improve much. This could be caused by **gradient vanishing** which we could talk about later. After serval experiment, we found out using 3 layers are the most efficient way.

* 1. **adjusting parameter**

There are a lot of hyperparameters that could affect the neural network. The first one I change is the learning rate. By testing 1000, 2000, 4000, 2500, 3500, et cetera, I finally find out 3000 would be the best learning rate with 98.39 % accuracy. The second hyperparameter I change is the output of the inner neural network. I've changed it to 13, 20, 42, et cetera. Finally, get an accuracy of 98.71%

I still tried a lots hyperparameter to revise the neural network, but they don’t perform well. I have changed the batch size into 500, 300, 2000, et cetera. Finally, I found out the best batch size is 1000. The best result of changing only the parameter would be 98.71%.

* 1. **gradient vanishing: solutions**

the accuracy is getting slower and slower while the number of fully connected layers is higher, even stopped growing at 4.12%, why would this happen? Before talking about gradient vanishing, I want to introduce the sigmoid function’s derivative function

图表

描述已自动生成

Graph [3-1]: the graph of derivative of sigmoid function

As we can notice here, when the number is getting larger than 4 or smaller than -4, the sensitivity of the function would be decreased, which is, when x = - infinite y = 0, x = -infinite + 1 y still equals to zero. So, when the x given is small enough or large enough, the function would provide a very close y similar to before, and that’s the reason why the accuracy would not grow up after using too much sigmoid, which is also known as gradient vanishing.

There are many solutions to this problem, like using the ReLu function as the activation layer.

图表, 折线图

描述已自动生成

Graph [3-2]: graph of ReLU function and it’s derivative

However, ReLu has a fatal problem that is when the bias of the neural network is too small, so the input of ReLu is always negative and would result in zero. If all input for ReLu is zero, the weight and bias can't be refined. This problem is also known as the dying ReLU problem. To solve this, we would replace Relu with LeakyRelu

图示

描述已自动生成

Graph [3-3]: the graph of LeakyReLU function

There is also another solving method. This method is called Cross-Entropy Loss. The formula of this function would like is

图片包含 图示

描述已自动生成

Graph [3-4]: the formula of Cross-Entropy loss function

The main reason why we use is its derivative is

图片包含 形状

描述已自动生成

Graph [3-5]: the formula of the derivative of Cross-Entropy loss function

Which would become h (theta, x) after it times the sigmoid function's derivative.

I use a simpler way to solve this problem, I removed one sigmoid layer in the inner layer because we know that two layers fully connect layer with two sigmoid layers would not affect a lot by the gradient vanishing problem. After reduced one sigmoid layer, I get an accuracy of 99.14 % with 0.0057 loss.

**Work cited:**

Quick guide to gradient descent and it's variants. Quick Guide to Gradient Descent and It's Variants. (n.d.). Retrieved September 28, 2021, from https://morioh.com/p/15c995420be6.