Image Preprocessing and the Neural Network Model in English Alphabet OCR Training

Yuyin Li
Ningbo XiaoShi High School
Email: 1324555186@qq.com

***Abstract* – The Optical Character Reader, or OCR for short, is an extremely power tool for converting images of texts into machine-encoded text. For the past few decades, people have developed AI algorithms to recognize text specifically. They function well, but with many limitations, such as not being able to learn from past errors, and minor noises and disturbances in the image make them exceptionally prone to misrecognition. However, with the recently established machine learning technology, the algorithm can allow itself to evolve through a simple learning process – trial & error. This paper will demonstrate how image preprocessing and neural networking combined could build up a fully functional OCR that has an accuracy of nearly 100%.**

I. Introduction

OCRs are now everywhere – image scanners in the office, cameras at parking lot entries, smartphones etc. It is easy for classifiers like decision trees to recognize well-written text; however, for distorted text, they usually make mistakes. To counter this problem, the neural network model is used to construct a better OCR.

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 model is used to predict a result. The neural network model proved to be effective in completing such task.

A neural network is essentially a web of interconnected neurons, each holding a bias value, and numerous weights depending on how many other neurons are connected to it. The neurons form layers, which then interconnect to form a fully functional neural network. There are basically three parts in a neural network, an ***input layer***, ***hidden layers*** and an ***output layer***.

The input layer gathers and preprocesses all the data needed for the machine to trial on and learn from. This data is known as the ***training data***. It contains raw data and its ***labels***, which are the "ground truth answers" to the raw data's predictions. The input layer passes the processed data to the hidden layers.

The hidden layers are the main layers that function and predict the results. Basically, for each neuron in the hidden layers, the input x is given, and the output y is pushed forward to the next layer. The calculation is done as shown below:

$$y = w * x + b$$

Where w is the weight and b is the bias.

Usually during coding, data flow is passed through layer by layer. The input matrix, dotted with a weight matrix and then added to a bias matrix, will generate the output matrix. The output matrix is then passed on to the next layer like the following:

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

During training, the output layer is not necessarily needed, and an accuracy layer is used alternatively in order to evaluate the training progress. During testing, the output layer is used to convert received inputs into human-friendly data. 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:

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

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

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

During training phase, a loss layer is used to evaluate how "off" the results are from the correct answers. This calculation is also counted 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, raw input data and the labels as its variables. A common cost function used in machine-learning is the ***Quadratic Cost Function***, depicted 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.

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

Through a method known as ***backpropagation***, which is essentially an algorithm that allows programs to perform ***gradient descent*** – the optimization, or the learning algorithm that the neural network uses – with unbelievable efficiency, the machine-learning system modifies the values of the weights and biases in the hidden layers so as to get a smaller loss value, and in turn a higher accuracy rate of prediction.

To have a full understanding of how neural networks "learn", we have to go deeper into backpropagation and gradient descent. The gradient is basically the partial derivatives of a function $F(x)$. It is an n-dimensional vector, the value of n depending on the number of variables in the function. In machine-learning neural networks, the job of the machine is to find the gradient of the cost function in order to decrease the cost function's result.

After forwarding, the loss layer takes the derivative of the cost function, plugs in its data and passes it backward into the hidden 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 $\frac{dJ}{d\theta}$ – the gradient of the function – directly, the hidden layers calculate their respective gradients with respect to the layer before it, represented as a backward function rather than a fixed value. The loss value is then propagated

backwards throughout the hidden layers, giving each layer their respective gradients. The layers can then calibrate their weights and biases according to the gradients. The rate of 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 speeds up the network, because it doesn't have to run through millions of neurons and calculate the gradient every time if one parameter changes.

Usually, the network trains with batches of input items, which means that 1000 or more items are parsed into the system at the same time. The end of one ***epoch*** means we have already processed all the existing input items. An ***accuracy layer*** can also be added in order to evaluate how accurate the prediction is. It compares the predictions with the correct labels and quantizes the assessment. 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 whether the neural network is good enough for application or not.

II. Materials and Methodology

For this task, Python 3.7 is the first-choice programming language. 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 hand-written English letters are downloaded from machine-learning data repositories [1][2].

After the data is downloaded, it should be processed into a format that could be read by the machine. In this case, the image data is converted into Numpy arrays.

Since the given data is only black and white pixels, it is reasonable to read them in directly and then resize, or more accurately, rearrange 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 build the network.

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 input data seem more random. Note that the input layer does not do anything in the backward function since it is the first layer in the whole network:

```
Class InputLayer
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
```

[1] **All source code for this project is available at the following URL:** https://github.com/JustRodneyLee/ML101-OCR

A fully connected layer consists of 26 neurons. The input for this layer is raw input data, and the output, an output matrix.

```
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(Data)
      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(Derivative)
      Self.Gradient = Derivative × Self.Data
      Return Self.Gradient
```

An activation layer, using the Sigmoid Function, takes in the output matrix and normalizes it to give out 26 probability values, each value corresponding 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, takes in activated data and truth labels to output a loss value:

```
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
```

Lastly, the accuracy layer, which takes in predicted outputs and truth labels to output a float between 0~1 that represents the average correct percentage:

```
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
```

Finally, to put them all together:

```
Program OCR_Training
DataLayer1 = InputLayer(Training Data, 1024)
DataLayer2 = InputLayer(validation Data, 10000)
FCL = FullyConnectedLayer(17*17,26)
QuadLoss = LossLayer()
SigLayer = ActivationLayer()
Accuracy = AccuracyLayer()
Add FCL and QuadLoss to HiddenLayers List
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 Hidden Layers with Data, Labels
            Calculate Loss Sum

            D = LossLayer.Backward()
            Backpropagate through Hidden Layers with D

            If Position = 0
                  DataLayer2.Forward()
                  Output Average Loss
                  Output Accuracy.Forward()

Save Weights.CSV
Save Biases.CSV
```

During training there are usually 3 datasets. The first as seen in the pseudocode is the training data, the items used to calibrate weights and biases. The second is validation data, a set used to prevent ***overfitting***, the phenomenon by which the machine fits the training data perfectly into the model, but however fails when foreign data is fed into the network. This set of data is used to calibrate the hyperparameters, but for a case simple as this, its function doesn't differ too much from the last type of data, which is the test data, the set used to give accuracy ratings for the program.

The last step is to train the network.

The neural network is trained to have an accuracy of roughly 94% after about 500 epochs. It is possible sometimes for a network to reach a limit of only 84% or 89% accuracy, since initialization is random, the lowest gradient might only be a local minimum.

B. Basic Application

Once the neural network is trained to have a relatively high accuracy, we can proceed to write a recognition program, which only does *forwarding* and outputs the letter prediction.

This time, the preprocessing needs to be slightly altered since the only input is the path of the image that will be recognized. Instead of Matplotlib, which only reads one image mode, Pillow (PIL) is used 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 > 0] = 1
Return Array
```

For the main function:

```
Function Preprocess
Create Instance of Neural Network Classes
If Weights.CSV and Biases.CSV Exists
      Load Weights and Biases
While True
      Get User Input
      If User Input = Bye or Exit or Quit
            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 alpha version of the OCR application.

The application works fine for the testing data in the downloaded data sets. However, for self-drawn images in MS Paint, the application malfunctions nearly every time (An accuracy of almost 0%). These issues will be discussed in the next minor section.

C. Issues & Debugging

A major issue in preprocessing is that large images tend to lose a lot of pixel data during resizing. A large E (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 mistakes, despite the 94% prediction accuracy of the network.

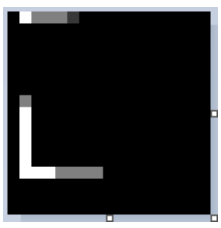

Figure 1. The resized 17*17px "E" magnified 800% in MS Paint

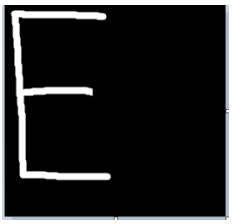

Figure 2. The original "E" magnified 120% in MS Paint

Therefore, preprocessing needs more sophisticated implementation. A method known as ***dilation*** [3] can be used to prevent too much pixel loss. It essentially makes the image "fatter", making it less prone to data obstruction during resize [4]. Using the Open Computer Vision library, opencv2-python, we can improve the preprocessing algorithm substantially with the help of a dilation function:

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

We resize the image to half of its original size, and then dilate it for every other time it is resized. This process is repeated for numerous times until the image is small enough to directly be scaled to the standard 17*17 pixels.

This alternative dilation algorithm proved well for most images. Yet, it is not perfect. For letters with small spaces such as "A", the frequent dilation caused the triangle inside of the "A" to disappear, making it likely for the machine to predict it as a "V" (Figure 3).

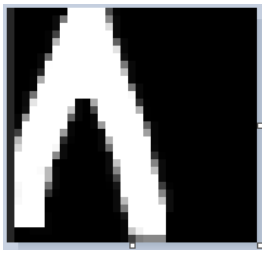

Figure 3. A frequently dilated "A" magnified 700% in MS Paint

Another issue is that the network still isn't accurate enough, and lacks of training. Since this set of weights and biases have already reached its limit, a new set of weights and biases should be trained rather than continue training the original set.

To ensure the accuracy is high enough, the original looping for a limited number of epochs can be changed to an indefinite loop, preferring a while loop with a condition that the training keeps on going until the accuracy gets higher than a certain threshold. To make the program more flexible, the threshold can be set as a user input.

The main program will be structured like the following:

```
Program OCR_Training
Initialization of Layers
Load Weights.CSV and Biases.CSV
TargetAccuracy = Get User Input

While Accuracy*100<TargetAccuracy
      Forwarding Data through Hidden Layers
      Backpropagation through Hidden Layers
      If Position = 0
            Forward Validation Dataset
            Calculate Accuracy
            Calibrate Learning Rate Accordingly
            Break
Save Weights.CSV and Biases.CSV
```

III. Results

The final program was trained to have an accuracy of 98%.

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

Figure 4. First test with normal images

Next, test with image distortions (Figure 5):

Figure 5. Second test with image distortions

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

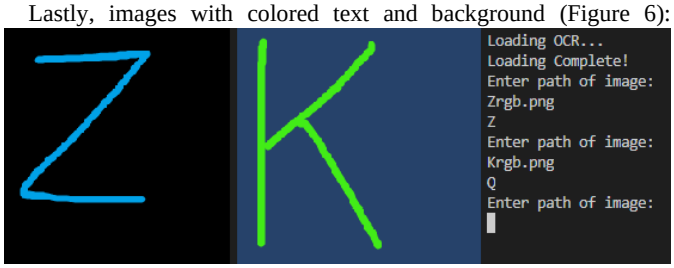

Figure 6. Third and final test with colored text and background

As can be seen from the figures, the results are mostly satisfactory (except the last colored image test). This concludes the beta stage of the OCR project.

IV. Discussion & Future Work

There are improvements that could be made to this OCR application.

For preprocessing, the dilation algorithm is still not at its best. For images with exceptionally thin or slim text, or very fat and plump text, the alternating dilation algorithm doesn't quite stand a chance. One proposal is to calculate the ratio of text pixels to the total amount of pixels, and after resize, if the ratio is in 10% range of the original ratio, no dilation or ***erosion*** [3] is needed. If otherwise, dilate or erode the image according to how much data is lost or how much excess data is produced.

Addressing the colored text and background problem, either more layers can be added to the network (this will be discussed later), or another filter can be added to preprocessing since this basic neural network is highly successful in predicting letter images with a pure black background and a white forecolor. In a random image, the text and the background must be contrasting, therefore the average grayscale value of the background and the text can be calculated, a difference can thus be worked out, and another normalization can be done. A good approach to this issue is probably using the threshold function in OpenCV [5]. After retrieving the grayscale of the target image, a threshold binary operation can be performed on the image, which yields an array with only 2 values, 0 and 1, creating a perfect canvas for the neural network to process.

```
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
```

For training, a ***Cross-Entropy Loss Function*** [6] can be used to speed up the training process.

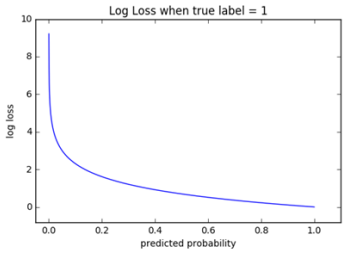

Figure 7. Graph of the Cross-Entropy Loss Function [6]

Since the function gives a higher loss value for a low accuracy value, the gradient and learning rate can be increased to speed up the learning process during backpropagation.

Also, more layers can be added to the hidden layers so that more aspects of the image can be computed. Single layered neural networks are capable of universal approximation [7], and is quite versatile; however, the tradeoff is that the accuracy will be much lower than one that has more layers and neurons, and essentially one that is more specialized to do the task.

For further development, a ***convolutional layer*** [8] can be added to make the current OCR capable of reading in large amounts of text rather than just one image of a letter. The convolutional layer does not only grant multiline text reading, it also makes the machine more tolerant towards colored images as well as distortion, since this layer samples the image, filters it, looks for and outputs specific features of the image before going deeper into the neural network. Or having a more holistic and even ambitious approach, we could rewrite the basic neural network into a ***Convolutional Recurrent Neural Network*** [9], or ***CRNN*** for short. Its hidden layers contain 3 main types of layers, convolutional layers, recurrent layers and transcription layers. They provide much sophisticated predictions through feature sequence extraction, labeling and transcription [8]. However, this paper will not 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-resize-image/.

[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, ml-cheatsheet.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