Neural Network Model in English Letters Recognition

Jinhao Pan
Texas A&M University
Email: JinhaoPan@outlook.com

***Abstract* – The Optical Character Recognition, OCR, is a popular technology to recognize the texts from documents and photos. This is a way of machine learning, which become more and more popular around the world. However, this approach still contains some of flaws that the accuracy cannot achieve absolutely perfect 100%, for instance, some poor images or documents will essentially increase the error rate via the noises and disturbances. The concept of machine learning is advancing all the time, the neural network model can allow itself to learn and develop, update parameters and increase the accuracy. This paper will demonstrate how this algorithm works that has the accuracy almost 100%.**

I. Introduction

Machine learning are now very popular, which almost everywhere need the application to make things conveniently. OCRs is one of the applications to identify certain things, like scanning printed papers and converting documents into editable texts, and recognizing the license plate number for certain departments' security use. Nevertheless, for some slightly damaged images or texts, machine will make unexpected mistakes. The neural network model will build a better OCR to solve issues.

This paper will focus on the process of constructing, training a neural network model with the preprocessing data. Basically, a model applied in machine learning will predict the output from the input data. The neural network model can finish the job effectively.

The neural network is modeled from the human brain with the combination of some algorithms. Also, the neural network is like nets of interconnected neurons, which each neuron has a bias value, and tremendous weights. Then, neurons make up different layers to shape the neural network. The neural network has simply three layers, an input layers, inner layers, and an output layer.

The input layer clusters and process the input date for the machine to learn. This data is the training data, which includes raw data and corresponding. Consequently, the input layer can pass the processed data to the next level, inner layers.

The inner layers are the crucial layers to process and predict the results. The basic conception of this approach is that each neuron in the inner layers receives input data, such as x, and feed the output y forward to the next layer. For fully connected layer, the calculation is done as shown below:

$$y = w * x + b$$

Where w is the weight and b is the bias.

In the program, 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 like the following:

$$\begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \cdots & \cdots & \cdots \\ w_{n1} & \cdots & w_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ \cdots \\ b_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \cdots \\ y_n \end{bmatrix}$$

During training process, the output layer is unnecessarily required, and an accuracy layer is used alternatively in order to evaluate the training progress. During testing process, the output layer is used to transform received inputs to human-readable data. Usually, an ***activation layer*** is concatenated with fully connected layer. It amends the predictions and converts them into floating-point decimals between 0 and 1, which could be treated as possibilities. The ***Sigmoid Function*** is a very commonly used activation function:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Its value ranges from 0 to 1, and therefore is very straight forward for the user..

This whole process from input to output is known as ***forwarding***.

During training phase, a loss layer is used to evaluate the extent of the errors from the correct answers. This calculation is also used in the forwarding process. The loss layer gives out a ***loss value*** indicating the prediction accuracy. The function used in this layer is also called the ***cost function.*** This ***cost function*** takes the weights, biases, input raw data and the labels as its variables. A common cost function used in machine-learning is the ***Quadratic Cost Function***, defined as below:

$$J(\theta) = (h(\theta, X) - Y)^2$$

Where θ is the parameters across the neurons (namely the weights and biases), X is the input and Y is the label.

During training, given X and Y as input, the cost function is a function of θ, and θ is the only independent variable.

There is another algorithm named ***backpropagation***, which is essentially an approach to perform ***gradient descent*** in ~~the program~~ the optimization, or the learning algorithm that the neural network applies – with implausible efficiency, the machine-learning system modifies the values of the weights and biases in the inner layers so as to get a smaller loss value, and in turn a higher accuracy of prediction.

To understand more of how neural networks "learn", we have to go deeper into the back-propagation and gradient descent. Gradient is the partial derivatives of a function $F(x)$. It's an n-dimensional vector, the value of n depending on the number of variables in the function. In neural networks, the task of the machine is to find the gradient of the cost function to decrease cost function value ~~result~~.

After forwarding, the loss layer takes the derivative of the cost function, plugs in its data and passes it backward into the inner layers.

Note that the backpropagation algorithm works by the chain rule:

$$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

In this case, instead of trying to calculate the gradient of the function $\frac{dJ}{d\theta}$ straightforwardly, the inner layers calculate their corresponding gradients with respect to the layer before it, represented as a backward function rather than a constant value. The loss value is then propagated backwards throughout the inner layers, giving each layer their corresponding gradients. The layers can revise their weights and biases according to the gradients. The step of the change is defined by a ***hyperparameter,*** which is known as the ***learning rate***, symbolized as α in equations. The change in the weights and biases depends on this hyperparameter as $\Delta\theta = \alpha \frac{dJ}{d\theta}$. This significantly accelerate the network, since it does not need to run through millions of neurons again and calculate the gradient every time if one parameter modifies.

Usually, the network trains with batch size of input items, which means that 1024 or more pieces of items are parsed into the system at the same time. One ***epoch*** means we already processed all the existing input items. An ***accuracy layer*** can also be added in order to evaluate how accurate the output results is. It compares the results with the correct labels and digitalizes the evaluation. With *1* as correct and *0* as incorrect, the accuracy layer sums up the results and takes the average, and then prints it out for users to decide if the neural network is ~~appropriate and~~ good enough for application.

II. Materials and Methodology

Python 3.7 is the first-choice programming language in the program. Its intuitive syntax and extensive machine-learning libraries makes coding more efficient [1]. The following libraries are planned to be used:

- Matplotlib
- PIL
- Numpy
- Scipy
- Os

A. Calibration & Training

The first thing is to acquire training data. For this project, 17*17 pixels grayscale images of handwritten English letters are downloaded from machine-learning data repositories [1][2].

After the data is downloaded, it should be preprocessed into a format that could become machine-readable. In this case, the image data is converted into Numpy arrays.

Since the given data is only black and white pixels, it is easily to read them straightforwardly and resize ~~later~~, or more accurately, resize them into 1D arrays. The converted images are then saved into *.npy files – Numpy arrays that can be directly read in with numpy.load().

The second step is to construct the network.

For the input layer, every time the last batch is read in, the original input list gets shuffled to make the order of the original input data ~~seems~~ more random. The input layer does not do anything in the backward function since it is the first layer in the whole network:

```
Class FullyConnectedLayer
Function Init(Source, Batch_Size)
        Load Data and Save it to Self
        Get Data Length and Save it to Self
        Save Batch_Size to Self
        Initialize Position

Function Forward()
        If Position + Batch_Size >= Length
                Get remaining data
                Initialize Position
                Shuffle Data List
        Else
                Get a batch of data
                Move Position
        Return Data and Position

Function Backward()
        Pass
```

A fully connected layer always contains 26 neurons. The input for this layer is raw input data, and output is a matrix.

An activation layer is a non-linear function, the Sigmoid Function, modifies the output matrix to 26 floating numbers between 0.0 – 1.0, like probabilities, which correspond to a certain English letter.

```
Class ActivationLayer
Function Forward(Data)
        Save Data
        Return Sigmoid(Data)

Function Backward(Derivative)
        Return Derivative × Sigmoid(Data)'
```

The loss layer using the Quadratic Loss Function, which compares the activation layer output data and labels to calculate the loss values.

```
Class LossLayer
Function Forward(Data, Label)
        Set Correct Answers to Self
        Loss = (Data - Self.Label)^2 / Data.Count / 2
        Return Loss

Function Backward()
        Return Derivative
```

The accuracy layer takes the predicted outputs and labels to output the value from 0 to 1, which show the average correctness.

```
Class AccuracyLayer
Function Forward(Data, Label)
        For X in Data
                For Y in Label
                        If X=Y
                                Accuracy+=1
        Accuracy = 100% * Accuracy / Data.Count
        Return Accuracy
```

Put everything together:

```
Program OCR
DataLayer1 = InputLayer(Training Data, 1024)
DataLayer2 = InputLayer(Validation Data, 10000)
FCL = FullyConnectedLayer(17*17,26)
QuadLoss = LossLayer()
SigLayer = ActivationLayer()
Accuracy = AccuracyLayer()
If Weights.CSV and Biases.CSV Exists
        Load CSVs into FCL
Set Layer Learning Rate

Epochs = 20
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()


Save Weights.npy
Save Biases.npy
```

There are three datasets, training data, validation data and test data. The training data is used to modify the weights and biases. The validation data set is applied to prevent ***overfitting***, the phenomenon by which the machine fits the training data perfectly into the model, but however fails when novel outside data is fed into the network. The test data can be used to calibrate the hyperparameters and give the accuracy rate eventually.

The last step is to train the network.

[1]**All source: https://github.com/JP-25/OCR_MachineLearning**

The neural network is trained to have an accuracy of roughly 92% after about 20 epochs. It is possible sometimes for a network to reach a limit of only 84% or 89% accuracy, since initialization is random, it is highly possible that the lowest gradient might only be local minimum.

B. Basic Application

Once the neural network model is trained to obtain a relatively high accuracy, we can write an application program, which applies forwarding process and outputs the letter prediction to recognize English letters.

Consequently, the preprocessing method should be slightly modified since the only input is the path of the image that will be recognized. Cv2 is applied to read in the image in grayscale mode. It is then scaled to have a size of 17*17 pixels and normalized by setting all non-black pixels as white.

```
Function Preprocess (Image Path)
Read in Image in Grayscale Mode
Resize Image to 17 * 17
Save Image as temp.png and Read it out as an Array
Resize Array to 289 * 1
Array = Array / 255
Return Array
```

For the main function:

```
Function Preprocess
Create Instance of Neural Network Classes
If Weights.npy and Biases.npy Exists
        Load Weights and Biases
While True
     Get User Input
     If User Input = Exit
        Break
     Data = Preprocess (User Input)
     Forward Data through Hidden Layers
     From Activation Layer get Index of the highest value
     Map Index to A~Z ASCII Code
     Print Result
```

Now we get our first version of our application.

The application works fine for the testing data in the downloaded data sets. However, for self-drawn images in MS Paint, the application cannot output the expected results almost every time, about 0% accuracy. These issues will be discussed in the next minor section.

C. Issues & Debugging

During training, after when I add one more inner layer, sigmoid function, to simulate a deep learning to higher the accuracy, but the output of accuracy rate in the validation set grows very slow. But when I increase the size of epoch to 100, the accuracy rate begins to grow faster, and the final accuracy achieved 97%. Based on the common sense, the deep learning will always better than the shallow learning. So why the rate of growth become so slow if I add another inner layer?

Based on the formula $\Delta\theta = \alpha \frac{dJ}{d\theta}$, the hyperparameter does not change. There must be something wrong with the sigmoid function. The gradients of the sigmoid function vanished somehow. Then, I looked into the derivative of the sigmoid function, it is like a parabola which Y value is between 0 and 1. It means there is no big difference for the output Y between X = 5 or X =10, the Y's value will always be approximately 0, that is why the gradients disappeared. Then, I tried use some other activated functions instead of sigmoid, such as Relu, Tanh. They did not raise the accuracy a lot, then I tried an alternative way to replace the loss layer from quadratic loss function to ***Cross-Entropy Loss Function*** [5], and keep the original inner layers, one fullyconnect and one sigmoid.

$$1 - h(\theta, x)$$
$$J(\theta) = -y * log\left(h(\theta, x)\right) - (1 - y) *$$

```
Class LossLayer
Function Forward(Data, Label)
         Set Correct Answers to Self
         Loss=-Self.Label*log(Data)-(1-Self.Label)*(1-Data)
         Loss = Loss / Data.Count
         Return Loss

Function Backward()
```

$$\frac{dJ(\theta)}{dx} = \frac{h(\theta, x) - y}{h(\theta, x) - \left(1 - h(\theta, x)\right)}$$

$$sigmoid(x)' = sigmoid(x) * \left(1 - sigmoid(x)\right) = \frac{e^x}{1 + e^{-x}}$$

This loss layer function will cancel out (1 – sigmoid(x)) with (1 – Data) part, thus the gradient will not vanish anymore.

Another issue is that there is something wrong related my CrossEntropyLoss function, because the program does not output the correct and expected results. First of all, I tried to output the values of each inner layer from the forward process. Then I find that all the output values of the first batch size are as expected and seemed correct, which means that the input of the loss layer function is correct. The issue may happen either in the forward function of the loss layer function or backpropagation of the loss layer function that does not update the parameters correctly. Then, I check the output of the self.loss of Quadratic Loss Function and CrossEntropyLoss function separately, they are correct as well, then the problem should be in the backpropagation of the loss layer function, I notice that I do not divide the self.loss by batch size (self.x.shape[0]), which the program needs an average value. After I fix the problem, I get 99% accuracy of this model.

A major issue in preprocessing is that large images tend to lose a lot of pixel data during resizing and I need to make sure that the color type is grayscale. A large K (Figure 1) drawn in MS Paint and saved, for example, will be read in and resized to lose up to 50% of its original data (Figure 2), causing recognition errors, despite the 99% prediction accuracy of the network.

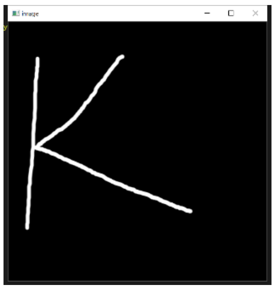


Figure 1. The original "K" magnified 120% in MS Paint

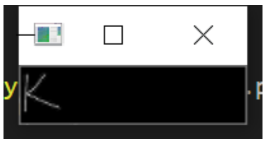


Figure 2. The resized 17*17px "K" magnified 700% in MS Paint

Therefore, preprocessing needs to be more complex and make it match the format of pictures used in training process. A method known as ***dilation*** [3] can be used to prevent too much pixel loss. It generally increases the white region of image or make the size of the prospect objects bigger during resize [4] process. Using the Open Computer Vision library, opencv2-python, we can improve the preprocessing algorithm substantially with the dilation function:

```
Function Preprocess (Image Path)
Image = Read (Image Path, Grayscale)
Flag = True
While Image.Height>34 and Image.Width>34
            Resize Image by Half
            If Flag
                        Dilation (Image)
                        Flag = False
            Else
                        Flag = True
)
Resize Image to 17*17
Normalize
Return Image
```

We resized the image to 50% of its original size and dilated it every time. We repeated this process for several times until it was small enough to match the 17*17 pixels standard.

This dilation algorithm was proved well for most images. However, it is not always perfect. For letters with small spaces, such as “B” with not professional hand drawing, the frequent dilation caused the right part of the “B” to disappear, making it likely for the machine to predict it as a “E” (Figure 3).

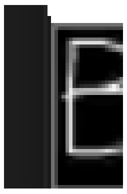


Figure 3. A frequently dilated “B” magnified 700% in MS Paint

Thus, I made the adjustment that when resizing the picture reached the ratio of 23% of the original image, the program will stop dilation. This method worked out, and I can get the correct outputs eventually.

III. Results

The final program was trained to have an accuracy of 99%.

The OCR worked quite fine, first test with grayscale images drawn in MS Paint (Figure 4):

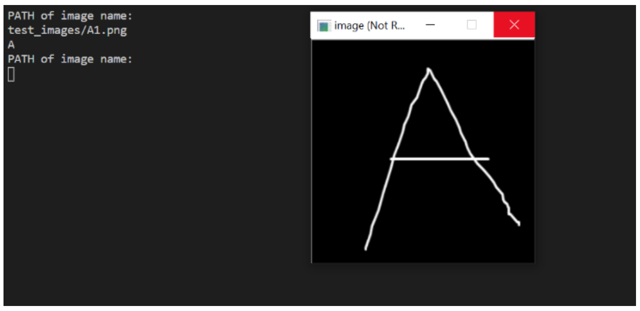

```
PATH of image name:
test_images/A1.png
A
PATH of image name:
▯
```

Figure 4. First test with normal image

Next, test with image distortions (Figure 5):

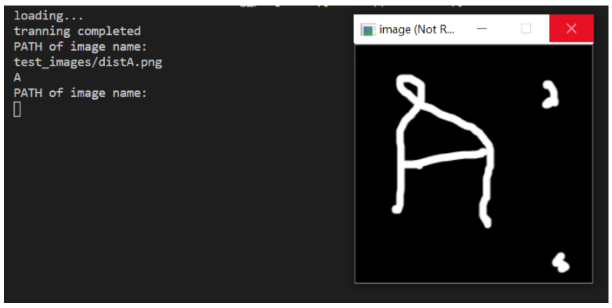
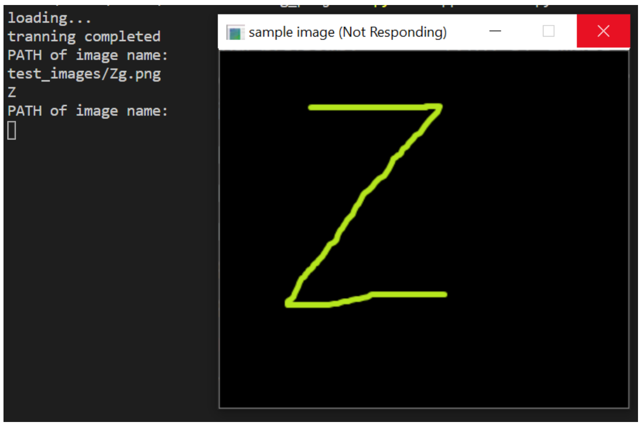

```
loading...
tranning completed
PATH of image name:
test_images/distA.png
A
PATH of image name:
▯
```

Figure 5. Second test with image distortions

Lastly, images with colored text and background (Figure 6 & 7):


```
loading...
tranning completed
PATH of image name:
test_images/Zg.png
Z
PATH of image name:
▯
```

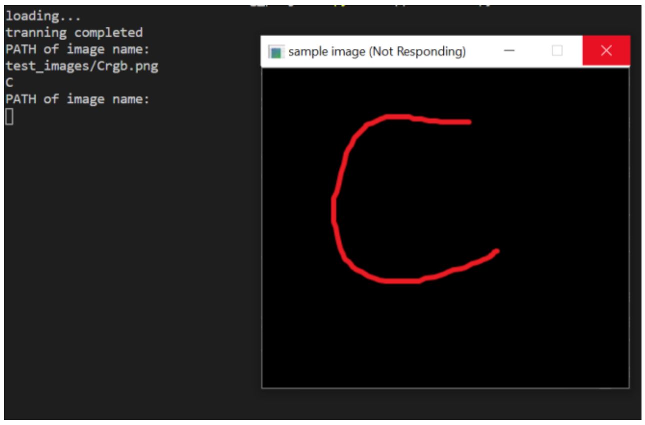

```
loading...
tranning completed
PATH of image name:
test_images/Crgb.png
C
PATH of image name:
▯
```

Figure 6 and 7. Final test with colored text and background

As can be seen from the figures, the results are correct. But when we changed color of the background instead of black. This concludes the beta stage of the OCR project.

IV. Discussion & Future Work

There are improvements that could be made to this application.

For preprocessing, the dilation algorithm is still not at its best. For images with very thin drawings or over thick drawings, the dilation algorithm does not have the proper way to deal with these situations, either it will make the resized image too thick that the application cannot recognize or it will make the resized image not fat enough to read. One approach can be calculating the ratio of the size of the pixels to the original size of pixels, and find the approximately best ratio to stop dilation or at some ratio with certain “fat” pixels we can apply ***erosion*** [3] method. For instance, if the ratio is in 15% range of the original ratio, no dilation or erosion is required. Or, dilating or eroding the image according to how much data is lost or how much extra data is provided.

```
Fore & Background Normalization Algorithm
Get set of grayscale pixel values
Sort values
Threshold = Median value
Data [Data >= Threshold] = 1
Data [Data < Threshold] = 0
Return Data
```

About the colored background and text issue, either more layers should be added to the network (this will be discussed later), or another filter can be added to preprocessing since this basic neural network model is almost accurately in predicting letter images with the black background and the white forecolor. In a random image, the text and the background have to be contrasting, then the average grayscale value of the background and the text can be calculated expectedly, a difference can thus be worked out, and another normalization can be done.

In addition, more layers can be added to the hidden layers so that more aspects of the image can be computed comprehensively. Neural networks with single inner layer "can approximate any function that contains a continuous mapping from one finite space to another" [7], and is protean; however, the flaw is that the accuracy will be far lower than one that has more layers and neurons. By adding more layers, we need to be careful to deal with the disappearance of nonlinear layers, like Sigmoid function.

For future advancement, a ***convolutional layer*** [8] can be added to make the current application be able to recognize the large amounts of text instead of only one letter. The convolutional layer can guarantee multiline text reading, it makes machine to be flexible in reading colored images even when pixels are distorted, because this layer samples the image effectively, filters it, searches and outputs specific features of the image before inputting deeper into the neural network. Also, we can rewrite the basic neural network into a ***Convolutional Recurrent Neural Network*** [9], or ***CRNN*** for short. Its hidden layers contain three major layers, convolutional layers, recurrent layers and transcription layers. They offer sophisticating predictions through featured sequential extraction, labeling and transcription [8]. Unfortunately, this paper is not going to discuss about this topic.

V. Acknowledgement

I would like to thank Xu Yuan for insightful ideas and comments on this topic. I would also like to thank Ye Cheng who gave corrections and suggestions to this paper.

VI. Literature Cited

[1] Cohen, G., Afshar, S., Tapson, J., & van Schaik, A. (2017). EMNIST: an extension of MNIST to handwritten letters. Retrieved from http://arxiv.org/abs/1702.05373

[2] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[3] "Eroding and Dilating." Eroding and Dilating - OpenCV 2.4.13.7 Documentation,docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/ erosion_dilatation.html.

[4] "cv2.Resize() - OpenCV Python Function to Resize Image - Examples." TutorialKart,www.tutorialkart.com/opencv/python/opencv-python-resizeimage/.

[5] "Basic Thresholding Operations¶." Basic Thresholding Operations OpenCV 2.4.13.7 Documentation, docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html.

[6] "Loss Functions¶." Loss Functions - ML Cheatsheet Documentation, mlcheatsheet.readthedocs.io/en/latest/loss_functions.html.

[7] "The Number of Hidden Layers." Heaton Research, 28 Dec. 2018, www.heatonresearch.com/2017/06/01/hidden-layers.html.

[8] "Convolutional Neural Network." Unsupervised Feature Learning and Deep Learning Tutorial, http://deeplearning.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/.

[9] Shi, Baoguang, et al. "An End-to-End Trainable Neural Network for ." An End-to-End Trainable Neural Network for Image-Based Sequence Recognition and Its Application to Scene Text Recognition, Huazhong University of Science and Technology, Wuhan, China, 21 July 2015, arxiv.org/pdf/1507.05717.pdf