Multivariable Calculus Final Project: Using Neural Networks and Backpropagation Algorithms to Recognize Hand-written Digits

Yitong (Tony) Zhao
Multivariable Calculus, Period 1
Princeton International School of Mathematics and Science

Contents

1. Introduction

We often take for granted for some of the technologies that we use in our daily lives. One example is OCR (Optical Character Recognition) technology, which is used in many applications such as scanning documents. The technology basically accept images of document and convert them into characters. In this project, we will try to implement a small subset of OCR technology by using neural networks and backpropagation algorithms to recognize hand-written digits.

1.1. Definition of the Problem

Provided a training dataset of images and the corresponding labels (correct answer) of hand-written digits, we want to let computers to recognize hand-written digits. The dataset we used is the NMIST dataset, which is consist of hand-written digit image of size 28×28.

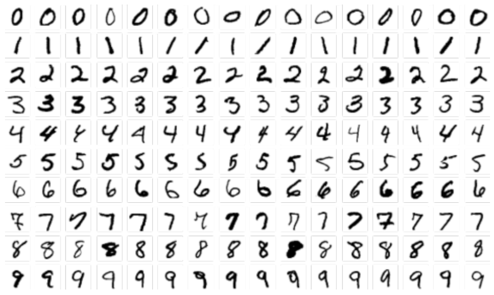


Figure 1: Examples of images in the NMIST dataset

The rough procedure of the hand-digit recognition program can be described as $\mathbb{R}^{28\times28} \to \mathbb{R}^{10} \to \mathbb{Z}$. The program will first read in a matrix of size 28×28 and output a vector of size 10, which represents the likelihood of the input image being each digit. The program will then output the digit with the highest probability.

2. Simpler Classification

The hand-written recognition task introduced previously is essentially a classification problem. Directly starting with it might be too complicated. In this section, we introduce a simpler classification problem to work on.

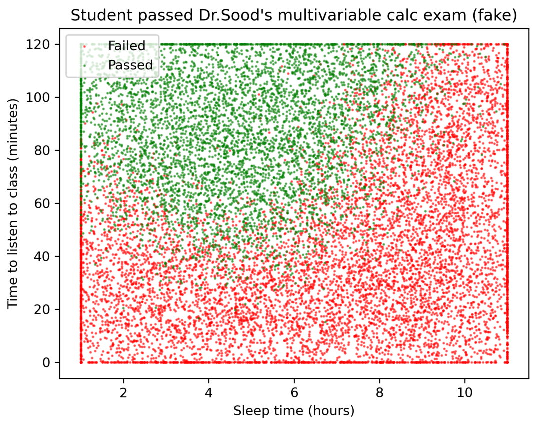


Figure 2: Fake Dataset of students' final exam result in Dr.Sood's multivariable calculus class. The x-axis is the time (in hour) that the student slept the night before the exam. The y-axis is the average minutes of time students listen to Dr.Sood's lecture per class meeting.

From Figure 2, we can see that the students' pass rate is maximized if they sleep for about 4 hours before the exam and listen as long as they can. Now we would like to design a algorithm to automatically find a function to predict if the student passed based on the time they slept and the time they listened to the lecture.

We assume that the function used for classification is a quadratic form:

$$f(\vec{x}) = w_1 x_1^2 + w_2 x_2^2 + w_3 x_1 x_2 + w_4 x_1 + w_5 x_2 + w_6 \tag{1}$$

Here the function f takes in a vector $x = (x_1, x_2)$, representing the sleepign time and lecture-listening time of a student, and output a scalar, representing the likelihood of the student passing the exam. We would like to make the computer automatically adjust the weights $w_1, w_2, w_3, w_4, w_5, w_6$ to make the function f as close to the real data as possible.

We first define a metric measuring the "closeness" of our prediction:

$$E = \text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(f(\vec{x_i}) - y_i\right)^2} \tag{2}$$

Here, RMSE stands for Root Mean Square Error. It first take the average of the square of the difference between the prediction and the real value, and then take the square root of the average. We would like to minimize the RMSE to make our prediction as close to the real data as possible.

2.1. Gradient Descent

To minimize error, we would like to investigate how change in different weights affact the error, and this is exactly what gradients are doing:

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_6} \end{bmatrix} \tag{3}$$

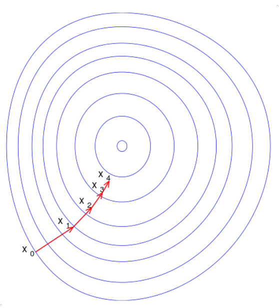
Assume that currently we selected a set of weight $\vec{w}_1$, we'd like to use information from the graident to find a direction to move the weights to a new set $\vec{w}_2$ that will reduce the error. Directional derivative in the direction of u can be calculated using the following formula:

$$D_u E = \nabla E \cdot u \tag{4}$$

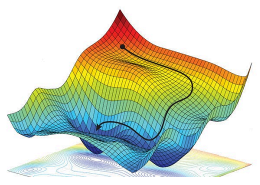
When u is the negative of the gradient, the directional derivative will be the most negative, which means the error will be reduced the most. Therefore, we can use the following formula to update the weights:

$$\vec{w}_{i+1} = \vec{w}_i - \eta \nabla E \tag{5}$$

The technique described in Equation 5 is called gradient descent. It is a widely used optimization algorithm in machine learning. Here η is the learning rate, with the smaller η the more stable the algorithm will be, but the slower it will converge.



(a) gradient descent in 2D

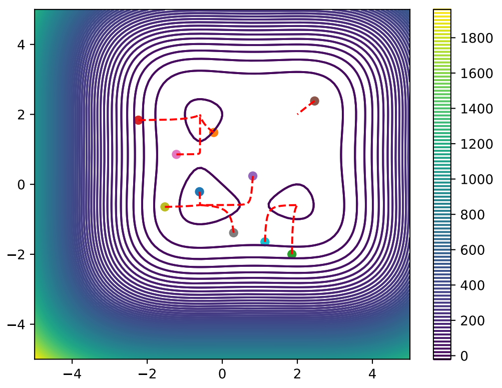


(b) gradient descent in 3D

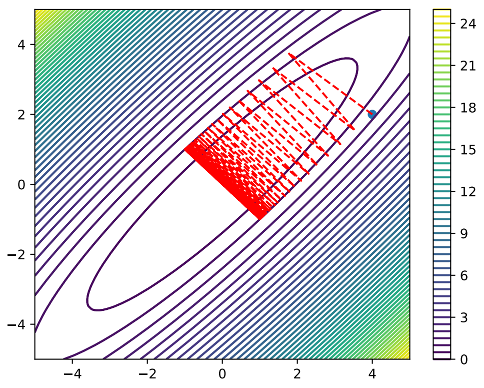
Figure 3: Illustration of gradient descent algorithm

Figure 3 shows the gradient descent algorithm in 2D and 3D. The algorithm starts at a random point and iteratively move towards the minimum of the function. In subfigure (b), one can see that the process of gradient descent is similar to a ball rolling down a hill.

Gradient descent also have certain limitations, which are illustrated in the following two graphs:



(a) Multiple local minima.
$f(x, y) = x^4 - 3x^3 + 4x + y^4 - 3y^3 + 4y$



(b) Oscilation
$f(x, y) = 0.26(x^2 + y^2) - 0.48xy$

Figure 4: Illustration of limitations of gradient descent algorithm

Figure 4 shows two limitations of gradient descent. In subfigure (a), one can see that the function has multiple local minima. And the initial guess on the weights will affact the eventual convergence point. In subfigure (b), one can see that the function has a oscillation pattern. The learning rate η will affact the convergence of the algorithm.

Applying gradient descent with selected starting point and learning rate, we can find a fairly good set of weights to predict the student's exam result. The following graph shows the prediction of the function f on the dataset:

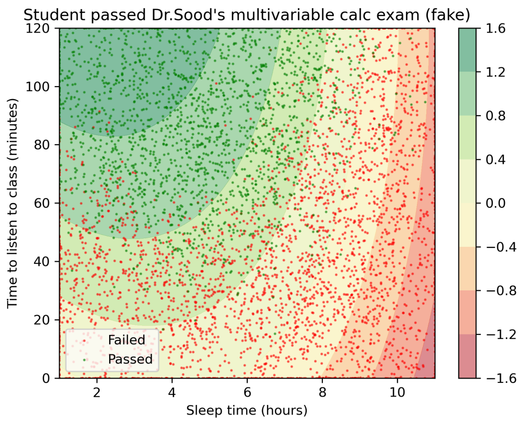
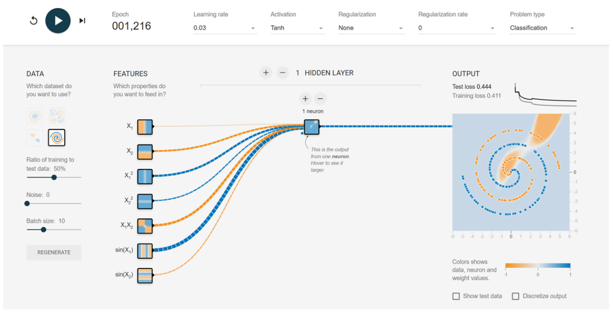


Figure 5: Prediction of the function f on the dataset

Here we can see that in the region with greater passing students, the function gives a higher value, and vice versa.

Proposing quadratic form as the function to predict the students' exam worked. However, the function is not flexible and complicated enough to capture all the features of hand-written digits. One can verify this by doing some test at Tensorflow Playground:


Epoch 001,216
Learning rate 0.03
Activation Tanh
Regularization None
Regularization rate 0
Problem type Classification

Ratio of training to test data: 50%
Noise: 0
Batch size: 10

Test loss 0.444
Training loss 0.411

This is the output from one **neuron**. Hover to see it larger.

Colors shows data, neuron and weight values.

Figure 6: The result of using a single layer neural network to classify the spiral dataset.

The figure shows the convergence of a single layer neural network on a spiral dataset. It can be seen that with a simple, one-layer network, the algorithm is not able to classify the spiral dataset. Therefore, we need to introduce a more complicated function to predict the hand-written digits.

3. Neural Networks

Neural networks are formulated with the inspiration of the human brain. Each neuron in human brain is connected to many other neurons as input, when the other neurons are activated in certain patterns, the neuron will be activated and send signals to other neurons. The same idea is used in neural networks.

Formally, a neuron can be described as:

$$a(\vec{x}) = \sigma(\vec{w} \cdot \vec{x} + b) \tag{6}$$

Here, a represent the output of the neuron, $\vec{w}$ is a vector representing the weight to each input, $\vec{x}$ is the input vector, b is the bias term, and σ is the activation function. The activation function is used to introduce non-linearity to the model, which add another layer of flexibility to the model. Further, if there exist no activation function, different layers of the network will be linear transformation, and can be represented by matrices. The multiplication of matrices is another matrix, so without activation function, separating the network by layers will be useless.

The domain of activation function should always be $(-\infty, \infty)$, as the dot product of $\vec{w}$ and $\vec{x}$ can be any real number. The activation function should also be non-linear, as the composition of linear functions is still linear. In this project, the sigmoid function is used:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{7}$$

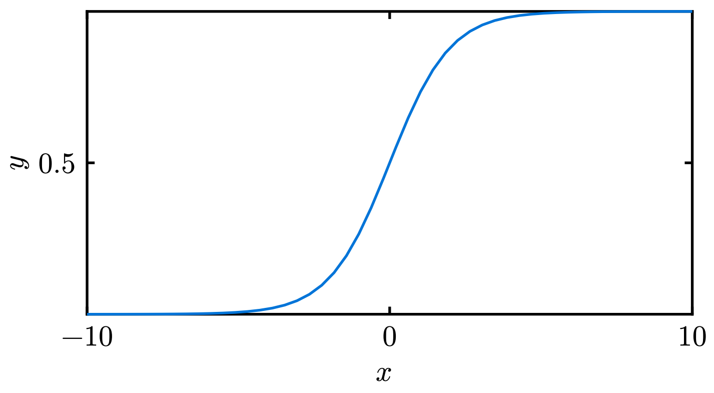


Figure 7: The sigmoid function

One can see that the function compresses the input from $(-\infty, \infty)$ to $(0, 1)$.

For this project, a neural network is configured with one input layer of size 28×28, two hidden layer of size 16, and an output layer of size 10. We can count the number of parameters in the network:

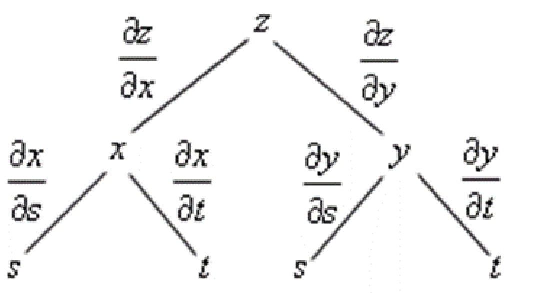
$$\begin{gathered} \text{Weights} : 28 \times 28 \times 16 + 16 \times 16 + 16 \times 10 = 12960 \\ \text{Bias} : 16 + 16 + 10 = 42 \\ \text{Total} : 12960 + 42 = 13002 \end{gathered} \tag{8}$$

When performing normal gradient descent optmization method, we need to calculate the gradient of this network as follows:

$$\begin{gathered} \frac{\partial f}{\partial w_1} = \frac{f(w_1+\varepsilon, w_2, ..., w_{13002}) - f(w_1-\varepsilon, w_2, ..., w_{13002})}{2\varepsilon} \\ \vdots \\ \frac{\partial f}{\partial w_{13002}} = \frac{f(w_1, w_2, ..., w_{13002}+\varepsilon) - f(w_1, w_2, ..., w_{13002}-\varepsilon)}{2\varepsilon} \\ \nabla E = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_{13002}} \end{bmatrix} \end{gathered} \tag{9}$$

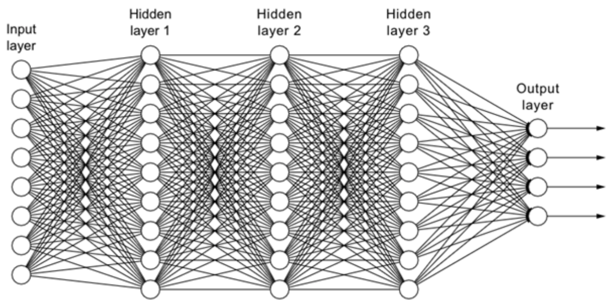
We can see that to calculate the gradient of the network, we will have to perform n times of network evaluation, where n is the number of parameters in the network. This is very inefficient.

Observing the structure of neural networks, one can see that it is similar to the tree diagram we used when calculating chain rule for partial derivatives:



z
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
x
y
$\frac{\partial x}{\partial s}$
$\frac{\partial x}{\partial t}$
$\frac{\partial y}{\partial s}$
$\frac{\partial y}{\partial t}$
s
t
s
t

(a) Chain rule for partial derivatives. The function can be expressed by: $z = f(x, y)$ $x = g(s, t)$ $y = h(s, t)$



(b) Neural network

Figure 8: Comparison between chain rule for partial derivatives and neural networks

In subfigure (a), the partial derivative of $\frac{\partial z}{\partial s}$ can be calculated by summing all the path connecting these two variables:

$$\frac{\partial z}{\partial s} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial s} \tag{10}$$

And we're able to calculate all the partial derivatives related to z while doing one complete evaluation of the network. This is the idea of backpropagation algorithm.

4. Backpropagation Algorithm

4.1. Notation

Here are some notations used in the backpropagation algorithm:

- σ: the activation function
- E: the final error
- Err(): the error function, in thi project, MSE is used.
- $\hat{y}$: the predicted value
- y: the actual value
- l: the layer number, smaller values indicate closer to the input layer, while larger values indicate closer to the output layer. When backpropagating, we start from the output layer and move towards the input layer.
- $w_{ji}^{[l]}$: the weight connecting the i-th neuron in layer l to the j-th neuron in layer $l + 1$
- $b_i^{[l]}$: the bias term of the i-th neuron in layer l
- $z_i^{[l]}$: the weighted sum of the input of the i-th neuron in layer l. Or the output without activation function.
- $a_i^{[l]}$: the output of the i-th neuron in layer l
- $n^{[l]}$: the number of neurons in layer l
- Previous: layer closer to the input layer
- Next: layer closer to the output layer

4.2. One Neuron Per Layer

We start analyzing the backpropagation algorithm by a simple case where each layer of the neural network have only one neuron:

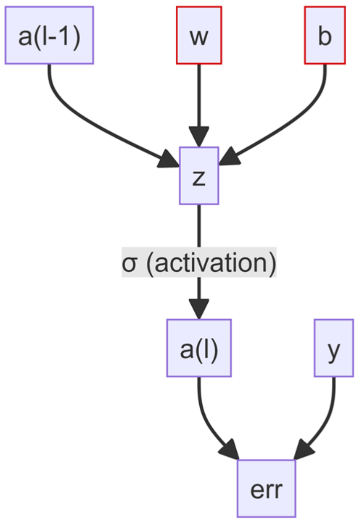


Figure 9: One neuron per layer

This network can be expressed by the following formula:

$$\begin{aligned} z^{[l]} &= w^{[l]} a^{[l-1]} + b^{[l]} \\ a^{[l]} &= \sigma\left(z^{[l]}\right) \\ E = \mathrm{Err}\left(a^{[l]}, y\right) &= \left(a^{[l]} - y\right)^2 \end{aligned} \tag{11}$$

We'd like to know the partial derivative of the error with respect to the weight of the neuron in layer l:

$$\begin{aligned} \frac{\partial E}{\partial w^{[l]}} &= \frac{\partial z^{[l]}}{\partial w^{[l]}} \frac{\partial a^{[l]}}{\partial z^{[l]}} \frac{\partial E}{\partial a^{[l]}} \\ \frac{\partial z^{[l]}}{\partial w^{[l]}} &= a^{[l-1]} \\ \frac{\partial a^{[l]}}{\partial z^{[l]}} &= \sigma'\left(z^{[l]}\right) \\ \frac{\partial E}{\partial a^{[l]}} &= 2\left(a^{[l]} - y\right) \end{aligned} \tag{12}$$

To calculate the partial derivative of the error with respect to the bias term of the neuron in layer l or with respect to the output of the previosu layer, we can just replace $\frac{\partial z}{\partial w}$:

$$\begin{aligned} \frac{\partial z^{[l]}}{\partial b^{[l]}} &= 1 \\ \frac{\partial z^{[l]}}{\partial a^{[l-1]}} &= w^{[l]} \end{aligned} \tag{13}$$

Now we have propagated the error from the error to the last layer, we now need to propagate the error back to the input layer.

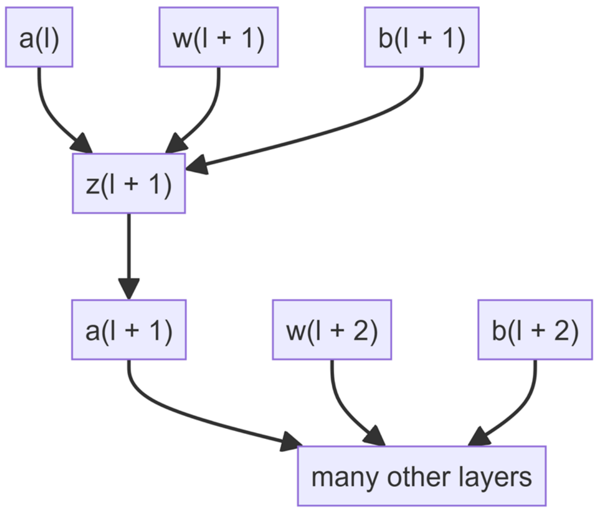


Figure 10: Backpropagation of error

The propagtion from the next layer to the current layer can be described as:

$$\frac{\partial E}{\partial a^{[l]}} = \frac{\partial z^{[l+1]}}{\partial a^{[l]}} \frac{\partial a^{[l+1]}}{\partial z^{[l+1]}} \frac{\partial E}{\partial a^{[l+1]}} \tag{14}$$

We can summarize the backpropagation when there are only one neuron per layer with the following algorithm and Equation 12. Equation 13, and Equation 14:

```
1:  function BackpropagationForSingle-neuronLayer(layer_cnt)
2:      l ← layer_cnt
3:      par_E_par_a[l] ← 2(a[l] - y)
4:      par_E_par_z[l] ← par_E_par_a * sigma'(z[l])
5:      par_E_par_w[l] ← par_E_par_z * a[l-1]
6:      while $l > 1$ do
7:          par_E_par_a[l-1] ← par_E_par_z[l] * w[l]
8:          par_E_par_z[l-1] ← par_E_par_a[l-1] * sigma'(z[l-1])
9:          par_E_par_w[l-1] ← par_E_par_z[l-1] * a[l-2]
10:         l ← l - 1
11:     return par_E_par_w, par_E_par_b
```

4.3. Multiple Neurons Per Layer

4.3.1. Weight

Now each neurons have multiple weights as it is connected to multiple neurons from the previous layer. We first consider the calculation of partial derivative of error with respect to the weight of the neuron in layer l, remember that $w_{ji}^{[l]}$ denote the weight connecting the i-th neuron in layer l to the j-th neuron in layer $l+1$:

$$\begin{aligned} \frac{\partial E}{\partial w_{ji}^{[l]}} &= \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial E}{\partial a_j^{[l]}} \\ &= a_i^{[l-1]} \sigma'\left(z_j^{[l]}\right) \frac{\partial E}{\partial a_j^{[l]}} \end{aligned} \tag{15}$$

Since the formula will be eventually coded into compouter programs, it is very important to improve the performance of the algorithm. One way of doing so is to vectorize the calculation, as vector and matrix operations are carefully optimized in computer systems.

From Equation 15, we can realized that some values share the same index, and we can group them:

$$\begin{aligned} r_j^{[l]} &= \sigma'\left(z_j^{[l]}\right)\frac{\partial E}{\partial a_j^{[l]}} \\ \frac{\partial E}{\partial w_{ji}^{[l]}} &= r_j^{[l]} a_i^{[l-1]} \end{aligned} \tag{16}$$

We can write all weights on a neuron in matrix form:

$$\frac{\partial E}{\partial w^{[l]}} = \begin{bmatrix} r_{j=1}a_{i=1}^{[l-1]} & r_1 a_2^{[l-1]} & \dots & r_1 a_n^{[l-1]} \\ r_2 a_1^{[l-1]} & r_2 a_2^{[l-1]} & \dots & r_2 a_n^{[l-1]} \\ \vdots & \vdots & \ddots & \vdots \\ r_n a_1^{[l-1]} & r_n a_2^{[l-1]} & \dots & r_n a_n^{[l-1]} \end{bmatrix} \tag{17}$$

We can see that this is actually a matrix multiplication of vector r and $a^{[l-1]^T}$:

$$\begin{aligned} \frac{\partial E}{\partial w^{[l]}} &= \begin{bmatrix} r_1^{[l]} \\ r_2^{[l]} \\ \dots \\ r_n^{[l]} \end{bmatrix} \begin{bmatrix} a_1^{[l-1]} & a_2^{[l-1]} & \dots & a_n^{[l-1]} \end{bmatrix} \\ &= r^{[l]}\left(a^{[l-1]}\right)^T \end{aligned} \tag{18}$$

4.3.2. Bias

Since there are only one bias term per neuron, the calculation of partial derivative of error with respect to the bias term is no difference from the case where there is only one neuron per layer:

$$\begin{aligned} \frac{\partial E}{\partial b^{[l]}} &= \frac{\partial z_j^{[l]}}{\partial b^{[l]}}\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}}\frac{\partial E}{\partial a_j^{[l]}} \\ &= 1 \cdot \sigma'\left(z_j^{[l]}\right)\frac{\partial E}{\partial a_j^{[l]}} \\ &= r_j^{[l]} \end{aligned} \tag{19}$$

4.3.3. Output of Previous Layer

To propagate the error back to previous layers, we would like to know the partial derivative of the error with respect to the output of the previous layers. Refer to the tree diagram in Figure 8, we can see that to take the partial derivative of any variable α with the respect of β, we have to sum up all the path connecting α and β.

As the output of a neuron in the previous layer is connected to multiple neurons in the next layer, we can sum up all the path connecting the output of the previous layer and the neuron in the next layer:

$$
\begin{aligned}
\frac{\partial E}{\partial a_i^{[l]}} &= \sum_{j=1}^{n} \frac{\partial z_j^{[l+1]}}{\partial a_i^{[l]}} \frac{\partial a_j^{[l+1]}}{\partial z_j^{[l+1]}} \frac{\partial E}{\partial a_j^{[l+1]}} \\
&= \sum_{j=1}^{n} w_{ji}^{[l+1]} r_j^{[l+1]}
\end{aligned}
\tag{20}
$$

For Equation 20, we also want to vectorize the calculation. We can first try to convert the summation to dot product:

$$
\frac{\partial E}{\partial a_i^{[l]}} = [w_{1i} \;\; w_{2i} \;\; \dots \;\; w_{ni}] \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}
\tag{21}
$$

We can then arrange $\frac{\partial E}{\partial a_i^{[l]}}$ into a vector:

$$
\begin{bmatrix} \frac{\partial E}{\partial a_1^{[l]}} \\ \frac{\partial E}{\partial a_2^{[l]}} \\ \dots \\ \frac{\partial E}{\partial a_n^{[l]}} \end{bmatrix} = \begin{bmatrix} w_{i=1,j=1} & w_{i=1,j=2} & \dots & w_{1,n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{bmatrix}^{[l+1]} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}^{[l+1]}
\tag{22}
$$

Here, we see that the row index used for w is i, which is not the same for the original definition of w. We can transpose the matrix w to make the row index the same as the original definition of w:

$$
\frac{\partial E}{\partial a^{[l]}} = \left(w^{[l+1]}\right)^T r^{[l+1]}
\tag{23}
$$

Here, $\frac{\partial E}{\partial a^{[l]}}$ and $r^{[l]}$ are vectors, and w is a matrix.

4.3.4. Summary

The following is the complete set of vectorized equations used in the backpropagation algorithm:

$$
\begin{aligned}
\frac{\partial E}{\partial a^{[l]}} &= \left(w^{[l+1]}\right)^T r^{[l+1]} \\
r_j^{[l]} &= \sigma'\left(z_j^{[l]}\right) \frac{\partial E}{\partial a_j^{[l]}} \\
\frac{\partial E}{\partial w^{[l]}} &= r^{[l]} \left(a^{[l-1]}\right)^T \\
\frac{\partial E}{\partial b^{[l]}} &= r^{[l]}
\end{aligned}
\tag{24}
$$

The equation set shows that to update the parameters in the current layer, we need to first calculate $r^{[l+1]}$ of the next layer. And this forms a recursive relationship between layers, enableing us to backward propagate the error from the output layer to the input layer.

5. Result

The source code for this project can be found at here. And with the setup of $784 \to 16 \to 16 \to 10$ and training iteration of 51419 (around 1m 50s training time), this model can reach an accuracy of 0.9208 on the NMIST dataset.