Korean J. Math. **30** (2022), No. 1, pp. 131–146
http://dx.doi.org/10.11568/kjm.2022.30.1.131

GRADIENTS IN A DEEP NEURAL NETWORK AND THEIR PYTHON IMPLEMENTATIONS

Young Ho Park

Abstract. This is an expository article about the gradients in deep neural network. It is hard to find a place where gradients in a deep neural network are dealt in details in a systematic and mathematical way. We review and compute the gradients and Jacobians to derive formulas for gradients which appear in the backpropagation and implement them in vectorized forms in Python.

1. Introduction

This is an expository article about the gradients appearing in neural networks. A *deep neural network* $\boldsymbol{n}_{\boldsymbol{W},\boldsymbol{b}}$ with weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$ consists of an input layer, one or more hidden layers $HL_1, \cdots, HL_{L-1}$ and an output layer.

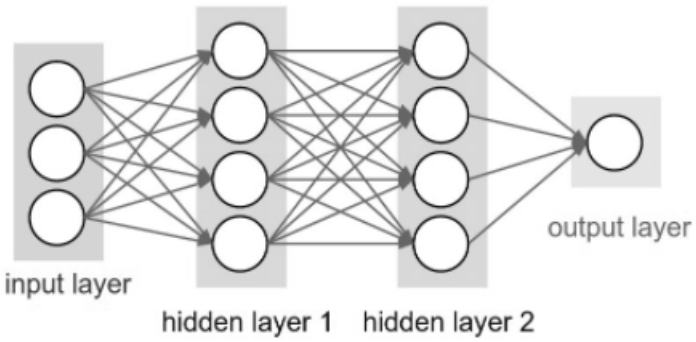


Figure 1. Deep neural networks

Mathematically, each hidden layer or a output layer is a function

$$HL_\ell : \mathbb{R}^{n[\ell-1]} \to \mathbb{R}^{n[\ell]}$$
$$\boldsymbol{a}[\ell-1] \mapsto \boldsymbol{a}[\ell]$$

which is a composition of a linear transformation

$$\boldsymbol{z}[\ell] = \boldsymbol{W}[\ell]\boldsymbol{a}[\ell-1] + \boldsymbol{b}[\ell]$$

Received December 5, 2021. Revised February 22, 2022. Accepted March 1, 2022.
2010 Mathematics Subject Classification: 68T07.
Key words and phrases: gradients, deep neural networks, backpropagations, machine learning.

and an activation function

$$g[\ell] : \mathbb{R}^{n[\ell-1]} \to \mathbb{R}^{n[\ell]}, \quad \boldsymbol{a}[\ell] = g[\ell](\boldsymbol{z}[\ell]).$$

Here $\boldsymbol{W}[\ell] \in \mathrm{Mat}_{n[\ell]\times n[\ell-1]}(\mathbb{R})$ is a *weight* matrix and $\boldsymbol{b}[\ell]$ is a *bias* vector. The most widely used activation function of the hidden layers are relu functions. The structure of the output layer is the same as hidden layers except that the activation function is usually the sigmoid for binary classification or the softmax function for multiclass classification and the identity function for the regression.

Let the training data for the network $\boldsymbol{n}_{\boldsymbol{W},\boldsymbol{b}}$ be given by

$$\{(\mathcal{X}_k, \boldsymbol{y}_k) \mid k = 1, \cdots, N\},$$

where $\mathcal{X}_k \in \mathbb{R}^n$, and $\boldsymbol{y}_k \in \mathbb{R}^m$. $\{\mathcal{X}_k\}$ are samples and $\boldsymbol{y}_k$ is the *target* or *label* of $\mathcal{X}_k$. Let $\boldsymbol{o}_k = \boldsymbol{n}_{\boldsymbol{W},\boldsymbol{b}}(\mathcal{X}_k)$ be the network's output. We will consider the cost function

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{N}\sum_{k=1}^{N} \mathcal{L}(\boldsymbol{o}_k, \boldsymbol{y}_k)$$

defined by the average of the losses $\mathcal{L}(\boldsymbol{o}_k, \boldsymbol{y}_k)$ over the N samples. The loss function $\mathcal{L}$ is usually one of the following:

1. L^2-loss (regression)

$$\mathcal{L}(\boldsymbol{o}, \boldsymbol{y}) = \frac{1}{2}||\boldsymbol{o} - \boldsymbol{y}||_2^2 = \frac{1}{2}(\boldsymbol{o} - \boldsymbol{y}) \cdot (\boldsymbol{o} - \boldsymbol{y}),$$

2. binary cross-entropy loss (binary classification, $m = 1$)

$$\mathcal{L}(\boldsymbol{o}, \boldsymbol{y}) = -[\boldsymbol{y} \log \boldsymbol{o} + (1 - \boldsymbol{y}) \log(1 - \boldsymbol{o})],$$

3. categorical cross-entropy loss (multiclass classification)

$$\mathcal{L}(\boldsymbol{o}, \boldsymbol{y}) = -\sum_{j=1}^{m} y_j \log o_j.$$

We want to find a neural network which maps $\mathcal{X}_k$ as close to $\boldsymbol{y}_k$ as possible and the learning process is to find the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ such that the total cost $J(\boldsymbol{W}, \boldsymbol{b})$ is as small as we want. This is basically done by the gradient descent algorithm:

$$\boldsymbol{W} \longleftarrow \boldsymbol{W} - \alpha\frac{\partial J}{\partial \boldsymbol{W}}, \qquad \boldsymbol{b} \longleftarrow \boldsymbol{b} - \alpha\frac{\partial J}{\partial \boldsymbol{b}},$$

where α is a constant called the *learning rate*. The exact meaning of these gradients will be given later.

The python packages such as TensorFlow and PyTorch automatically compute the gradients according to the structure of the network, and hence the users of such packages do not have to worry about the actual computation of gradients. See [1] for example. However, we mathematicians care very much about how the formulas are derived. One can find thousands of web pages in the internet, which explain the mathematics of the gradients. However, it is hard to find an article which derive the formulas in a systematic way. Furthermore, a lot of web pages use ugly notations and sometimes the wrong formulas. Even a very famous professor gives partially incorrect formulas for gradients, incorrect in the middle by a constant factor but correct in the final formulas (Compare `dZ, dW, db` in [2] with ours in Python Code 4). This is the motivation of this article presenting the exact mathematical computation of gradients

and their python implementation. We hope that this article will be of some help to those who study mathematics for neural networks.

2. Jacobians

Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$ with $\boldsymbol{y} = f(\boldsymbol{x})$. The **Jacobian** of f at a point $\boldsymbol{x} \in \mathbb{R}^n$ is the $m \times n$ matrix of partial derivatives

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \left[\frac{\partial y_i}{\partial x_j}\right] = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & & \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \in M_{m\times n}(\mathbb{R}), \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \ \boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}.$$

Jacobian tells us the relation between the changes in the input and changes in the output

$$f(\boldsymbol{x} + \Delta\boldsymbol{x}) \simeq \boldsymbol{y} + \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\Delta\boldsymbol{x}.$$

Moreover, if we have another function $g : \mathbb{R}^m \to \mathbb{R}^k$ with $\boldsymbol{z} = g(\boldsymbol{y})$, then $g \circ f : \mathbb{R}^n \to \mathbb{R}^k$ and

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}.$$

For a scalar function $g : \mathbb{R}^n \to \mathbb{R}$, $\boldsymbol{y} = g(\boldsymbol{x})$, we define its **gradient** as the column vector

$$\nabla_{\boldsymbol{x}} g = \begin{bmatrix} \frac{\partial g}{\partial x_1} \\ \vdots \\ \frac{\partial g}{\partial x_n} \end{bmatrix}.$$

Notice that the *gradient is the transpose of Jacobian*:

$$\nabla_{\boldsymbol{x}} g = \left[\frac{\partial g}{\partial \boldsymbol{x}}\right]^T.$$

See any calculus text, such as [5], for Jacobians and gradients,

3. Activation functions

We will mainly consider the following activation functions.

- **relu** : The function relu : $\mathbb{R} \to \mathbb{R}$ is defined by

$$\text{relu}(z) = \max(0, z).$$

  We naturally extend relu to matrices as relu : $\text{Mat}_{m\times n}(\mathbb{R}) \to \text{Mat}_{m\times n}(\mathbb{R})$ by

$$\text{relu}([x_{ij}]) = [\text{relu}(x_{ij})].$$

- **sigmoid**: The sigmoid function $\sigma : \mathbb{R} \to \mathbb{R}$ is defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

  The sigmoid also extends to matrices.

- **softmax**: The softmax is a function from $\boldsymbol{s} : \mathbb{R}^m \to (0,1]^m$ defined by

$$\boldsymbol{s}(\boldsymbol{z}) = \left[\frac{e^{z_i}}{\sum_{k=1}^{m} e^{z_k}}\right] \in M_{m\times 1}(\mathbb{R}).$$

  It is a generalization of sigmoid function.

The relu function itself is not differentiable at $z = 0$. However we use a little trick to modify the definition as follows:

$$\text{relu}_\epsilon(x) = \begin{cases} -\frac{2}{\epsilon^3}z^4 - \frac{3}{\epsilon^2}z^3 + z, & z \in [-\epsilon, 0], \\ \text{relu}(z), & \text{elsewhere.} \end{cases}$$

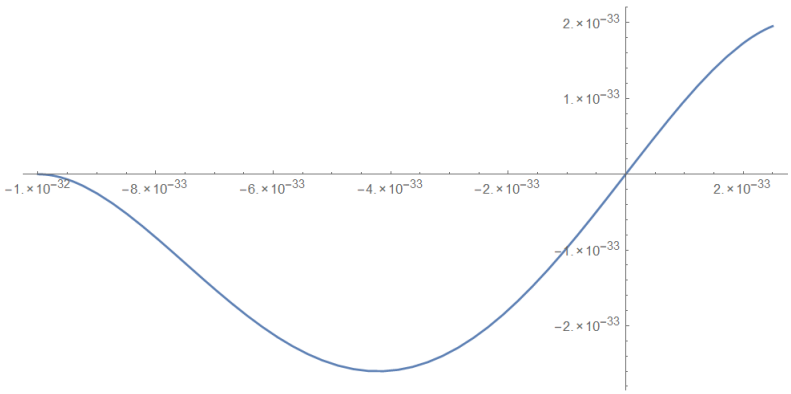


FIGURE 2. Plot of relu_ϵ with $\epsilon = 10^{-32}$

The following is easy to check.

LEMMA 3.1. relu_ϵ *is a differentiable function such that*

$$\text{relu}'_\epsilon(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z \leq -\epsilon. \end{cases}$$

We take ϵ to be smaller than the precision of the computer, so that any $\delta < \epsilon$ will be treated as 0 and then relu and relu_ϵ will be the same functions to the computer. Thus we may replace relu with relu_ϵ and simply put

$$\text{relu}'(z) = \begin{cases} 1, & z \geq 0, \\ 0, & z < 0. \end{cases} \tag{1}$$

In a similar manner, we might choose that $\text{relu}'(z)$ is 1 for $z > 0$ and 0 for $z \leq 0$.

On the other hand, the derivative of the sigmoid activation σ is given by

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)). \tag{2}$$

PYTHON CODE 1. Activation functions and their derivatives are defined for numpy arrays. The softmax activation is used only at the output layer and its derivative is not explicitly required but given in the equation (39).

```
import  numpy as np

def sigmoid(Z):
    A = 1/(1+np.exp(-Z))
    return A

def d_sigmoid(Z):
    s = sigmoid(Z)
    return s * (1 - s)

def relu(Z):
    A = np.maximum(0,Z)
    return A

def d_relu(Z):
    return (Z>=0)*1.0

def softmax(Z):
    Z_exp = np.exp(Z)
    Z_sum = np.sum(Z_exp, axis=1, keepdims=True)
    A = Z_exp/Z_sum
    return A
```

4. Flow of network functions

In order to simplify notations, we fix a hidden layer or the output layer and omit the index ℓ. Output layer will be treated at a later section in details.

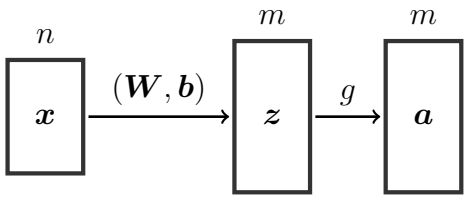


FIGURE 3. Hidden or output layer

Let $\boldsymbol{x} \in \mathbb{R}^n$ be the input vector to the layer, $\boldsymbol{W} = [w_{ij}] \in \mathrm{Mat}_{m\times n}(\mathbb{R})$ be a weight matrix, $\boldsymbol{b} \in \mathbb{R}^m$ be a bias vector and

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \in \mathbb{R}^m \tag{3}$$

be the linear output. Let $g : \mathbb{R} \to \mathbb{R}$ be the activation function in the layer. We then always extends g to $g : \mathbb{R}^m \to \mathbb{R}^m$ in an obvious way.

$$\boldsymbol{a} = g(\boldsymbol{z}) = \begin{bmatrix} g(z_1) \\ \vdots \\ g(z_m) \end{bmatrix}. \tag{4}$$

The case of the softmax activation function will be treated in later section. Let $\boldsymbol{a} = g(\boldsymbol{z})$ be the output of the layer.

Equation (1) can be written componentwise:

$$z_i = \sum_{j=1}^{n} w_{ij}x_j + b_i. \tag{5}$$

Now we compute various Jacobians. From (5), we get

$$\frac{\partial z_i}{\partial x_j} = w_{ij}, \quad \frac{\partial z_i}{\partial b_j} = \delta_{ij}.$$

From (4), we have that $a_i = g(z_i)$, and hence

$$\frac{\partial a_i}{\partial z_j} = g'(z_i)\frac{\partial z_i}{\partial z_j} = g'(z_i)\delta_{ij}.$$

Let S be any real-valued function of $\boldsymbol{z}$. Then $S = S(\boldsymbol{z}(\boldsymbol{W}))$ is a function of $\boldsymbol{W}$. Notice that $\nabla_{\boldsymbol{W}} S$ is a $mn \times 1$ matrix. However, it will be inconvenient with this notation to do the gradient descent algorithm

$$\boldsymbol{W} \longleftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} S.$$

Hence we follow the convention to reshape the gradients:

Convention. The shape of the gradient is the same as the shape of parameters.

Following this convention, we reshape the gradient of S with respect to $\boldsymbol{W}$ as

$$\nabla_{\boldsymbol{W}} S = \left[\frac{\partial S}{\partial w_{ij}}\right] \in M_{m\times n}(\mathbb{R}).$$

Note that this convention can also be applied to vectors, without a contradiction. For example, the convention says that

$$\nabla_{\boldsymbol{x}} g = \begin{bmatrix} \frac{\partial g}{\partial x_1} \\ \vdots \\ \frac{\partial g}{\partial x_n} \end{bmatrix} = \left[\frac{\partial g}{\partial \boldsymbol{x}}\right]^T,$$

which agrees with the usual definition.

Since S is a function of components z_k's of $\boldsymbol{z}$, we have that

$$\frac{\partial S}{\partial w_{ij}} = \sum_{k=1}^{m} \frac{\partial S}{\partial z_k}\frac{\partial z_k}{\partial w_{ij}}. \tag{6}$$

Recalling the equation (5)

$$z_k = w_{k1}x_1 + w_{k2}x_2 + \cdots + w_{kn}x_n + b_k,$$

we get $\frac{\partial z_k}{\partial w_{ij}} = \delta_{ik}x_j$. Thus

$$\frac{\partial S}{\partial w_{ij}} = \sum_{k=1}^{m} \frac{\partial S}{\partial z_k}\delta_{ki}x_j = \frac{\partial S}{\partial z_i}x_j.$$

Therefore,

$$\nabla_{\boldsymbol{W}} S = \left[\frac{\partial S}{\partial w_{ij}}\right] = \left[\frac{\partial S}{\partial z_i} x_j\right] = \begin{bmatrix} \frac{\partial S}{\partial z_1}x_1 & \frac{\partial S}{\partial z_1}x_2 & \cdots & \frac{\partial S}{\partial z_1}x_n \\ \frac{\partial S}{\partial z_2}x_1 & \frac{\partial S}{\partial z_2}x_2 & \cdots & \frac{\partial S}{\partial z_2}x_n \\ \vdots & & & \\ \frac{\partial S}{\partial z_m}x_1 & \frac{\partial S}{\partial z_m}x_2 & \cdots & \frac{\partial S}{\partial z_m}x_n \end{bmatrix}$$
$$= \left[\frac{\partial S}{\partial \boldsymbol{z}}\right]^T \boldsymbol{x}^T = (\nabla_{\boldsymbol{z}} S)\boldsymbol{x}^T.$$

Now we collect various gradients we have computed.

THEOREM 4.1. *For $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, $\boldsymbol{a} = g(\boldsymbol{z})$ and any scalar function $S = S(\boldsymbol{z})$,*

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{W}, \tag{7}$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = I, \tag{8}$$

$$\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} = \operatorname{diag}(g'(\boldsymbol{z})), \tag{9}$$

$$\nabla_{\boldsymbol{W}} S = (\nabla_{\boldsymbol{z}} S)\boldsymbol{x}^T. \tag{10}$$

Here $\operatorname{diag}(g'(\boldsymbol{z})) = \operatorname{diag}(g'(z_1), \cdots, g'(z_m))$ *for* $g : \mathbb{R} \to \mathbb{R}$.

5. Backpropagation through a layer

5.1. Function flow through a network. Consider the function flow of the ℓ-th layer of a network with the input $\boldsymbol{x}[\ell]$, the weight $\boldsymbol{W}[\ell]$, the bias $\boldsymbol{b}[\ell]$ and the activation $g[\ell]$:

$$\boldsymbol{x}[\ell] = \boldsymbol{a}[\ell-1] \in \mathbb{R}^{n[\ell-1]} \quad \text{(input into } l\text{-th layer)}$$
$$\boldsymbol{z}_[\ell] = \boldsymbol{W}[\ell]\boldsymbol{x}[\ell] + \boldsymbol{b}[\ell] \in \mathbb{R}^{n[\ell]} \quad \text{(linear output)}$$
$$\boldsymbol{a}[\ell] = g[\ell](\boldsymbol{z}[\ell]) \in \mathbb{R}^{n[\ell]} \quad \text{(activation)}.$$

We will simplify the notations a little bit by dropping the indices and writing $\boldsymbol{a}[\ell]$ as $\boldsymbol{a}$ and $\boldsymbol{a}[\ell-1]$ as $\boldsymbol{a}_\triangleleft$, etc.

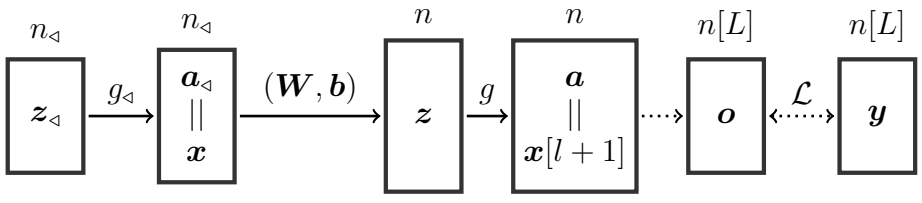


FIGURE 4. $(\ell-1)$-th and ℓ-th layers with output layer

Let $\mathcal{L} = \mathcal{L}(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}, \cdots)$ be any loss function such as the cross-entropy we previously considered. We assume that we already have computed

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = (\nabla_{\boldsymbol{z}} \mathcal{L})^T \in M_{1\times m}(\mathbb{R}).$$

We will inductively compute the gradients

$$\nabla_{\boldsymbol{W}}\mathcal{L}, \quad \nabla_{\boldsymbol{b}}\mathcal{L}, \quad \nabla_{\boldsymbol{z}_\triangleleft}\mathcal{L}.$$

It is customary to write

$$\boldsymbol{\delta} = \nabla_{\boldsymbol{z}}\mathcal{L}, \quad \boldsymbol{\delta}_\triangleleft = \nabla_{\boldsymbol{z}_\triangleleft}\mathcal{L}$$

excerpted from the name *delta rule* [4]. These are the error signals passed down to $\boldsymbol{z}$ when we do the backpropagation. We will find the recurrence relation between $\boldsymbol{\delta}$ and $\boldsymbol{\delta}_\triangleleft$. Equation (10) gives us

$$\nabla_{\boldsymbol{W}}\mathcal{L} = (\nabla_{\boldsymbol{z}}\mathcal{L})\boldsymbol{x}^T = \boldsymbol{\delta}\boldsymbol{x}^T = \boldsymbol{\delta}\boldsymbol{a}_\triangleleft{}^T. \tag{11}$$

The chain rule $\frac{\partial\mathcal{L}}{\partial\boldsymbol{b}} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{z}}\frac{\partial\boldsymbol{z}}{\partial\boldsymbol{b}}$ together with (8) gives us

$$\nabla_{\boldsymbol{b}}\mathcal{L} = \boldsymbol{\delta}. \tag{12}$$

From (7) we obtain

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{z}}\frac{\partial\boldsymbol{z}}{\partial\boldsymbol{x}} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{z}}\boldsymbol{W}.$$

Thus

$$\nabla_{\boldsymbol{x}}\mathcal{L} = \nabla_{\boldsymbol{a}_\triangleleft}\mathcal{L} = \left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}}\right)^T = \boldsymbol{W}^T\boldsymbol{\delta}. \tag{13}$$

By the chain rule again, we obtain

$$\begin{aligned}\boldsymbol{\delta}_\triangleleft^T &= \frac{\partial\mathcal{L}}{\partial\boldsymbol{z}_\triangleleft} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{a}_\triangleleft}\frac{\partial\boldsymbol{a}_\triangleleft}{\partial\boldsymbol{z}_\triangleleft} \\ &= \left(\boldsymbol{\delta}^T\boldsymbol{W}\right)\begin{bmatrix} g_\triangleleft'(z_{\triangleleft 1}) & 0 & \cdots & 0 \\ 0 & g_\triangleleft'(z_{\triangleleft 2}) & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & (g_\triangleleft'(z_{\triangleleft n}) \end{bmatrix},\end{aligned}$$

which implies

$$\boldsymbol{\delta}_\triangleleft = \begin{bmatrix} g_\triangleleft'(z_{\triangleleft 1}) & 0 & \cdots & 0 \\ 0 & g_\triangleleft'(z_{\triangleleft 2}) & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & (g_\triangleleft'(z_{\triangleleft n}) \end{bmatrix}\left(\boldsymbol{W}^T\boldsymbol{\delta}\right). \tag{14}$$

Now $\boldsymbol{W}^T\boldsymbol{\delta}$ has shape $(n\times m)(m\times 1) = (n\times 1)$, and thus it is a column vector. By noticing that

$$\operatorname{diag}(d_1, d_2, \cdots, d_n)\cdot(c_1, c_2, \cdots, c_n)^T = (d_1c_1, d_2c_2, \cdots, d_nc_n)^T \in \operatorname{Mat}_{n\times 1}(\mathbb{R})$$

and using the notation $g(\boldsymbol{z}) = (g(z_1), \cdots, g(z_n))^T$ as before, we obtain

$$\boldsymbol{\delta}_\triangleleft = g_\triangleleft'(\boldsymbol{z}_\triangleleft) *_\cdot \left(\boldsymbol{W}^T\boldsymbol{\delta}\right) = \left(\boldsymbol{W}^T\boldsymbol{\delta}\right) *_\cdot g_\triangleleft'(\boldsymbol{z}_\triangleleft) \tag{15}$$

where $*_\cdot$ denotes the componentwise multiplication. Now we have proved the following theorem:

THEOREM 5.1. *For any scalar function $\mathcal{L} = \mathcal{L}(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}, \cdots)$ with $\boldsymbol{\delta} = \nabla_{\boldsymbol{z}}\mathcal{L}$ and $\boldsymbol{\delta}_\triangleleft = \nabla_{\boldsymbol{z}_\triangleleft}\mathcal{L}$, we have that*

$$\boldsymbol{\delta}_\triangleleft = g_\triangleleft'(\boldsymbol{z}_\triangleleft) *_\cdot (\boldsymbol{W}^T\boldsymbol{\delta}).$$

Moreover,

$$\nabla_{\boldsymbol{a}_\triangleleft}\mathcal{L} = \boldsymbol{W}^T\boldsymbol{\delta},$$
$$\boldsymbol{\delta} = g'(\boldsymbol{z}) *_. \nabla_{\boldsymbol{a}}\mathcal{L},$$
$$\nabla_{\boldsymbol{W}}\mathcal{L} = \boldsymbol{\delta}{\boldsymbol{a}_\triangleleft}^T,$$
$$\nabla_{\boldsymbol{b}}\mathcal{L} = \boldsymbol{\delta}.$$

If we use the layer numbers $\ell = 1, \cdots, L$ (the 0-th layer is the input layer), the identities become

$$\nabla_{\boldsymbol{a}[\ell-1]}\mathcal{L} = \boldsymbol{W}[\ell]^T\boldsymbol{\delta}[\ell], \tag{16}$$

$$\boldsymbol{\delta}[\ell-1] = g[\ell-1]'(\boldsymbol{z}[\ell-1]) *_. (\boldsymbol{W}[\ell]^T\boldsymbol{\delta}[\ell]), \tag{17}$$

$$\nabla_{\boldsymbol{W}[\ell]}\mathcal{L} = \boldsymbol{\delta}[\ell]\boldsymbol{a}[\ell-1]^T, \tag{18}$$

$$\nabla_{\boldsymbol{b}[\ell]}\mathcal{L} = \boldsymbol{\delta}[\ell]. \tag{19}$$

Here, $\boldsymbol{a}[0]$ denotes a sample input data in $\{\mathcal{X}_k\}$.

5.2. Vectorization. Now we vectorize all these results for a fixed ℓ-th layer. Vectorization is a style of computer programming where operations are applied to whole arrays instead of individual samples. The numpy package automatically vectorize many numpy operations. Even though these activation functions are defined for real numbers, numpy automatically vectorize these functions and we can apply relu to numpy arrays. Utilizing this property, we can simplify the network computation and get simple formulas for gradients.

As earlier, the training dataset is given by $(\mathcal{X}_k, \boldsymbol{y}_k)$ for $k = 1, \cdots, N$. Each sample $\mathcal{X}_k$ flows through the each layer and gives $\boldsymbol{z}_k$'s, $\boldsymbol{a}_k$'s for each layer until it produces the network output $\boldsymbol{o}_k$ at the output layer. The vectorization means to pack up the input dataset into a big matrix

$$\mathcal{X} = [\mathcal{X}_1\ \mathcal{X}_2\ \cdots\ \mathcal{X}_N] \in \mathrm{Mat}_{n\times N}(\mathbb{R})$$

whose columns are samples $\mathcal{X}_k$, $k = 1, \cdots, N$ and process it in a unified fashion.

Remark 5.2. However, we must note that in this current world of data science, it is customary that data samples are packed into rows of the dataset under the influence of engineers, which makes us to take transposed versions of our formulas. Refer to [1] for example and for generalities of the machine learning.

We use the capital letters $\boldsymbol{Z} = \boldsymbol{Z}[\ell]$ and $\boldsymbol{A} = \boldsymbol{A}[\ell]$ to denote the packages of linear outputs and activation outputs at the ℓ-th layer (dropping the index ℓ temporarily)

$$\boldsymbol{Z} = [\boldsymbol{z}_1\ \boldsymbol{z}_2\ \cdots\ \boldsymbol{z}_N], \quad \boldsymbol{A} = [\boldsymbol{a}_1\ \boldsymbol{a}_2\ \cdots\ \boldsymbol{a}_N] \in \mathrm{Mat}_{m\times N}(\mathbb{R})$$

where

$$\boldsymbol{z}_k = \boldsymbol{W}\boldsymbol{a}_{\triangleleft,k} + \boldsymbol{b}, \quad \boldsymbol{a}_k = g(\boldsymbol{z}_k)$$

corresponding to the initial sample $\mathcal{X}_k$. Notice that the parameters $\boldsymbol{W} = \boldsymbol{W}[\ell]$ and $\boldsymbol{b} = \boldsymbol{b}[\ell]$ are independent on $\boldsymbol{Z}, \boldsymbol{A}$, but only dependent on the layers. Weights and biases are updated after all inputs $\mathcal{X}_k$ are processed (under the assumption that we use the batch gradient descent algorithm). The total cost function is

$$J(\boldsymbol{W}, \boldsymbol{b}, \cdots) = \frac{1}{N}\sum_{k=1}^{N}\mathcal{L}_k(\boldsymbol{W}, \boldsymbol{b}, \cdots),$$

where $\mathcal{L}_k(\boldsymbol{W},\boldsymbol{b}) = \mathcal{L}(\boldsymbol{o}_k,\boldsymbol{y}_k,\boldsymbol{W},\boldsymbol{b})$ is the individual loss from an input $\mathcal{X}_k$. As before, we write

$$\boldsymbol{\delta}_k = \nabla_{\boldsymbol{z}_k}\mathcal{L} \in \mathrm{Mat}_{n\times 1}(\mathbb{R})$$

for $k = 1,\cdots,N$ and

$$\boldsymbol{\Delta} = \nabla_{\boldsymbol{Z}}\mathcal{L}.$$

More precisely,

$$\boldsymbol{\Delta} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_{11}} & \cdots & \frac{\partial \mathcal{L}}{\partial z_{N1}} \\ \vdots & & \vdots \\ \frac{\partial \mathcal{L}}{\partial z_{1n}} & \cdots & \frac{\partial \mathcal{L}}{\partial z_{Nn}} \end{bmatrix} = [\boldsymbol{\delta}_1\ \boldsymbol{\delta}_2\ \cdots\ \boldsymbol{\delta}_N].$$

where

$$\boldsymbol{Z} = \begin{bmatrix} z_{11} & \cdots & z_{N1} \\ \vdots & & \vdots \\ z_{1n} & \cdots & z_{Nn} \end{bmatrix} \in \mathrm{Mat}_{n\times N}(\mathbb{R})$$

($\boldsymbol{\Delta}$ and $\boldsymbol{Z}$ have the same shape.) Notice that $\boldsymbol{\Delta} \neq \nabla_{\boldsymbol{Z}}J$. In fact,

$$\nabla_{\boldsymbol{z}_k}J = \frac{1}{N}\sum_{k'=1}^{N}\nabla_{\boldsymbol{z}_k}\mathcal{L}_{k'} = \frac{1}{N}\nabla_{\boldsymbol{z}_k}\mathcal{L}_k = \frac{1}{N}\boldsymbol{\delta}_k,$$

so that

$$\nabla_{\boldsymbol{Z}}J = \frac{1}{N}\nabla_{\boldsymbol{Z}}\mathcal{L} = \frac{1}{N}\boldsymbol{\Delta}.$$

Similarly,

$$\nabla_{\boldsymbol{A}_\lhd}J = [\nabla_{\boldsymbol{a}_{\lhd 1}}J\ \cdots\ \nabla_{\boldsymbol{a}_{\lhd N}}J] = \frac{1}{N}[\boldsymbol{W}^T\boldsymbol{\delta}_1\ \cdots\ \boldsymbol{W}^T\boldsymbol{\delta}_N] = \frac{1}{N}\boldsymbol{W}^T\boldsymbol{\Delta} = \boldsymbol{W}^T(\nabla_{\boldsymbol{Z}}J).$$

Now the gradient of $\boldsymbol{W}$ is computed as follow:

$$\begin{aligned}\nabla_{\boldsymbol{W}}J &= \frac{1}{N}\sum_{k=1}^{N}\nabla_{\boldsymbol{W}}\mathcal{L}_k = \frac{1}{N}\sum_{k=1}^{N}\boldsymbol{\delta}_k\boldsymbol{a}_{\lhd k}^T \\ &= \frac{1}{N}[\boldsymbol{\delta}_1\ \boldsymbol{\delta}_2\ \cdots\ \boldsymbol{\delta}_N]\begin{bmatrix}\boldsymbol{a}_{\lhd 1}^T \\ \boldsymbol{a}_{\lhd 2}^T \\ \vdots \\ \boldsymbol{a}_{\lhd N}^T\end{bmatrix} = \frac{1}{N}\boldsymbol{\Delta}\boldsymbol{A}_\lhd^T = (\nabla_{\boldsymbol{Z}}J)\boldsymbol{A}_\lhd^T.\end{aligned}$$

For the gradient $\nabla_{\boldsymbol{b}}J$, we use the equation (12) to obtain

$$\nabla_{\boldsymbol{b}}J = \frac{1}{N}\sum_{k=1}^{N}\nabla_{\boldsymbol{b}}\mathcal{L}_k = \frac{1}{N}\sum_{k=1}^{N}\boldsymbol{\delta}_k = \sum_{k=1}^{N}\nabla_{\boldsymbol{z}_k}J, \tag{20}$$

which is the sum of the columns of $\nabla_{\boldsymbol{Z}}J$. We have finally proved the following formulas about the gradients we need for the backpropagations.

Remark 5.3. For a function $J = J(\mathbf{T})$, let us use the notation

$$d\mathbf{T} = \nabla_{\mathbf{T}}J.$$

This notation is useful when we carry out the python coding.

THEOREM 5.4. *For $\ell = 1, \cdots, L$, let $\boldsymbol{\Delta}[\ell] = \nabla_{\boldsymbol{Z}[\ell]}\mathcal{L}$. Then*

$$d\boldsymbol{Z}[\ell] = \frac{1}{N}\boldsymbol{\Delta}[\ell] \tag{21}$$

and the vectorized gradients at the ℓ-layer for the backpropagation are

$$d\boldsymbol{A}[\ell-1] = \boldsymbol{W}[\ell]^T d\boldsymbol{Z}[\ell] \tag{22}$$

$$d\boldsymbol{Z}[\ell-1] = g'(\boldsymbol{Z}[\ell-1]) *_{\cdot} d\boldsymbol{A}[\ell-1] \tag{23}$$

$$d\boldsymbol{W}[\ell] = d\boldsymbol{Z}[\ell]\boldsymbol{A}[\ell-1]^T \tag{24}$$

$$d\boldsymbol{b}[\ell] = \textit{sum of columns of } d\boldsymbol{Z}[\ell] \tag{25}$$

PYTHON CODE 2. (propagation at a relu layer)

We assume that the activation function is given by relu at the given i-th layer ($1 \le i < L$). The letters $\boldsymbol{Z}, \boldsymbol{A}, \boldsymbol{W}, \boldsymbol{b}$ are written as `Z`, `A`, `W`, `b` at the code.

```
Z[i] = W[i] @ A[i-1] + b[i]
A[i] = relu(Z[i])

dA[i] = W[i+1].T @ dZ[i+1]
dZ[i] = d_relu(Z[i]) * dA[i]
dW[i] = dZ[i] @ A[i-1].T
db[i] = np.sum(dZ[i], axis=1, keepdims=True)
```

6. At the output layer

We consider the function flow of the output layer (L-th layer), where $\boldsymbol{x} = \boldsymbol{a}_{L-1}$:

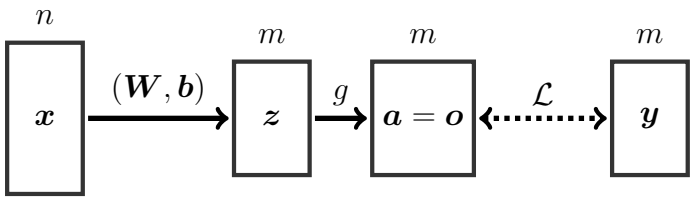


FIGURE 5. Last(L-th) layer

6.1. Linear logistic regression. Suppose we are given a set of data $\{(\mathcal{X}_k, \boldsymbol{y}_k) \mid k = 1, \cdots, N\}$. We want to find a 'good' linear function (called hypothesis)

$$h(\mathcal{X}) = \boldsymbol{W}\mathcal{X} + \boldsymbol{b}$$

such that $h(\mathcal{X}_k) \approx \boldsymbol{y}_k$ for all k, where $\boldsymbol{W} \in \mathrm{Mat}_{1\times n}(\mathbb{R})$ and $\boldsymbol{b} \in \mathbb{R}$. Actually, the good function means that the total error

$$J = \frac{1}{N}\sum_{k=1}^{N} \mathcal{L}(h(\mathcal{X}_k), \boldsymbol{y}_k)$$

is as small as possible.

$$\mathcal{L}(h(\mathcal{X}), \boldsymbol{y}) = \frac{1}{2}\|h(\mathcal{X}) - \boldsymbol{y}\|^2 \tag{26}$$

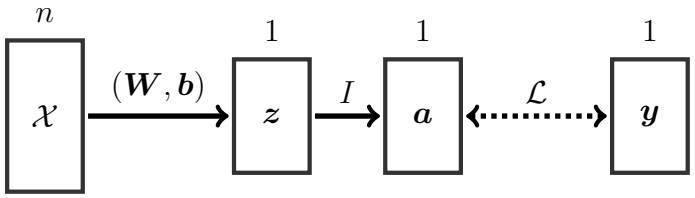


FIGURE 6. Neural network for a logistic regression

is the L^2-loss (or mean squared error) for the data sample $(\mathcal{X}, \boldsymbol{y})$. In this way, we can use $h(\mathcal{X})$ to predict the value of $\boldsymbol{y}$ at a new input $\mathcal{X}$.

In this case, the network has no hidden layer, and the activation function is the identity so that $\boldsymbol{z} = \boldsymbol{a} = \boldsymbol{o} = h(\mathcal{X})$. Thus

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}} = (\boldsymbol{a} - \boldsymbol{y})^T$$

from (26) and hence

$$\boldsymbol{\delta} = \nabla_{\boldsymbol{z}} \mathcal{L} = \boldsymbol{a} - \boldsymbol{y} \tag{27}$$

$$\nabla_{\boldsymbol{W}} \mathcal{L} = \boldsymbol{\delta} \boldsymbol{x}^T \tag{28}$$

$$\nabla_{\boldsymbol{b}} \mathcal{L} = \boldsymbol{\delta} \tag{29}$$

In vectorized notations, we have that

$$d\boldsymbol{Z} = \frac{1}{N}(\boldsymbol{A} - \boldsymbol{Y}) \tag{30}$$

$$d\boldsymbol{W} = d\boldsymbol{Z} \mathcal{X}^T \tag{31}$$

$$d\boldsymbol{b} = \text{sum of columns of } d\boldsymbol{Z} \tag{32}$$

where $\boldsymbol{Y} = [\boldsymbol{y}_1 \ \cdots \ \boldsymbol{y}_N]$.

PYTHON CODE 3. The training data is given by (X, y) with $X \in \text{Mat}_{n \times N}(\mathbb{R})$, $y \in \text{Mat}_{1 \times N}(\mathbb{R})$.

```
# Linear Regression

# After reading in the data (X,y):
#  X = matrix of shape (n, N) whose columns are samples
#  y = array of shape (1,N)

# initialization
n, N = X.shape
W = np.random.rand(1,n)*0.01
b = 0

def forward_propagation(X, W, b):
    Z = W @ X + b
    A = Z   # not necessary in practice
    return A
```

```
def back_propagation(X, y, W, b)
    A = forward_propagation(X, W, b)
    N = X.shape[1]
    dZ = 1/N * (A - y)
    dW = dZ @ X.T
    db = np.sum(dZ, axis=1, keepdims=True)
    return dW, db

def update_params(W, b, dW, db, learning_rate=0.1):
    W -= learning_rate * dW
    b -= learning_rate * db
    return W, b

def cost(A,y):
    return np.sum(((A-y)**2)) / (2*N)

# Gradient Descent Algorithm
# hyperparameters
n_iter = 300
learning_rate = 0.01

cost_history = []
for i in range(n_iter):
    A = forward_propagation(X, W, b)
    dW, db = backward_propagation(X, y, W, b)
    W, b = update_params(W, b, dW, db, learning_rate)
    cost_history.append(cost(A,y))

print(f'Final cost : {cost_history[-1]}')
print(f'W = {W}')
print(f'b = {b}')
```

6.2. Binary classification. In this case, notice that $\boldsymbol{z}, \boldsymbol{a}, \boldsymbol{y} \in \mathbb{R}$.

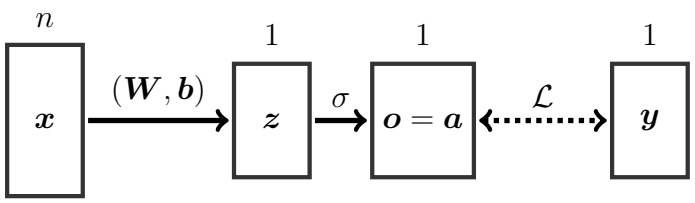


FIGURE 7. Last layer with sigmoid activation

We have the sigmoid activation function $g = \sigma$ and the binary cross-entropy loss

$$\mathcal{L}(\boldsymbol{a}, \boldsymbol{y}) = -\Big[\boldsymbol{y} \log \boldsymbol{a} + (1 - \boldsymbol{y}) \log(1 - \boldsymbol{a})\Big].$$

From $\boldsymbol{a} = \sigma(\boldsymbol{z})$, we know that $\boldsymbol{a}'(\boldsymbol{z}) = \boldsymbol{a}(1-\boldsymbol{a})$. Thus

$$\frac{d\mathcal{L}}{d\boldsymbol{z}} = -\boldsymbol{y}\frac{\boldsymbol{a}(1-\boldsymbol{a})}{\boldsymbol{a}} - (1-\boldsymbol{y})\frac{-\boldsymbol{a}(1-\boldsymbol{a})}{1-\boldsymbol{a}} = \boldsymbol{a} - \boldsymbol{y}.$$

In a vectorized notation, we therefore have that

$$\boldsymbol{\delta} = \nabla_{\boldsymbol{Z}}\mathcal{L} = \boldsymbol{a} - \boldsymbol{y}, \tag{33}$$

$$\nabla_{\boldsymbol{W}}\mathcal{L} = \Delta\boldsymbol{x}^T, \tag{34}$$

$$\nabla_b\mathcal{L} = \boldsymbol{\Delta}. \tag{35}$$

Notice that they are exactly the same as the linear logistic regression case. The vectorized gradients at this last L-th layer are

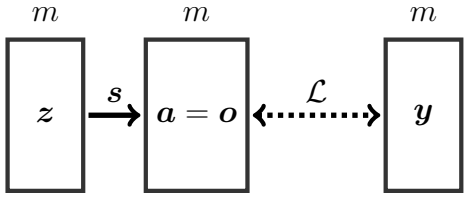
$$d\boldsymbol{Z}[L] = \frac{1}{N}(\boldsymbol{A}[L] - \boldsymbol{Y}), \tag{36}$$

$$d\boldsymbol{W}[L] = d\boldsymbol{Z}[L]\boldsymbol{A}[L-1]^T, \tag{37}$$

$$d\boldsymbol{b}[L] = \text{sum of columns of } d\boldsymbol{Z}[L]. \tag{38}$$

6.3. Softmax output. Suppose that the softmax activation function $\boldsymbol{o} = \boldsymbol{s}(\boldsymbol{z})$ is used at the last L-th layer. It is defined to be the function from $\mathbb{R}^m \to (0,1]^m$ for classifying the inputs into m objects:

$$\boldsymbol{o} = \boldsymbol{s}(\boldsymbol{z}) = \left[\frac{e^{z_i}}{\sum_{k=1}^n e^{z_k}}\right] \in M_{m\times 1}(\mathbb{R}).$$



In this case, we assume that the labels $\{\boldsymbol{y}_k\}$ are given by the *one-hot vectors*. For example, the class 0 is represented by $\boldsymbol{y}_k = (1, 0, \cdots, 0)^T$.

We need to compute $\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{z}} = \left[\frac{\partial o_i}{\partial z_j}\right]$. Since

$$o_i = \frac{e^{z_i}}{e^{z_1} + \cdots + e^{z_i} + \cdots + e^{z_m}}$$

we have that

$$\frac{\partial o_i}{\partial z_j} = \frac{\delta_{ij}e^{z_i}(\sum e^{z_k}) - e^{z_i}e^{z_j}}{(\sum e^{z_k})^2} = \begin{cases} o_i(1-o_j), & i = j \\ -o_i o_j, & i \neq j \end{cases} = o_i(\delta_{ij} - o_j).$$

That is,

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{z}} = [o_i(\delta_{ij} - o_j)] = \begin{bmatrix} o_1(1-o_1) & -o_1o_2 & \cdots & -o_1o_m \\ -o_2o_1 & o_2(1-o_2) & \cdots & -o_2o_m \\ \vdots & & & \\ -o_mo_1 & -o_mo_2 & \cdots & o_m(1-o_m) \end{bmatrix}. \tag{39}$$

The categorical cross-entropy function for the softmax activation is defined to be

$$\mathcal{L}(\boldsymbol{o}, \boldsymbol{y}) = -\sum_{i=1}^m y_i \log(o_i), \quad \boldsymbol{o} = \boldsymbol{s}(\boldsymbol{z})$$

Recalling that $\sum_i o_i = 1$, we get

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial z_j} &= -\sum_i y_i \frac{\partial \log(o_i)}{\partial z_j} = -\sum_i o_i \frac{1}{o_i}\frac{\partial o_i}{\partial z_j} \\ &= -\sum_i o_i \frac{1}{o_i} \cdot o_i(\delta_{ij} - o_j) = -\sum_i o_i(\delta_{ij} - o_j) \\ &= \sum_{i \neq j} y_i o_j + y_j(-1 + o_j) = (\sum_i y_i)o_j - y_j = o_j - y_j\end{aligned}$$

Hence the final answer is again

$$\boldsymbol{\delta} = \boldsymbol{o} - \boldsymbol{y}. \tag{40}$$

THEOREM 6.1. *For each of the following output layer of*

1. *the linear logistic regression (identity activation, L^2-loss),*
2. *the binary classification(sigmoid activation, cross-entropy loss)*
3. *the multi-class classification (softmax activation, cross-entropy loss),*

we have the same gradient

$$\boldsymbol{\delta} = \nabla_{\boldsymbol{z}} \mathcal{L} = \boldsymbol{o} - \boldsymbol{y}.$$

at the last layer for single data input.

We consider the deep neural network for the classification into m-classes ($m \geq 3$). The training data is given by (X, Y) with $X \in \mathrm{Mat}_{n\times N}(\mathbb{R})$, $Y \in \mathrm{Mat}_{m\times N}(\mathbb{R})$. We assume that the activation at the hidden layers are given by relu function and activation at the output layer is given by the softmax. We only give the forward and backward propagations using Numpy.

PYTHON CODE 4.

```
# multiclass classification (softmax output)

# After reading in the data (X,y):
# X = numpy array of shape (n, N) whose columns are samples.
# Y = numpy array of shape (m,N) whose columns are one-hot vectors.
# W, b are list of weights and biases at the layers.
# dW, db are list of gradients of the cost with respect to weights
#    and biases.
# Z, A are list of linear outputs and activation outputs at the layers.
# dZ, dA are lists of gradients of the cost with respect to Z and A
# Set A[0] = X

def forward_propagation(X, W, b):
    for i in range(1, L):
        Z[i] = W[i] @ A[i-1] + b[i]
        A[i] = relu(Z[i])

    Z[L] = W[L] @ A[L-1] + b[L]
    A[L] = softmax(Z[L])
    return Z, A

def back_propagation(X, Y, W, b)
    Z, A = forward_propagation(X, W, b)
```

```
    N = X.shape[1]

    dZ[L] = 1/N * (A[L] - Y)
    dW[L] = dZ[L] @ A[L-1].T
    db[L] = np.sum(dZ[L], axis=1, keepdims=True)

    for i in range(L-1,0,-1):
        dA[i] = Ws[i+1].T @ dZ[i+1]
        dZ[i] = d_relu(Z[i]) * dA[i]
        dW[i] = dZ[i] @ A[i-1].T
        db[i] = np.sum(dZ[i], axis=1, keepdims=True)
    return dW, db
```

Finally, we carry out an experiment to calculate the gradients using our formulas given in Theorem 5.4 and check if they agree with the gradients computed by autograd in PyTorch. The result can be found in [3].

References

[1] A. Géron, Hands-on Machine Learning with Scikit-Learn & TensorFlow (핸즈온 머신러닝), Hanbit Media, 2018
[2] Andrew Ng, Note on neural network and deep learning, https://github.com/ashishpatel26/Andrew-NG-Notes, accessed February 23, 2022
[3] Y.H. Park, Verifying gradient formulas by PyTorch, https://deepmath.kangwon.ac.kr/∼yhpark/verify_gradients.pdf, accessed February 23, 2022
[4] T. Rashid, Make your own neural network (신경망 첫걸음), Hanbit Media, 2017
[5] J. Stewart, Calculus, Books-Hill, 2021

Young Ho Park
Department of Mathematics, Kangwon National University,
Chuncheon 24341, Korea.
E-mail: yhpark@kangwon.ac.kr