Introduction to machine learning with a simple OCR system on uppercase English characters

*Ning Zhu*

*University of Nottingham, Ningbo*

*Email: scynz1@nottingham.edu.cn*

***Abstract* – As the rapid development of AI, machine learning**

**has been a basic skill almost all CS students need to learn. OCR system on English characters as a typical example of machine learning, can help students command a basic understanding of the machine learning. This paper will focus on how image preprocessing and neural networking combined could build up a simple OCR system with relatively high accuracy, show the basic steps in machine learning.**

1. **Introduction**

AI has been a critical part in human-life.They are nearly everwhere-vending machine in school buildings, cameras in the road, the famous AlphaGo in weiqi etc. In order to keep up with the rapid development of computer technology, CS students start with a simple OCR system which provides a basic understanding of machine learning with students.

This paper will focus on the process of building & training a neural network, as well as the preprocessing of data. In machine learning, a set of data is given and a trained model is created to predict the results. The neural network model proved to be effective in completing such task.

Neural network is a computing model with connected nodes, and its hierarchical structure is similar to that of neural network in brain. Neural network can ”learn“ from data, therefore it can train its pattern recognition, classify data and predict future events. Neural networks subdivide your input into multiple abstraction layers. For example, a large number of examples can be used to train whether the recognition mode is voice or image, just like the behavior of human brain. The behavior of neural network is determined by the connection mode of its elements and the strength or weight of these connections. During the training, the system will automatically adjust the relevant weights according to the specified learning rules until the neural network performs the required tasks normally. In genral, there are basically three parts in a neural network, an ***input layer***, many ***hidden layers*** and an ***output layer***.

The input layer obtains the training data and the expected output values from the file, which are regarded as ***data*** and ***label*** respectively. Then sends the processed ***data*** to the hidden layers.

The hidden layers are the main layers that process ***data*** and predict the results. Basically, for the layers in the hidden layers, the output y has a linear relationship with the input x. The relationship is done as shown below:

Where is the weight and is the bias.

Usually during coding, data flow is passed through layer by layer. The matrix of input, dotted with a matrix of weight and then added to a matrix of bias, will generate the matrix of output. The output matrix is then passed to the next layer as input ”x“. The process is as follows:

Usually, an ***activation layer*** is considered as an output layer. It normalizes the predictions and converts them into floating-point decimals between 0 and 1 which could be treated as probabilities. The ***Sigmoid Function*** is a very commonly used activation function:

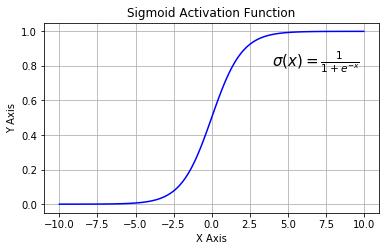


Figure 1: sigmoid function

Its value ranges from 0 to 1, and therefore is one of the optimal activation functions in the industry.

This whole process from input to output is known as ***forwarding***, the ***backwarding*** process which is used to modify the model will be introduced below.

In the ***backwarding*** process, there are two main concepts: ***cost function*** and ***backpropagation***.

***Loss*** is used to reflect the difference between the actual output value of the model and the expected correct output value. ***Loss*** also reflects the equality of a model. ***Cost function*** are used to caculate the ***loss.*** The ***cost function*** takes the weights, biases, input ***data*** and the ***label*** as its inputs. A common ***cost function*** used in machine-learning is the ***Quadratic Cost Function***, is shown as below:

Where is the parameters across the neurons (namely the weights and biases), is the corresponding***data*** and is the ***label***. In genral, the square of the difference between the actual output value and the expected output value.

***Backpropagation*** is essentially an algorithm that allows programs to perform ***gradient descent***：

IMG_256

Where is the parameters across the neurons (namely the weights and biases), is the current gradient, α is step lengrh we set.

The original meaning of gradient is a vector (vector), which means that the directional derivative of a function at this point gets the maximum value along this direction, that is, the function at this point changes the fastest along this direction (the direction of the gradient) and the rate of change is the maximum (the modulus of the gradient).

Through it, the machine-learning system modifies the values of the weights and biases in the hidden layers so as to decrease the ***loss***, and in turn to get a higher equality of the model.

To help you better understand how ***graduent descent*** being achieved in machine learning, specify steps will be introduced below.

After forwarding, the ***output layer*** takes the derivative of the cost function, plugs in its data and passes it to the last layer of the ***hidden layers***.

Note that the ***backpropagation algorithm*** works by the ***chain rule***:

***Backpropagation*** will stop at the first layer of the hidden layers, and get the gradient of current model. In that case, we just need to set a appropriate hyper-parameter α to represent the step and use the step and gradient to update weights, biases.

Given that the mean value of the sigmoid function is not 0. In the real use, the network should be trained with batch of input items, which means that 1000 or more pieces of items are parsed into the system at the same time. An epoch means that we have completed a ***forward propagation*** of all the data and updated the weights and biases with ***back propagation***.

An ***accuracy layer*** can also be added to directly repersent how accurate the prediction is (***Loss*** is not intuitive). It compares the predictions with the correct labels and quantizes the assessment. Using boolean values as results, the accuracy layer sums up the results and takes the average, and then prints it out for users to decide whether the neural network is good enough for application or not.

1. **Materials and Methodology**

For this task, Python 3 is the first-choice programming language. Its intuitive syntax and extensive machine-learning libraries makes coding more efficient. The following libraries are used:

* PIL
* Numpy

1. ***Calibration & Training***

The first thing is to acquire training datas. For this project, preprocessed datas are downloaded from [machine-learning data repositories.](https://github.com/znzz1/Machine-learning) There are three data sets. Training set is used to train the model, validation set is used to get the accuracy of the trained model and testing set is used to evaluate the generalization ability of the final model. Here we use the training set to train the model.

Through processing, the 17 \* 17 pixel files are converted into NPY files for training the model.

The second step is to construct a simple network, which consists of a ***fullyConnected layer*** as ***hidden layers*** and a ***sigmoid layer*** as ***output layer***.

For the ***input layer***, every time the last batch is read in, the original input list gets shuffled around to make the order of the original input data seems more random. Note that the input layer does not do anything in the backward function since it is the ***input*** ***layer*** which only reserves the datas.

**Class InputLayer**

Def Init(Source, Batch\_Size):

Load datas and save them

Get datas’ length and save it

Save Batch\_Size

Set pos to 0

Def Forward():

If pos + Batch\_Size >= length

Take all remaining datas

Set pos to 0

Shuffle data list

Else

Get a batch of data

Updata pos

Return data and pos

Def Backward():

Pass

For the ***hidden layer***, using ***weights*** and ***biases*** (the current model) to calculate the input data from the ***input layer***. Note that the calculation here is linear.

In this OCR system, a FullyConnected layer consists of 26 neurons which correspond to 26 uppercase English characters respectively.

**Class FullyConnectedLayer**

Def Init(l\_x,l\_y):

Initialize the weights and biases randomly

The size of weights is (l\_y,l\_x)

The size of biases is (l\_y, 1)

Here l\_y is the neurons of the fullyConnected layer and l\_x is the sizeof the input data

Def Forward(input data):

New data = input data \* weights + biases

Return New data

Def Backward(derivative):

Calculate the gradient by chain rule

Update weights and biases

Return New derivative(Can be ignored)

For the ***output layer***, using the ***sigmoid layer*** as the ***activation layer***. If you want, you can get the output at this layer, but during training, it makes no sense. Note that the size of the data has not been changed in this and following layers.

To get the ***loss*** of the model, we add a new layer after ***output layer*** called ***Quadratic loss layer*** and use the method of ***Quadratic Cost Function***.

**Class SigmoidLayer**

Def Init():

pass

Def Forward(input data):

New data = sigmoid(input data)

Return New data

Def Backward(derivative):

Return Derivative◊Sigmoid(Data)’

**Class *QuadraticLossLayer***

Def Init():

Pass

Def Forward(Data, Label):

Labels are the expected values

Loss = (Data – Label)2 / Data.Count

Return Loss

Def Backward():

Derivative = 2(Data - Label)/Data.Count

Return Derivative

In order to show the performance of the model more intuitively, I added an ***accuracy layer*** at the end. Here I use the trained model and the validation set as the input data.

Finally, use the main function to train the model through the constructed model:

**Main function to train the model**

DataLayer1 = InputLayer(Training Data, 1024)

DataLayer2 = InputLayer(Validation Data, 10000)

FCL = FullyConnectedLayer(17\*17,26)

SigLayer = ActivationLayer()

QuadLoss = LossLayer()

Accuracy = AccuracyLayer()

If Weights.CSV and Biases.CSV Exists

Load CSVs into FCL

Set Learning Rate

Set epochs

For i in Epochs

While True

Data, Labels = DataLayer1.Forward()

Forward all Layers with Data, Labels

Calculate Loss Sum

D = LossLayer.Backward()

Backpropagate through Layers with D

If Position = 0

DataLayer2.Forward()

Output Average Loss

Output Accuracy.Forward()

break

Save Weights.CSV

Save Biases.CSV

**Class AccuracyLayer**

Def Init()

Pass

Def Forward(Data, Label)

For every Data, the biggest number in 26 neurons is the output.

If output = Label, acc += 1

Return acc/Data.Count \* 100%

The last step is to train the network.

The neural network is trained to have an accuracy of roughly 92% after about 500 epochs. It is possible sometimes for a network to reach a limit of only 86% or 88% accuracy, since initialization is random, the lowest gradient might only be local minimum.

1. ***Basic Application***

Once the neural network is trained to have a relatively high accuracy, we can proceed to write the OCR system using the weights and biases.

The program ask the user for the path of the image. Then, use PIL to open the image and convert the 17\*17 image to an array. Note that the OCR system only recept 17\*17 files. Meanwhile, the program load the weights and biases in to build the model. Finally, use the model to predict the uppercase English character and print the result to the user.

**OCR system**

Load weights, biases

Take user’s input and open the image

Convert the image

Use model to predict the result

Print the result

The application works fine for the testing data in the downloaded data sets. For self-drawn images in the drawing tool that comes with windows system, the application also works most time, but the accuracy has a certain degree of decline. The issues will be discussed in the next minor section.

1. ***Issues & Debugging***

The main reason is the network is too simple. The network can not accurately capture the characteristics of the letters. Because our training pictures(Figure 2) are all standardized letters, it has little effect on the test set. However, when we use hand-written image (Figure 3), it becomes very hard for the system to recognize the character(My system output E for Figure 3).

C5

Figure 2. “C” in test set Figure 3. Hand-written “C” in drawing tool

To ensure the accuracy for the hand-written image, we can use two or more ***FullyConnected layer*** in the ***input layer***, which is called ***deep learning***. ***Deep learning*** can make the model have higher accuracy and generalization ability. The model of ***deep learning*** can better grasp the characteristics of letters and in that case reduce the error caused by writing nonstandard.

Therefore we take two ***FullyConnected layer*** in the ***input layer***. Note that because of the properties of the sigmoid function, it is very likely to cause ***gradient vanish***. Use the derivative of sigmoid function(Figure 4) to explain that.

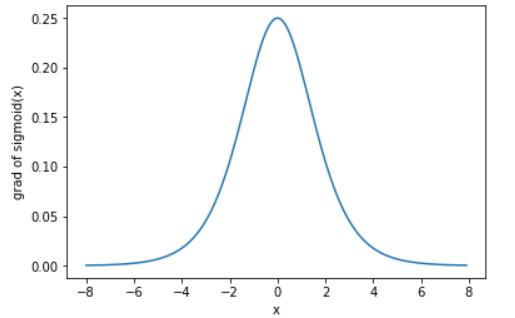


Figure 4: the derivative of sigmoid function

From the graph, we can see that when the value x exceeds 6, the derivative decreases very slowly, which means that there is no difference between the speed of decreasing from 100 and that of decreasing from 8. When the weighrs or biases are large, the derivative decreases very slowly and needs many epochs.

To deal with this, we use ***Cross entropy loss function*** to replace the ***Quadratic Cost Function***. The ***Cross entropy loss function*** is designed to neutralize the ***gradient vanish*** caused by sigmoid function.

1. **Results**

**Class *CrossEntropyLossLayer***

Def Init():

Pass

Def Forward(Data, Label):

Labels are the expected values

Loss = ((–Label\*log(Data))-((1-label)\*log(1-Data)))/ Data.Count

Return Loss

Def Backward():

Derivative = (Data-Label)/Data/(1-Data)/Data.Count

Return Derivative

The final program was trained to have an accuracy of 99.3%.

For those hand-written characters, the improved system keeps a high accuracy to recognize them.

First, test with a image in test set:

**35FS3}DQWH}C}3%5I5@E{KMCL**

Figure 6: the result of my system

Figure 5: the graph of “N”

The system works perfectly.

HFinally, test with a hand written “H”:

`5Q}JFYP0ZNEEF56%]3E[%U

Figure 8: the result of my system

Figure 7: the graph of “H”

As can be seen from the figures, the results are all satisfactory. This concludes the beta stage of the OCR project.

1. **Discussion & Future Work**

Sigmoid function is one of the activation function. However, it has been proved that ***ReLu*** function is better than sigmoid function and ***ReLu*** is the most popular activation function people use now.

If we want to increse the accuracy of the OCR system, we may use ***ReLu*** function to replace the sigmoid function as the ***output layer***.

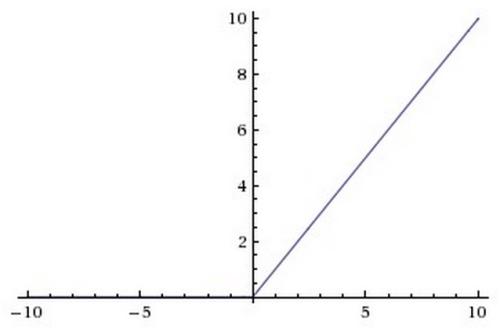
The graph of ReLu function is below:

Figure 9: the graph of ReLu function

1. **References**

[1] <https://gimg2.baidu.com/image_search/src=http%3A%2F%2F5b0988e595225.cdn.sohucs.com%2Fimages%2F20171102%2F3c33d1500b5f44ba8ae8d07c0f534d45.png&refer=http%3A%2F%2F5b0988e595225.cdn.sohucs.com&app=2002&size=f9999,10000&q=a80&n=0&g=0n&fmt=jpeg?sec=1613806502&t>

[4] <https://image.baidu.com/search/detail?ct=503316480&z=0&ipn=d&word=sigmoid%20%E5%87%BD%E6%95%B0%E7%9A%84%E5%AF%BC%E6%95%B0&step_word=&hs=0&pn=1&spn=0&di=41690&pi=0&rn=1&tn=baiduimagedetail&is=0%2C0&istype=0&ie=utf-8&oe=utf-8&in=&cl=2&lm=-1&st=undefined&cs=3>

[9] <https://image.baidu.com/search/detail?ct=503316480&z=0&ipn=d&word=Relu&step_word=&hs=0&pn=0&spn=0&di=90530&pi=0&rn=1&tn=baiduimagedetail&is=0%2C0&istype=0&ie=utf-8&oe=utf-8&in=&cl=2&lm=-1&st=undefined&cs=2958235369%2C1229304140&os=2824302674%2C2333762212&s>