

## Chapter 5

# Basics of Neural Networks

The term “neural network” has its origins in attempts to find mathematical representation of information processing in biological systems. From the perspective of practical applications of machine learning and pattern recognition, however, biological realism would impose entirely unnecessary constraints. Therefore, we will treat neural networks as efficient models for statistical learning.

This chapter will cover the basics of neural networks. A logistic regression (discussed in the previous chapter) is a typical element in neural networks, called neuron. We will re-visit logistic regression and explain the principle of neural networks from a perspective of extended or generalized logistic regression. We represent the parameters of a neural network by matrices and vectors. The data processed by the neural network are also represented by matrices or vectors. To train a neural network by gradient descent algorithm, we need to compute the derivatives (i.e., gradients) of the cost function, with respect to parameters. Derivative backpropagation is an efficient way to compute the gradients.

The detailed implementation of neural networks from scratch will be presented through two examples: binary classification on synthetic data and multi-class classification on handwritten digits.

In this chapter you will learn

- Neural network architecture and its representation in math
- Activation functions in neural networks
- Backpropagation for training neural networks
- Softmax and cross entropy loss for multi-classification tasks
- Python program for neural networks from scratch

## 5.1 A simplest Neural Network: A Logistic Regression Unit

A neural network generally comprises multiple layers of nodes, known as *neurons*. In this text, the relationship between the neuron in machine learning and the neuron in biological literatures is not our concern. We will see that a neuron in machine learning is a logistic regression unit. In this section, we will re-visit logistic regression model with new notations and terminology which will be used to discuss general issues of neural networks in the subsequent text. The methodology we use to train a logistic unit can be easily extended to train a typical neural network.

The purpose of logistic regression is to estimate the conditional probability of the label  $y$  given an observation  $\mathbf{x} \in \mathbb{R}^{n_x}$ , where  $n_x$  is the number of features, i.e.  $p(y = 1|\mathbf{x})$ , and a label “1” will be predicted if the probability is more than 0.5, otherwise a label “0” will be predicted. Specifically, a logistic regression learns a weight vector  $W \in \mathbb{R}^{n_x}$  and a scalar  $b \in \mathbb{R}$  from a training dataset, and then estimate the conditional probability for a new value of  $\mathbf{x}$  as

$$\hat{y} = p(y = 1|\mathbf{x}) = \sigma(z) = \sigma(W^T \mathbf{x} + b) \quad (5.1)$$

where  $z = \sum_{j=1}^{n_x} w_j x_j + b$  is a linear combination of input features with a bias  $b$ ,  $\sigma(\cdot)$  is the sigmoid activation function, shown in Fig.5.1, defined by

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad (5.2)$$

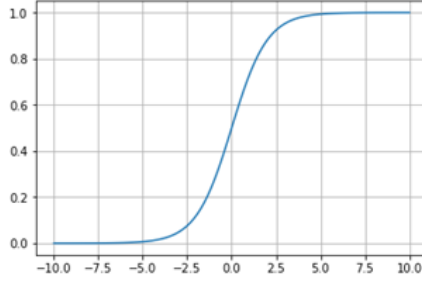


Fig. 5.1 Sigmoid function

To learn the optimal parameters  $W$  and  $b$ , we first define a cost function, and then minimize the cost function with respect to  $W$  and  $b$ . The cost function of logistic regression, for a single data example  $(\mathbf{x}, y)$ , is defined as the cross entropy loss

$$L(\hat{y}, y) = -y \ln(\hat{y}) - (1 - y) \ln(1 - \hat{y}) \quad (5.3)$$

Thus, the average cost over  $m$  data samples is given by

$$J(W, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \ln(\hat{y}^{(i)}) + (1 - y^{(i)}) \ln(1 - \hat{y}^{(i)})] \quad (5.4)$$

where

$$\hat{y}^{(i)} = \sigma(W^T x^{(i)} + b) \quad (5.5)$$

A gradient descent algorithm is used to find  $W$  and  $b$  that minimize the cost function  $J(W, b)$ :

Repeat {

$$W := W - \alpha \frac{dJ(W,b)}{dW} \quad (5.6.a)$$

$$b := b - \alpha \frac{dJ(W,b)}{db} \quad (5.6.b)$$

}

where  $\alpha$  is the learning rate, a hyperparameter.

The logistic regression can be viewed as a special neural network that has only one neuron, illustrated in Fig.5.2. Multiple logistic regression units can be used to construct a multi-layer neural network, shown in Fig.5.3, which includes multiple layers, and each layer includes multiple logistic regression units. The outputs of the previous layers serve as the inputs of units in the present layer. We will discuss the mathematical representation of neural networks later.

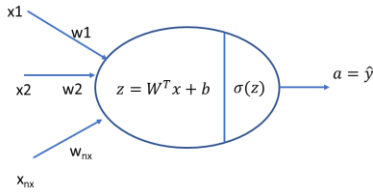


Fig.5.2 A logistic regression unit

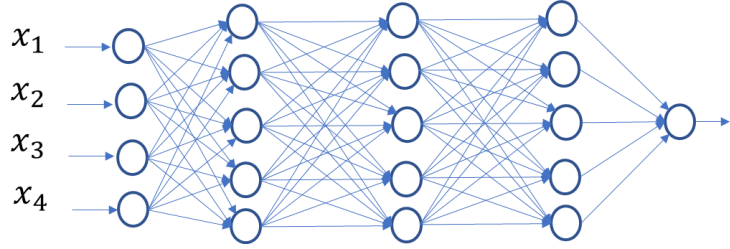


Fig.5.3 An example of multi-layer neural network

## 5.2 From Regression to Neural Networks

Before giving the mathematical representation of neural networks, let's introduce the core idea of neural networks by generalizing the concepts of linear and logistic regressions presented in previous chapters. The development of neural network concepts from regression will help us deeply understand the key characteristics of neural networks. The linear models for regression and classification discussed in Chapter 3 and 4, are based on linear combinations of input features and take the form

$$\hat{y}(x; w) = f\left(\sum_{j=1}^{n_x} w_j x_j + w_0\right) = f\left(\sum_{j=0}^{n_x} w_j x_j\right) \quad (5.7)$$

where  $x_j, j = 1, 2, \dots, n_x$  are input features,  $x_0 = 1$ ,  $f(\cdot)$  is a nonlinear activation function (e.g. sigmoid function) in the case of classification and is the identity in the case of linear regression.  $w_j$  are weight parameters in the model. The semicolon in  $\hat{y}(x; w)$  separates the variables and parameters. However, the resulting fitting curve for linear regression or the resulting decision boundary for classification is linear with respect to input features.

To learn a nonlinear relationship in the case of curve fitting, we can define the curve in the form of linear combination of fixed nonlinear basis functions  $\phi_j(\mathbf{x})$  of input features, instead of a linear combination of input features,

$$\hat{y}(\mathbf{x}; w) = \sum_{j=0}^M w_j \phi_j(\mathbf{x}) \quad (5.8)$$

For example,  $\hat{y}(\mathbf{x}; w) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$ , with  $\{\phi_j(\mathbf{x})\} = \{1, x_1, x_2, x_1 x_2, x_1^2, x_2^2\}$ , can fit a quadratic surface in a two-feature regression. In general,  $\{\phi_j(\mathbf{x}), j = 1, 2, \dots, M\}$  is called a set of *basis functions*, or *kernels*. Similarly, in the case of classification, if we replace the linear combination of input features with the linear combination of nonlinear basis function  $\phi_j(\mathbf{x})$ ,

$$\hat{y}(\mathbf{x}; w) = f\left(\sum_{j=0}^M w_j \phi_j(\mathbf{x})\right) \quad (5.9)$$

then the decision boundaries are nonlinear with the input features. A nonlinear decision boundary obviously can classify more complex data pattern, and thus being more powerful compared to linear decision boundaries. For example,  $\hat{y}(\mathbf{x}; w) = \sigma(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2)$ , with  $\{\phi_j(\mathbf{x})\} = \{1, x_1, x_2, x_1^2, x_2^2\}$ , can learn a circle decision boundary.

So far the nonlinear basis functions,  $\phi_j(\mathbf{x})$ , are fixed for a particular problem, and a training process is to fit the parameters  $w_j$  using a training dataset. Our goal here is to extend this model (5.9) by

making the basis functions  $\phi_j(\mathbf{x})$  depend on parameters and then to allow these parameters to be adjusted, along with the parameters  $w_j$ , during training. To achieve this, we construct  $M$  linear logistic regression units to generate  $M$  basis functions,  $\phi_j(\mathbf{x})$  for (5.9) (note that  $\phi_0(\mathbf{x})$  is always equal to 1 so that  $w_0$  is a bias)

$$z_j = \sum_{i=0}^{n_x} w_{ji} x_i \quad (5.10)$$

$$\phi_j(\mathbf{x}; \mathbf{w}_{j\cdot}) = a_j = h(z_j) \quad (5.11)$$

where  $j=1,2,\dots,M$ , and in the notation  $w_{ji}$  and  $z_j$ ,  $j$  indicates that the parameter or the quantity is associated with the  $j$ th basis function, and  $i$  indicates that the parameter is the weight for input feature  $x_i$ . Thus  $z_j$ ,  $j=1,2,\dots,M$ , are the linear combinations of input features. The basis function  $\phi_j$  is generated by passing the linear combination  $z_j$  through a differentiable nonlinear activation function  $h(\cdot)$ , for example a sigmoid function. The quantities  $a_j$  are known as *activation*, which is the output of activation function. These activations, used as the basis functions in (5.9), are linearly combined to give

$$z = \sum_{j=1}^M w_j a_j + w_0 \quad (5.12)$$

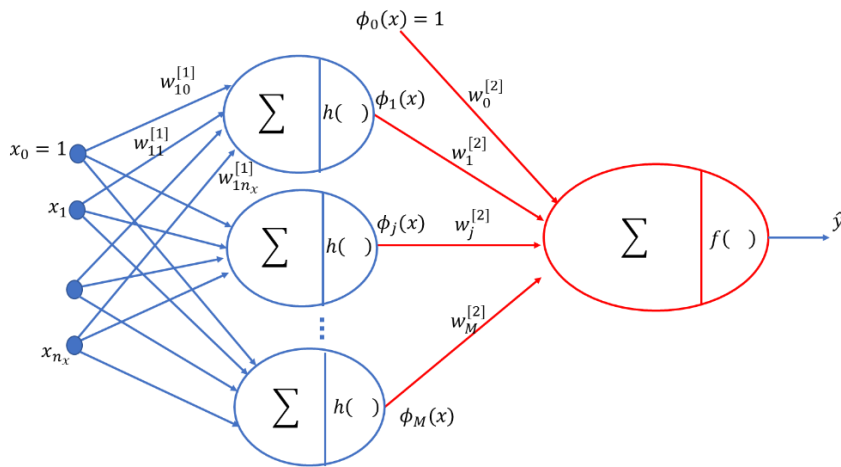
$$\hat{y}(\mathbf{x}; W) = f(z) \quad (5.13)$$

Finally, we can combine equations (5.10)-(5.13) to give the overall network function

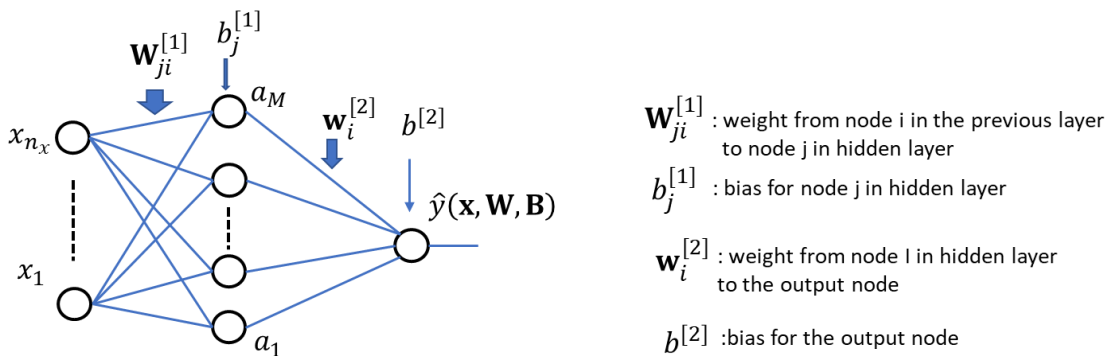
$$\begin{aligned} \hat{y}(\mathbf{x}; \mathbf{W}, \mathbf{B}) &= f \left( \sum_{j=1}^M w_j h \left( \sum_{i=1}^{n_x} w_{ji} x_i + w_{j0} \right) + w_0 \right) \\ &= f \left( \sum_{j=1}^M w_j^{[2]} h \left( \sum_{i=1}^{n_x} w_{ji}^{[1]} x_i + b_j^{[1]} \right) + b^{[2]} \right) \end{aligned} \quad (5.14)$$

where the set of all weight and bias parameters have been grouped into  $\mathbf{W}$  and  $\mathbf{B}$ , respectively. In our text, we use superscripts [ ] to denote the network layer for the parameters. Although  $w_{j0}$  and  $w_0$  can be grouped into the weight set by introducing constant nodes  $x_0 = 1$  and  $a_0 = 1$ , sometimes it may be convenient to denote them as biases  $b_j^{[1]}$  and  $b^{[2]}$  separately. In (5.14), the outputs of  $h(\cdot)$  serve as a role of parameter-dependent (not fixed) basis functions in (5.8). In this example, the first layer includes all logistic regression units which generate  $\phi_j$  while the second layer has only one logistic regression unit that generates  $\hat{y}$ , as shown in Fig.5.4(a). Each logistic regression unit is called a *neuron* or *node*.

A concise graphic representation for the neural network is shown in Fig.5.4 (b), without showing the details within a node. The neural network has a three-layer structure: input layer, hidden layer, and output layer, even though the neural network is two-layer (the input layer does not count because there is no operation in the input layer). The process of computing (5.14) can be interpreted as a forward propagation of information through the network. The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes.



(a) A neural network with one hidden layer and one output layer.



(b) a simplified graphic representation of the two-layer neural network.

Fig.5.4 Diagram for the two-layer neural network corresponding to (5.14)

We can extend the neural network in Fig.5.4 by adding additional hidden layers and/or more nodes in each hidden layer. Theoretically, there is no limit on the number of layers or on the number of nodes in each hidden layer. In general, a larger neural network is expected to recognize more complicated patterns but may suffer overfitting if the training data is not sufficient. The choice of activation functions such as  $h(\cdot)$  and  $f(\cdot)$  depends on the nature of data and the task of the network. We will discuss activation functions when specific neural networks are presented. Another generalization of the network architecture is to have  $K$  nodes, instead of a single node, in the output layer so that it can solve  $K$ -class classification problems with each node delivering the probability of a class. Furthermore, the network can be sparse, with not all possible links between two consecutive layers being present. Convolutional neural networks are examples of sparse network and are widely used in computer vision.

### 5.3 Neural Network Representation: Feed-forward Propagation

This section describes the mathematical representation of a general feedforward neural network, with a focus on notations of variables and parameters. It is essential to understand these notations for efficiently implementing neural network algorithms in computer program languages.

We will use the example shown in Fig.5.5 to demonstrate the fundamental concepts of neural networks though a typical neural network in practice is much larger. Fig.5.5 shows a neural network that has two layers: hidden layer with 4 units and output layer with one unit.

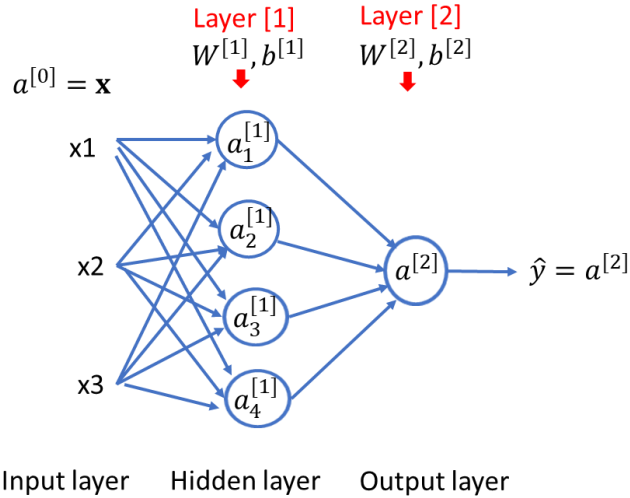


Fig.5.5 A two-layer neural network: an example

The important notations are described as follows. First,  $a_j^{[i]}$  denotes the output of the  $j$ th node in layer  $i$ , and is also the name of this node. Thus,  $a^{[i]}$ , a concatenation of  $a_j^{[i]}$ , is the vector that represents the outputs of layer  $i$ .  $a^{[i]}$  is also the input vector to layer  $i+1$ . The input layer is defined as layer 0. Therefore, for the neural network in Fig.5.5, we have

$$a^{[0]} = \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} \quad a^{[2]} = \hat{y}$$

The parameters associated with layer  $i$  are denoted by

$$W^{[i]} = \begin{bmatrix} - & - & - & W_1^{[i]T} & - & - & - \\ - & - & - & W_2^{[i]T} & - & - & - \\ & & & \vdots & & & \\ - & - & - & W_n^{[i]T} & - & - & - \end{bmatrix}_{(n^{[i]}, n^{[i-1]})} \quad b^{[i]} = \begin{bmatrix} b_1^{[i]} \\ b_2^{[i]} \\ \vdots \\ b_n^{[i]} \end{bmatrix}$$

where  $W_j^{[i]}$  represents a weight column-vector for node  $j$  in layer  $i$  and its element  $W_{jk}^{[i]}$  is the weight from node  $k$  in the previous layer (i.e. layer  $i-1$ ) to node  $j$  in layer  $i$ . The weight vectors of all nodes in layer  $i$  are organized as a weight matrix  $W^{[i]}$ . All weights to node  $j$  in layer  $i$  form row  $j$  in the

matrix  $W^{[i]}$  and thus the weight matrix has a shape of  $(n^{[i]}, n^{[i-1]})$ , i.e., matrix  $W^{[i]}$  has  $n^{[i]}$  rows and  $n^{[i-1]}$  columns, and  $n^{[i]}$  is the number of nodes in layer  $i$ . The bias vector for layer  $i$  is denoted by  $\mathbf{b}^{[i]}$ , whose element  $b_j^{[i]}$  is the bias to the node  $j$  in layer  $i$ . In general, the superscripts in square brackets indicate the layer number.

Therefore, given input vector  $\mathbf{x}$ , the forward propagation can be described by a series of equations:

1) hidden layer

$$\mathbf{z}^{[1]} = W^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \quad (5.15.a)$$

$$\mathbf{a}^{[1]} = \sigma(\mathbf{z}^{[1]}) \quad (5.15.b)$$

2) output layer

$$\mathbf{z}^{[2]} = W^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \quad (5.15.c)$$

$$\mathbf{a}^{[2]} = \sigma(\mathbf{z}^{[2]}) \quad (5.15.d)$$

If  $m$  examples are given, we can represent  $m$  examples in a format of  $(n_x, m)$  matrix, where  $n_x$  is the number of features of  $\mathbf{x}$ . ( $n_x = 3$  in the case of Fig.5.5)

$$X = \begin{bmatrix} | & | & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \mathbf{x}^{(m)} \\ | & | & | \end{bmatrix}_{(n_x, m)}$$

where one example forms a corresponding column, the superscript in parentheses indicates the index of an example. Thus, the forward propagation for  $m$  examples can be described by a set of equations:

1) hidden layer

$$Z^{[1]} = W^{[1]}X + \mathbf{b}^{[1]} \quad (5.16.a)$$

$$A^{[1]} = \sigma(Z^{[1]}) \quad (5.16.b)$$

2) output layer

$$Z^{[2]} = W^{[2]}A^{[1]} + \mathbf{b}^{[2]} \quad (5.16.c)$$

$$A^{[2]} = \sigma(Z^{[2]}) \quad (5.16.d)$$

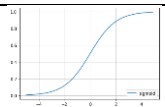
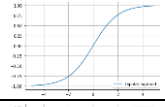
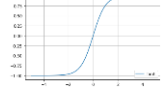
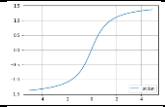
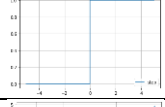
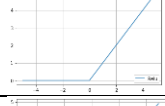
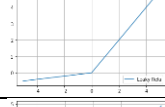
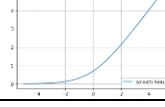
Please note that the operator “+” in (5.16.a) and (5.16.c) imposes broadcasting function since the shape of the two operands, e.g.  $W^{[1]}X$  and  $\mathbf{b}^{[1]}$ , are not the same and different data examples share the same parameters. For example, the shape of  $W^{[1]}X$  is  $(4, m)$ , and the shape of  $\mathbf{b}^{[1]}$  is  $(4, 1)$ . The broadcasting “+” means that  $\mathbf{b}^{[1]}$  is added to each column of  $W^{[1]}X$ . It is helpful for a reader to verify the shape of each variable in (5.16.a)-(5.16.d).

## 5.4 Activation Functions

An activation function is used to introduce non-linearity in a network. This non-linearity allows the model to learn complex mappings based on the available data, and thus the network becomes a universal approximator. The activation function defined in (5.2) is the sigmoid function. In fact, there are other commonly used activation functions. The choice of activation function for each layer is critical. Another important aspect of the activation function is that it should be differentiable. This is required when we compute gradients, and thus tune our weights accordingly.

## Summary of activation functions

For a quick reference, we summarize the activation functions in the following table.

Name	Plot	Equation	derivative
Sigmoid range: (0,1)		$\sigma(z) = \frac{1}{1 + \exp(-z)}$	$\sigma'(z) = \sigma(z)(1 - \sigma(z))$
Bipolar sigmoid Range:(-1,1)		$f(z) = \frac{1 - \exp(-z)}{1 + \exp(-z)}$	$f'(z) = \frac{2e^{-z}}{(1 + e^{-z})^2}$
Tanh Range: (-1,1)		$f(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} = 2\sigma(2z) - 1$	$f'(z) = 1 - f(z)^2$
Arctan Range:( $-\pi/2, \pi/2$ )		$f(z) = \tan^{-1}(z)$	$f'(z) = \frac{1}{1 + z^2}$
Binary step Range: {0,1}		$f(z) = \begin{cases} 0 & z < 0 \\ 1 & z > 0 \end{cases}$	0 if $z \neq 0$
ReLU Range: $[0, +\infty)$		$f(z) = \max(0, x)$	$f'(z) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$
Leaky ReLU Range: $(-\infty, +\infty)$		$f(z) = \max(\alpha z, z)$	$f'(z) = \begin{cases} \alpha & x < 0 \\ 1 & x > 0 \end{cases}$
Smooth ReLU (softplus) Range: $(0, +\infty)$		$f(z) = \ln(1 + \exp(z))$	$f'(z) = \frac{1}{1 + e^{-z}}$
Softmax		$f(z^{[i]}) = \frac{\exp(z^{[i]})}{\sum_{k=1}^K \exp(z_k^{[i]})}$	See Section 3.6.3

To visually compare the activation functions, we plot them in Fig.5.6.

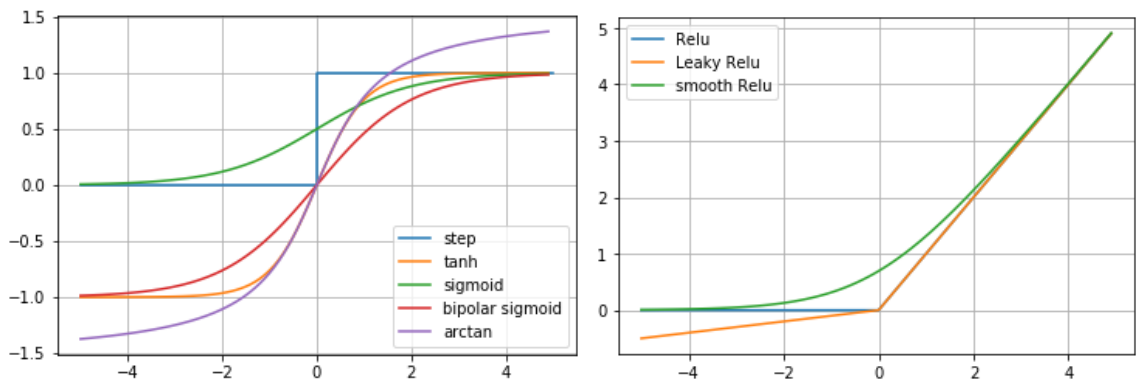


Fig.5.6 Comparison of activation functions



Among the above-mentioned activation functions, three widely used activation functions are sigmoid function, ReLU, and softmax.

### Sigmoid function

The sigmoid or logistic activation function maps the input values into the range (0,1), and its output is interpreted as the probability of the input  $x$  belonging to a class. So, it is mostly used for classification. However, it suffers from the vanishing gradient problem. Also, the output is not zero-centered, which causes difficulties during optimization. It also has a low convergence rate.

### ReLU (Rectified linear unit)

ReLU has the output 0 if its input is less than or equal to 0, otherwise, its output is equal to its input. This has been widely used in convolutional neural networks. It is also superior to the sigmoid and tanh activation function, as it does not suffer from the vanishing gradient problem. Thus, it allows for faster and effective training of deep neural architectures.

### Softmax

The softmax function is widely used for the output layer activation of a neural network in multi-class classification tasks. It is an extension of sigmoid function. Sigmoid function maps a real number to a probability value, which is in the range of (0, 1). In a  $K$ -class classification neural network, the softmax function maps a real vector  $\mathbf{z} \in \mathbb{R}^K$  to a probability distribution vector  $\mathbf{a} \in (0, 1)^K$ , as shown in Fig.5.7. It is obvious that a larger  $z_j$  results in a larger probability element  $a_j$ , and the sum of all elements in  $\mathbf{a}$  is equal to 1. Thus, the output  $\mathbf{a}$  is interpreted as the predicted probability distribution of the input  $x$  over  $K$  classes.

$$a_j = \frac{\exp(z_j)}{\sum_{k=1}^K \exp(z_k)} \quad (5.17)$$

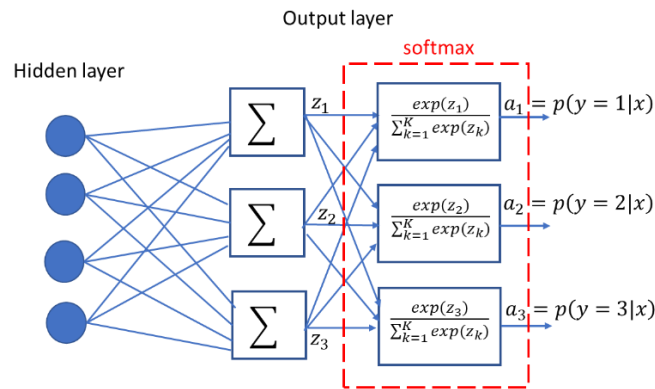


Fig.5.7 Softmax activation (in a case of  $K=3$ )

## 5.5 Network Training: Backward Propagation

The cost function achieves a minimal value at optimal values of parameters, e.g.  $W$  and  $b$ . Searching for the optimal parameter values is called neural network training. In this section, we will discuss how to use gradient descent to train a neural network. Let's consider a 2-layer neural network shown in Fig.5.5. The neural network has one hidden layer with 4 units (or nodes), and

one output layer with one node. The activation function in the hidden layer is now chosen to be tanh function for an easier mathematical representation. The output layer uses the sigmoid function as its activation function.

In the neural network in Fig.5.5, the parameters include:

- $n_x$  is the number of features in input  $x$ .
- $n^{[i]}$  is the number of units in layer  $i$ . Thus  $n^{[0]} = n_x = 3$ ,  $n^{[1]}=4$ ,  $n^{[2]} = 1$
- $W^{[i]}$  is the weight matrix in layer  $i$ . The shape of  $W^{[i]}$  is  $(n^{[i]}, n^{[i-1]})$ . Thus  $W^{[1]}$  shape is  $(4,3)$ .  $W^{[2]}$  shape is  $(1,4)$ .
- $\mathbf{b}^{[i]}$  is the bias of layer  $i$ . The shape of  $\mathbf{b}^{[i]}$  is  $(n^{[i]}, 1)$ . Thus  $\mathbf{b}^{[1]}$  shape is  $(4,1)$ ,  $\mathbf{b}^{[2]}$  shape is  $(1,1)$ .

The cost function is given by

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^m L(a^{[2]^{(i)}} , y^{(i)}) \quad (5.18)$$

where the loss function for one example is cross-entropy loss

$$L(a^{[2]}, y) = -\left(y \cdot \ln a^{[2]} + (1 - y) \ln(1 - a^{[2]})\right) \quad (5.19)$$

Note that the superscript ( $i$ ) indicates the  $i$ th example in the dataset, and the superscript  $[i]$  indicates the layer  $i$ . The training dataset is organized as a matrix with each column corresponding to one example.

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(m)} \\ | & | & | \end{bmatrix}_{(n_x, m)} \quad (5.20)$$

$$Y = [y^{(1)} \quad y^{(2)} \quad y^{(m)}]_{(1, m)} \quad (5.21)$$

The gradient descent algorithm can be partitioned into three steps: 1) forward propagation, 2) backward propagation, and 3) parameter updating. The forward propagation is to calculate the output of each layer following the direction from input to output and the cost (or loss) function. In the backward propagation, the derivatives of the cost function (i.e. gradient), with respect to the parameters  $W$  and  $b$ , are computed in a backward direction. Finally, the parameters are updated based on the derivatives calculated in the backward propagation. The forward propagation is implemented by just following the equations (5.16) and (5.29). The backward propagation requires a series of derivative computations in a backward order.

First, let's consider the derivatives of one example loss function. For the simplicity of notations, we use  $du$  to denote the derivative of the loss  $L$  with respect to a parameter or variable  $u$ , i.e.,  $du \triangleq \frac{d}{du}L$ . The derivatives of the loss function with respect to parameters can be calculated in the following sequence by applying calculus chain rule.

$$da^{[2]} \triangleq \frac{d}{da^{[2]}}L(a^{[2]}, y) = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \quad (5.22)$$

$$dz^{[2]} \triangleq \frac{d}{dz^{[2]}} L(a^{[2]}, y) = da^{[2]} \cdot \frac{d}{dz^{[2]}} (a^{[2]})$$

$$= \left( -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \right) \cdot \sigma(z^{[2]}) (1 - \sigma(z^{[2]})) = a^{[2]} - y \quad (5.23)$$

$$dW^{[2]} \triangleq \frac{d}{dW^{[2]}} L(a^{[2]}, y) = dz^{[2]} \cdot \frac{d}{dW^{[2]}} (z^{[2]}) = dz^{[2]} \cdot (a^{[1]})^T \quad (5.23)$$

Note:  $dW^{[2]}$  has the same shape as  $W^{[2]}$ : (1,4) in the case of Fig.5.5.

$$db^{[2]} \triangleq \frac{d}{db^{[2]}} L(a^{[2]}, y) = dz^{[2]} \cdot \frac{d}{db^{[2]}} (z^{[2]}) = dz^{[2]} \quad (5.24)$$

$$da^{[1]} \triangleq \frac{d}{da^{[1]}} L(a^{[2]}, y) = dz^{[2]} \cdot \frac{d}{da^{[1]}} (z^{[2]}) = (W^{[2]})^T \cdot dz^{[2]} \quad (5.25)$$

$$dz^{[1]} \triangleq \frac{d}{dz^{[1]}} L(a^{[2]}, y) = da^{[1]} \cdot \frac{d}{dz^{[1]}} (a^{[1]}) = (W^{[2]})^T \cdot dz^{[2]} * g^{[1]}'(z^{[1]}) \quad (5.26)$$

Note: \* indicates element-wise multiplication.

$$dW^{[1]} \triangleq \frac{d}{dW^{[1]}} L(a^{[2]}, y) = dz^{[1]} \cdot \frac{d}{dW^{[1]}} (z^{[1]}) = dz^{[1]} \cdot (x)^T \quad (5.27)$$

$$db^{[1]} \triangleq \frac{d}{db^{[1]}} L(a^{[2]}, y) = dz^{[1]} \cdot \frac{d}{db^{[1]}} (z^{[1]}) = dz^{[1]} \quad (5.28)$$

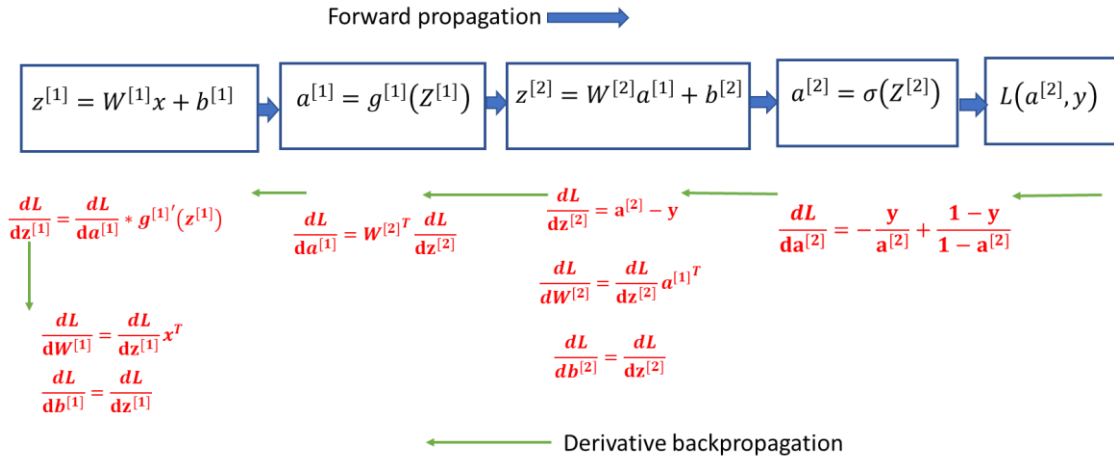


Fig.5.8 Data forward propagation and derivative backward propagation

Fig.5.8 shows the data forward propagation and derivative backward propagation for one data example  $x$ . The data forward propagation diagram illustrates how to calculate the data output at each layer (or each node). After the forward propagation has been completed, the derivatives of loss function  $L$  with respect to different intermediate data or parameters can be computed in an opposite (backward) direction by using the results of forward propagation.

Now we consider the cost function for  $m$  data examples defined by (5.18). By extending the input  $x$  from a one-dimensional vector to two-dimensional matrix with a shape of  $(n_x, m)$ , we can develop the vectorized gradient descent algorithm as follows. It is essential to understand the shape of each variable.

### Gradient descent algorithm for the neural network in Fig.5.5.

Initialize the parameters and shapes:  $W^{[1]}$ : (4,3),  $b^{[1]}$ : (4,1),  $W^{[2]}$ : (1,4),  $b^{[2]}$ : (1,1)

Repeat the loop {

1) Forward propagation:

Equations

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

Note: "+" broadcasting

matrix shape verification

$$(4, m) = (4, 3) \times (3, m) + (4, 1)$$

$$(4, m)$$

$$(1, m) = (1, 4) \times (4, m) + (1, 1)$$

$$(1, m)$$

2) Back propagation:

$$dZ^{[2]} = A^{[2]} - Y \quad (1, m) = (1, m) - (1, m)$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \quad (1, 4) = (1, m) \times (4, m)^T$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \quad (1, 1)$$

Note: sum along axis=1, and keep the two dimensions

$$dZ^{[1]} = (W^{[2]T} \cdot dZ^{[2]}) * g^{[1]'}(Z^{[1]}) \text{ (Note: * element-wise product)}$$

$$(4, m) = ((1, 4)^T \times (1, m)) * (4, m)$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T \quad (4, 3) = (4, m) \times (3, m)^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True) \quad (4, 1)$$

3) Parameter update:

$$W^{[1]} := W^{[1]} - \alpha \cdot dW^{[1]} \quad (4, 3)$$

$$b^{[1]} := b^{[1]} - \alpha \cdot db^{[1]} \quad (4, 1)$$

$$W^{[2]} := W^{[2]} - \alpha \cdot dW^{[2]} \quad (1, 4)$$

$$b^{[2]} := b^{[2]} - \alpha \cdot db^{[2]} \quad (1, 1)$$

}

To make the gradient descent algorithm work, we usually initialize all  $W$  parameters with small random numbers, and  $b$  parameters with (but not necessarily) zeros. If we initialized  $W$  parameters with all zeros, all the units in one layer would be identical, and thus be redundant.

## 5.6 Multi-class Classification: Softmax and Cross Entropy Loss

In this section, we will discuss how to develop a neural network for a multi-class classification task. Softmax regression (or multinomial logistic regression) is a generalization of logistic regression for the case where we want to handle multiple classes. In logistic regression we assumed that the labels were binary:  $y^{(i)} \in \{0, 1\}$ . Softmax regression allows us to handle  $y^{(i)} \in \{1, \dots, K\}$  where  $K$  is the number of classes. The recognition of hand-written digits is a good example for  $K=10$ . In order for

a neural network to perform a multiple-class classification task, we need to implement the output layer with softmax activation. The softmax activation delivers a vector that represent the probabilities of all classes given an input  $x$ .

### 5.6.1 Softmax activation in neural network

Consider a multiple-layer neural network for a 4-class classification task, illustrated in Fig.5.9. The neural network has five layers. Each of the first four layers has a regular activation function (e.g., tanh, or sigmoid). The activation function of the last layer (i.e., the output layer) is the softmax function.

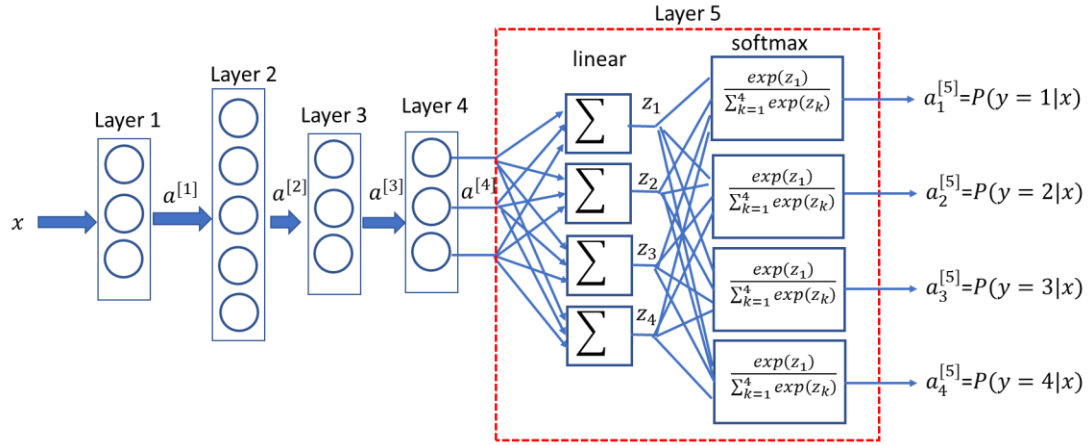


Fig.5.9 A multiple-layer neural network for a 4-class classification task

Specifically, the output layer consists of a linear function and the softmax function, which are described by

$$\text{Linear function: } z = W^{[L]}a^{[L-1]} + b^{[L]} \quad (5.29)$$

Softmax:

$$\mathbf{t} = \exp(z) \quad (5.30.a)$$

$$a^{[L]} = \text{softmax}(z) = \frac{\mathbf{t}}{\sum_{k=1}^K t_k} \quad (5.30.b)$$

where  $L$  is the total number of layers in the neural network,  $L=5$  in Fig.5.9.

$W^{[L]}$  is the weight in the output layer, its shape is  $(4,3)$  in Fig.5.9.

$b^{[L]}$  is the bias parameter, its shape is  $(4,1)$  in Fig.5.9.

$a^{[L-1]}$  is the output of the previous layer, its shape is  $(3,1)$  in Fig.5.9.

$a^{[L]}$  is the output of the output layer, its shape is  $(4,1)$  in Fig.5.9.

Thus,  $z$  and  $\mathbf{t}$  are vectors.  $t_k$  is the  $k$ -th element of vector  $\mathbf{t}$ . The output vector  $a^{[L]}$  can be interpreted as the probabilities of the predicted label over classes for a given input  $x$ . We need to calculate the derivative of softmax function and pass it back to the previous layer during backpropagation. Let

the input be a vector  $\mathbf{z}$  and the output be a vector  $\mathbf{a}$ . The derivative is denoted as  $\frac{da_i}{dz_k}$ . It can be shown (see exercise) that

$$\frac{da_i}{dz_k} = \begin{cases} a_i(1 - a_i) & i = k \\ -a_i a_k & i \neq k \end{cases} \quad (5.31)$$

### 5.6.2 Cross entropy loss and backpropagation

Now let's consider the loss function for one example, say  $(x, y)$ . To calculate the loss, we usually represent the label  $y$  in a one-hot code vector, i.e., for class  $j$ , only the  $j$ th element in  $y$  is 1 and other elements are all zero. For example, the following label represents class 3 out of classes (1,2,3,4).

$$y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Thus, the ground truth  $y$  is the desired ideal output of the neural network.

Cross entropy indicates the distance between the predicted probability distribution  $\hat{y}$  and the ground truth (i.e. label) vector  $y$ . It is defined as

$$L(\hat{y}, y) = -\sum_{j=1}^K y_j \ln(\hat{y}_j) \quad (5.32)$$

Cross entropy measure is a widely used as a loss function in neural networks which have softmax activations in the output layer. Now we can derive the derivative of the loss function with respect to the input vector  $\mathbf{z}$ . Since  $\hat{y} = \text{softmax}(z)$ , we have

$$\frac{dL(\hat{y}, y)}{dz_i} = -\sum_{j=1}^K y_j \frac{d}{dz_i} (\ln \hat{y}_j) = -\sum_{j=1}^K y_j \frac{1}{\hat{y}_j} \frac{d}{dz_i} (\hat{y}_j) \quad (5.33)$$

Using the derivative of softmax function in (5.31) with  $a_i = \hat{y}_i$ , (5.33) can be reduced to (exercise)

$$\frac{dL(\hat{y}, y)}{dz_i} = \hat{y}_i - y_i \quad (5.34)$$

Thus, the derivative of the loss function *w.r.t* the input vector  $\mathbf{z}$  is

$$\frac{dL(\hat{y}, y)}{dz} = \hat{y} - y \quad (5.35)$$

As an example, Fig.5.10 shows the forward propagation and backpropagation for a two-layer neural network (i.e.,  $L=2$ ).

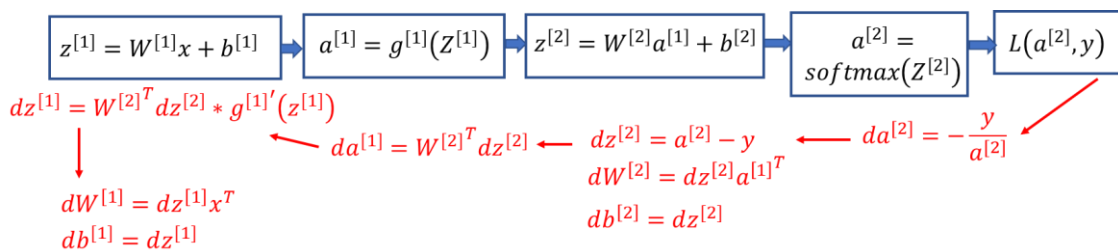


Fig.5.10 Backpropagation for a two-layer neural network ( $L=2$ ) with softmax and cross entropy loss function.

Although the softmax activation and cross entropy loss function are introduced for multi-class classification, the derivative backpropagation (Fig.5.10) has the same equations as logistic regression (see Fig.5.8 and equations (5.22)-(5.28)), except that the dimensions of the variables are different. Therefore, the way to build a neural network for multi-class is the same as binary classification. The only difference is that we label the target  $y$  in a one-hot code format. The shape of parameters in the output layer should match the multiple outputs. For example, the original labels for 4 samples in a 4-class task is

$$Y_{orig} = [1,4,2,3]$$

Then the label used in the neural network should be in the form of one-hot code

$$Y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

And the weight matrix for the output layer is of shape (4, 3), where 3 is assumed to be the number of units in the previous layer.

For  $m$  samples, the cost function can be calculated by

$$J(\hat{Y}, Y) = \frac{-1}{m} \sum (Y * \ln(A^{[L]})) \quad (5.36)$$

Note: in (5.36),  $*$  represents element-wise multiplication of matrices and  $\sum$  represents the sum of all elements in a matrix.  $Y$  is the one-hot code label matrix with shape  $(n^{[L]}, m)$  given by the training dataset.  $\hat{Y}$  is the output of the neural network  $A^{[L]}$  for  $m$  examples, with the same shape of  $Y$ .

In summary, based on the neural networks for binary classifications, we modify them by replacing sigmoid activation with softmax activation function (5.30.b) in the output layer, for forward propagation. The backward propagation does not need to change. The resulting neural network will work for K-class classification tasks. The output of the output layer is a vector predicting the probabilities of K classes given input  $x$ . The predicted class is the class corresponding to the largest element in the output vector.

## 5.7 Practice in Python

In this section, we will guide you to build and train neural networks with Python from scratch.

### 5.7.1 A simple two-layer neural network for binary classification

First, we are going to build a neural network with one hidden layer for a 2-class classification task. The input of the neural network has two features. The network architecture is shown in Fig.5.11. The hidden layer has 4 units with  $\tanh$  activation function. The output layer uses sigmoid activation function to predict the probability of the class ( $y=1$ ) to which the input  $x$  belongs.

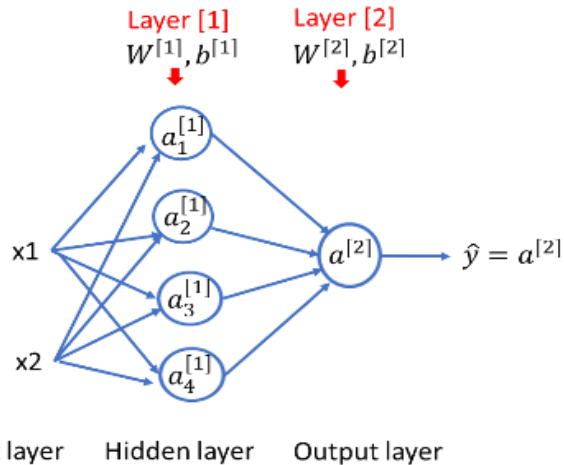


Fig.5.11 NN architecture

### Step 1: import packages and generate training examples

We start the project in Jupyter Notebook with importing the packages.

```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import sklearn.datasets
import sklearn.linear_model
```

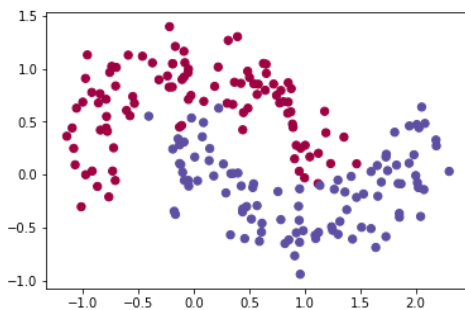
Now let's generate and visualize training examples. The training set to be generated has 200 examples. Each example has two features and a binary label. Specifically, we generate the dataset using the function in the sklearn package. The features and the labels are stored in tensor X (2, 200), and tensor Y (1, 200), respectively.

```
np.random.seed(1)

m=200 # the number of samples
X, Y = sklearn.datasets.make_moons(n_samples=m, noise=.2)
X, Y = X.T, Y.reshape(1, Y.shape[0])

# X: X(2,m), Y: Y(1,m)
# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y[0, :], s=40, cmap=plt.cm.Spectral);

plt.show()
```





## Step 2: build the NN model

First, we need to define the functions (or components) which will be used to build the neural network.

### Define sigmoid function.

```
# sigmoid function
def sigmoid(x):

    s = 1/(1+np.exp(-x))
    return s
```

**Determine the input layer size and the output size.** The size of input layer,  $n_x$ , is determined by the shape of dataset. The size of output layer is determined by the shape of label Y.

```
def layer_sizes(X, Y):
    """
    Argument:
    X -- input dataset of shape (input size, number of examples)
    Y -- labels of shape (output size, number of examples)
    Return:
    n_x -- the number of nodes in the input layer
    n_y -- the number of nodes in the output layer
    """
    ### extract the layer sizes from input and output shapes ###
    n_x = X.shape[0] # size of input layer
    n_y = Y.shape[0] # size of output layer
    #####
    return (n_x, n_y)
```

**Parameter initialization.** The shapes of parameters W1, b1, W2 and b2 are determined by the layer sizes ( $n_x, n_h, n_y$ ). The elements in W1 and W2 are initialized to small random numbers while those in b1 and b3 are initialized to zeros. The results are stored in a python dictionary *parameter*.

```
# FUNCTION: initialize_parameters

def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer
    Returns:
    params -- python dictionary containing parameters:
                W1 -- weight matrix of shape (n_h, n_x)
                b1 -- bias vector of shape (n_h, 1)
                W2 -- weight matrix of shape (n_y, n_h)
                b2 -- bias vector of shape (n_y, 1)
    """
    np.random.seed(2) # set up a seed for reproductivity.

    W1 = np.random.randn(n_h, n_x) * 0.01 # random numbers for weights
    b1 = np.zeros((n_h, 1)) # zeros for bias
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))
```

```

assert (W1.shape == (n_h, n_x))
assert (b1.shape == (n_h, 1))
assert (W2.shape == (n_y, n_h))
assert (b2.shape == (n_y, 1))

parameters = {"W1": W1,
              "b1": b1,
              "W2": W2,
              "b2": b2}

return parameters

```

**Forward propagation.** In the gradient descent algorithm, we iteratively update the parameters to reduce the value of cost function. During each iteration, we first calculate the forward propagation, then the backward propagation (derivatives), and finally update the parameters.

```

# FUNCTION: forward_propagation

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing your parameters

    Return:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Implement Forward Propagation to calculate A2 (probabilities)
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
            "A1": A1,
            "Z2": Z2,
            "A2": A2} # Results at all layers

    return A2, cache

```

**Cost function.** Computing cost is not required for training the model, but it is a useful tool to monitor the training process.

```

# FUNCTION: compute_cost

def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy loss

    Argument:
    A2 -- The output of the second layer, of shape (1, number of examples)

```

```

Y -- "true" labels vector of shape (1, number of examples)
parameters -- python dictionary containing parameters W1, b1, W2 and b2

Return:
cost -- cross-entropy loss
"""

m = Y.shape[1] # number of examples

# Compute the cross-entropy cost
logprobs = np.multiply(np.log(A2),Y) + np.multiply(np.log(1-A2), (1-Y))
cost = -1/m*np.sum(logprobs)

cost = np.squeeze(cost) # makes sure cost is the dimension we expect.
                        # E.g., turns [[17]] into 17
assert(isinstance(cost, float))

return cost

```

**Backward propagation.** This function takes the forward propagation results as inputs, and computes the gradient in a backward order.

```

# FUNCTION: backward_propagation
def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation.

    Argument:
    parameters -- python dictionary containing parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (n_x, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

    Return:
    grads -- python dictionary containing gradients with respect to different p
arameters
    """
    m = X.shape[1] # number of examples

    # First, retrieve W1 and W2 from the dictionary "parameters".
    W1 = parameters["W1"]
    W2 = parameters["W2"]

    # Second, retrieve A1 and A2 from dictionary "cache".
    A1 = cache["A1"]
    A2 = cache["A2"]

    # Backward propagation: calculate dW1, db1, dW2, db2.
    dZ2= A2-Y
    dW2 = 1./m*np.dot(dZ2, A1.T)
    db2 = 1./m*np.sum(dZ2, axis = 1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
    dW1 = 1./m* np.dot(dZ1, X.T)
    db1 = 1./m*np.sum(dZ1, axis = 1, keepdims=True)

    # Save the results
    grads = {"dW1": dW1,
            "db1": db1,
            "dW2": dW2,
            "db2": db2}

    return grads

```

**Parameter update.** This function updates the parameters according to gradient descent algorithm, based on the current parameters and the gradient from backward propagation function.

```
# FUNCTION: update_parameters

def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Update parameters using the gradient descent

    Argument:
    parameters -- python dictionary containing parameters
    grads -- python dictionary containing gradients

    Return:
    parameters -- python dictionary containing updated parameters
    """
    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]

    # Retrieve each gradient from the dictionary "grads"
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]

    # Update for each parameter
    W1 = W1 - dW1 * learning_rate
    b1 = b1 - db1 * learning_rate
    W2 = W2 - dW2 * learning_rate
    b2 = b2 - db2 * learning_rate

    # Save updated parameter results
    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

**Build the NN model** by putting all previous functions together: specify layer sizes, initialize parameters, and define the iteration loop. In each iteration, forward propagation, cost, and backward propagation (or gradient) are calculated, and the parameters are updated.

```
# FUNCTION: nn_model

def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    """
    define a model: input, output, hidden layer size, iteration numbers, print_
    cost or not.

    Argument:
    X -- input shape (2, number of examples)
    Y -- label shape (1, number of examples)
    n_h -- size of the hidden layer
    num_iterations -- number of iterations in update loop
    print_cost -- if True, print the cost every 1000 iterations

    Return:
```

```

parameters -- learned parameters.
"""

np.random.seed(3)
n_x = layer_sizes(X, Y)[0]
n_y = layer_sizes(X, Y)[1]

# Initialize parameters, then retrieve W1, b1, W2, b2.
# Inputs: "n_x, n_h, n_y".
# Outputs: "parameters".
parameters = initialize_parameters(n_x, n_h, n_y)
W1 = parameters["W1"]
W2 = parameters["W2"]
b1 = parameters["b1"]
b2 = parameters["b2"]

# Loop (gradient descent)
for i in range(0, num_iterations):

    # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
    A2, cache = forward_propagation(X, parameters)

    # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
    cost = compute_cost(A2, Y, parameters)

    # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
    grads = backward_propagation(parameters, cache, X, Y)

    # Gradient descent parameter update.
    # Inputs: "parameters, grads". Outputs: "parameters".
    parameters = update_parameters(parameters, grads)

    # Print the cost every 1000 iterations
    if print_cost and i % 1000 == 0:
        print ("Cost after iteration %i: %f" %(i, cost))

return parameters

```

**Predict.** This function predicts the labels for a given test dataset X, based on the trained model.

```

# FUNCTION: predict

def predict(parameters, X):
    """
    predicts a class for each example in X

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (n_x, m)
    Returns
    predictions -- vector of predictions of our model (red: 0 / blue: 1)
    """
    # Computes probabilities using forward propagation,
    # and classifies to 0/1 using 0.5 as the threshold.

    A2, cache = forward_propagation(X, parameters)
    predictions = A2 > 0.5

    return predictions

```

**plot\_decision\_boundary.** This function computes and plots the predictions of the model on points in the two-dimensional grid.

```
# this function plots the predictions for all points.

def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole grid
    Z = model(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.ylabel('x2')
    plt.xlabel('x1')
    plt.scatter(X[0, :], X[1, :], c=y[0, :], cmap=plt.cm.Spectral)
```

### **Step 3: train and test the NN model**

The top-level model is `nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False)`. Now we train the neural network model on the loaded dataset (X,Y), and then plot the decision regions, and evaluate the classification accuracy.

```
from sklearn.metrics import accuracy_score
# Build a model with a n_h-dimensional hidden layer
n_h=4
parameters = nn_model(X, Y, n_h, num_iterations = 10000, print_cost=True)

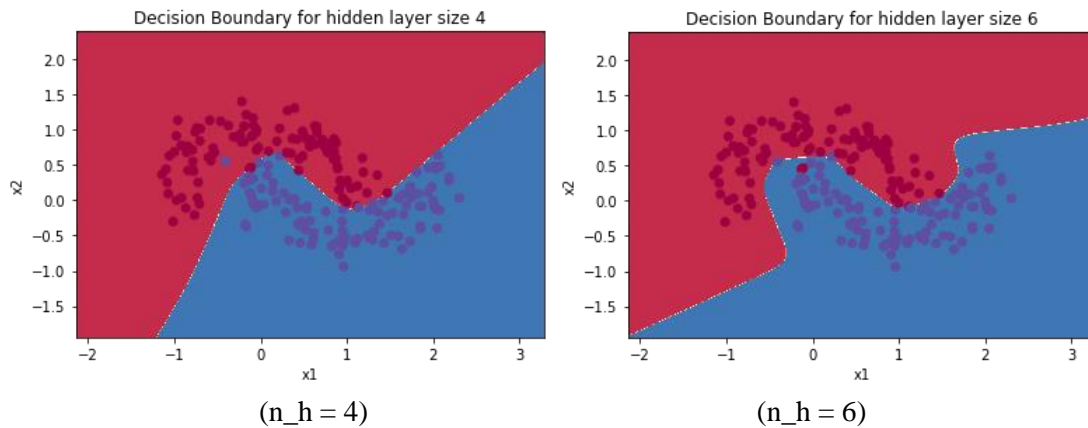
# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(n_h))

predictions = predict(parameters, X)
accuracy_nn = accuracy_score((Y.T).flatten(), predictions.T)
print('Accuracy of nn: ', accuracy_nn)

print('parameters are ', parameters)
```

```
Cost after iteration 0: 0.692995
Cost after iteration 1000: 0.090063
Cost after iteration 2000: 0.064688
Cost after iteration 3000: 0.056812
Cost after iteration 4000: 0.050300
Cost after iteration 5000: 0.044030
Cost after iteration 6000: 0.043814
Cost after iteration 7000: 0.042719
Cost after iteration 8000: 0.041756
Cost after iteration 9000: 0.040949
Accuracy of nn: 0.985
parameters are {'W1': array([[ -4.60909564,  4.50592485],
 [ 2.77721501,  2.62387821],
 [-6.12507597,  0.32178735],
 [12.79405415, -6.32560328]]), 'b1': array([[ 6.12498998],
 [-2.22646889],
 [-4.01149508],
```

```
[ 3.28497244]]], 'W2': array([[ -10.37014269, -13.33751484, -4.32423522,
6.68597777]]), 'b2': array([[ -0.48635814]])}
```



In the plots, the red color indicates the predicted label  $y=0$ , and the blue color indicates the predicted label  $y=1$ . We can see that the feature space is divided into multiple regions with either red or blue color.

## 5.7.2 Multi-class classification on MNIST dataset

### Revisit of two-class classification

Before working on multi-class tasks, it is helpful to summarize the model we developed in the previous section. The model is defined by

```
nn_model(X, Y, n_h, num_ iterations = 1000, print_cost=False)
```

where  $X$  is the matrix of  $m$  data examples,  $Y$  is the label vector,  $n_h$  is the number of units in the hidden layer (only one hidden layer in the model),  $num\_iterations$  is the number of iterations for updating parameters, and  $print\_cost$  is a command for whether print the cost during the training. It is important to identify the shape of  $X$  and  $Y$ ,

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(m)} \\ | & | & | \end{bmatrix}_{(n_x, m)}$$

$$Y = [y^{(1)} \quad y^{(2)} \quad y^{(m)}]_{(1, m)} \quad y^{(i)} \in \{0, 1\}$$

The model consists of two input features ( $n_x = 2$ ), one hidden layer with 4 units ( $n_h = 4$ ), and the output layer with single unit. The hidden layer uses  $\text{Tanh}()$  as activation functions and the output layer uses sigmoid function for binary classification.

In this section, we will show how to modify the previous work for implementing multi-class classification. Specifically, we consider the recognition of handwritten digits in MNIST dataset. Since there are totally 10 handwritten digits (i.e., 0,1,2,3,4,5,6,7,8,9), the recognition is a 10-class classification task.

## MNIST dataset

The MNIST database (Modified National Institute of Standards and Technology database) of handwritten digits consists of a training set of 60,000 examples, and a test set of 10,000 examples.

The images from the data set have the size of 28 x 28 pixels. They are saved in the csv data files `mnist_train.csv` and `mnist_test.csv` (note: there are other ways to load MNIST dataset, such as `torchvision.datasets`). Every row of these files consists of 785 numbers between 0 and 255. The first number in a row is the label, i.e. the digit which the image represents. The next 784 numbers are the pixels of the image. A part of `mnist_train.csv` opened by Excel is shown below (one row is too long to display). The first five images represent digits 5, 0, 4, 1, 9, respectively.

	A	B	C	D	E	F	G	H
1	5	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0
5	9	0	0	0	0	0	0	0

## Neural network architecture

For this MNIST classification task, we adopt the neural network shown in Fig.5.12. The input layer is an image organized as a vector. The hidden layer has 25 nodes with `tanh()` activation. The output layer has 10 nodes with `softmax` activation.

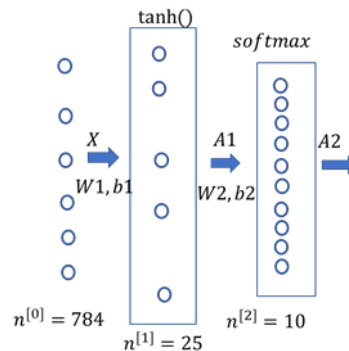


Fig.5.12 Neural network for MNIST classification

## Start Jupyter Notebook

Let's start the project in Jupyter Notebook.

## Import packages and explore the dataset

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

image_size = 28 # width and length
no_of_different_labels = 10 # i.e. 0, 1, 2, 3, ..., 9
image_pixels = image_size * image_size
train_data = np.loadtxt("C:/machine_learning/NN_nn_overview/mnist_train.csv",
                        delimiter=",")
test_data = np.loadtxt("C:/machine_learning/NN_nn_overview/mnist_test.csv",
                       delimiter=",")
```



The following codes read the data and display some examples. The images of the MNIST dataset are greyscale and the pixels range between 0 and 255 including both bounding values. We map these values into an interval from [0.01, 1] by multiplying each pixel by 0.99 / 255 and adding 0.01 to the result. This way, we avoid 0 values as inputs, which may prevent weight updates.

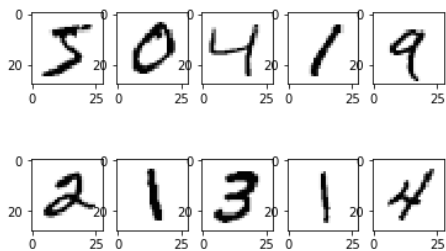
```
fac = 0.99/255 # convert [0,255] to [0,1]
train_imgs = np.asfarray(train_data[:, 1:])*fac+0.01
test_imgs = np.asfarray(test_data[:, 1:])*fac+0.01
# first column is labels

train_labels = np.asfarray(train_data[:, :1])
test_labels = np.asfarray(test_data[:, :1])
```

```
print ('The shape of train_imgs is: ' + str(train_imgs.shape))
print ('The shape of train_labels is: ' + str(train_labels.shape))
print ('The shape of test_imgs is: ' + str(test_imgs.shape))
print ('The shape of test_labels is: ' + str(test_labels.shape))
```

```
The shape of train_imgs is: (60000, 784)
The shape of train_labels is: (60000, 1)
The shape of test_imgs is: (10000, 784)
The shape of test_labels is: (10000, 1)
```

```
# display the first 10 examples
for i in range(10):
    img = train_imgs[i].reshape((28,28))
    plt.subplot(2,5,1+i)
    plt.imshow(img, cmap="Greys")
plt.show()
```



**Prepare the data for the neural network.** The following codes convert the original labels to one-hot-code format, and also transpose the input feature matrix to match the format defined earlier (one column represents one example). The resulting X is shape of (784,60000), X\_test is shape of (784,10000) and Y is shape of (10, 60000).

```
np.random.seed(1) # set a seed so that the results are consistent

lr = np.arange(no_of_different_labels)

# transform labels into one hot representation
train_labels_one_hot = (lr==train_labels).astype(np.float)
test_labels_one_hot = (lr==test_labels).astype(np.float)

X=np.transpose(train_imgs)
Y=np.transpose(train_labels_one_hot)
X_test=np.transpose(test_imgs)
```

```

### find the shape of X and Y
### find the number of examples
shape_X = X.shape
shape_Y = Y.shape
m = X.shape[1] # training set size

print ('The shape of X is ' + str(shape_X))
print ('The shape of Y is ' + str(shape_Y))
print ('the number of examples is m = %d' % (m))

```

```

The shape of X is (784, 60000)
The shape of Y is (10, 60000)
the number of examples is m = 60000

```

## Build neural network

To build the neural network for MNIST classification, we just need to make the following changes to our previous binary classification project.

- 1) Network size: we still use a two-layer network but a larger hidden layer with 25 units (you can select a different number, say 20 or 31), as shown in Fig.5.12. The input layer and output layer sizes are automatically determined by the shape of X and Y. Thus n\_h is assigned to 25.
- 2) Output layer: we use softmax activation function to replace sigmoid function in the output layer.

In `forward_propagation(X, parameters)`, `A2 = sigmoid(Z2)` should be changed to

```

A2 = softmax(Z2)
assert(A2.shape == (W2.shape[0], X.shape[1])) ## double check the shape

```

No change is needed for `backward_propagation`.

- 3) Cost function computation. `compute_cost(A2, Y, parameters)` is only for cost display, and does not contribute training process.

```

# for 2-class task
logprobs = np.multiply(np.log(A2),Y) + np.multiply(np.log(1-A2), (1-Y))

```

```

# for multiple-class task
logprobs = np.multiply(np.log(A2),Y)

```

- 4) Prediction. If we use `X_test` as the input, the output `A2` is a matrix of shape (10,10000). Each column of `A2` represents the probability distribution over 10 classes for one example. Thus we can use

```

predictions = np.argmax(A2, axis=0)

```

to find the predicted digits. The following statement is used to calculate the accuracy.

```
accuracy = accuracy_score(test_labels.flatten(), predictions)
```

In the end, it shows the classification accuracy is 98% based on the test dataset.

**The Python code is attached below.** Compared to the previous binary classification, the changes are highlighted.

Sigmoid() is replaced by softmax().

```
# Neural network from scratch
def softmax(x):
    t=np.exp(x)
    s = t/np.sum(t, axis=0)

    return s
```

No change to layer\_sizes() function.

```
def layer_sizes(X, Y):
    """
    Argument:
    X -- input dataset of shape (input size, number of examples)
    Y -- labels of shape (output size, number of examples)
    Return:
    n_x -- the number of nodes in the input layer
    n_y -- the number of nodes in the output layer
    """
    ### extract the layer sizes from input and output shapes ###
    n_x = X.shape[0] # size of input layer
    n_y = Y.shape[0] # size of output layer
    #####
    return (n_x, n_y)
```

No change to initialize\_parameters() function.

```
# FUNCTION: initialize_parameters

def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer
    Returns:
    params -- python dictionary containing parameters:
                W1 -- weight matrix of shape (n_h, n_x)
                b1 -- bias vector of shape (n_h, 1)
                W2 -- weight matrix of shape (n_y, n_h)
                b2 -- bias vector of shape (n_y, 1)
    """
    np.random.seed(2) # set up a seed for reproductivity.

    W1 = np.random.randn(n_h, n_x) * 0.01 # random numbers for weights
    b1 = np.zeros((n_h, 1)) # zeros for bias
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))
```

```

assert (W1.shape == (n_h, n_x))
assert (b1.shape == (n_h, 1))
assert (W2.shape == (n_y, n_h))
assert (b2.shape == (n_y, 1))

parameters = {"W1": W1,
              "b1": b1,
              "W2": W2,
              "b2": b2}

return parameters

```

Changes to forward\_propagation () are highlighted.

```

# FUNCTION: forward_propagation

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing current parameters

    Returns:
    A2 -- The softmax output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Implement Forward Propagation to calculate A2 (probabilities)
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2

    # multiple classes
    A2 = softmax(Z2)
    assert(A2.shape == (W2.shape[0], X.shape[1])) #for multiple classes

    cache = {"Z1": Z1,
            "A1": A1,
            "Z2": Z2,
            "A2": A2}

    return A2, cache

```

Changes to compute\_cost () are highlighted.

```

# FUNCTION: compute_cost

def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost

    Arguments:
    A2 --The softmax output of the second activation, of shape (n_y, number of
examples)
    Y -- "true" labels vector of shape (n_y, number of examples)
    parameters --python dictionary containing your parameters W1, b1, W2 and b2

```

```

Returns:
cost -- cross-entropy cost
"""

m = Y.shape[1] # number of example

# for multiple-class task
logprobs = np.multiply(np.log(A2),Y)
cost = -1/m*np.sum(logprobs)
cost = np.squeeze(cost) # makes sure cost is the dimension we expect.
                        # E.g., turns [[17]] into 17
assert(isinstance(cost, float))

return cost

```

No changes for backward\_propagation ().

```

# FUNCTION: backward_propagation

def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation

    Arguments:
    parameters -- python dictionary containing current parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (n_x, number of examples)
    Y -- "true" labels vector of shape (n_y, number of examples)

    Returns:
    grads -- python dictionary containing the gradients with respect to different parameters
    """
    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary "parameters".
    W1 = parameters["W1"]
    W2 = parameters["W2"]

    # Retrieve also A1 and A2 from dictionary "cache".
    A1 = cache["A1"]
    A2 = cache["A2"]

    # Backward propagation: calculate dW1, db1, dW2, db2.
    dZ2 = A2 - Y
    dW2 = 1./m*np.dot(dZ2, A1.T)
    db2 = 1./m*np.sum(dZ2, axis = 1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
    dW1 = 1./m* np.dot(dZ1, X.T)
    db1 = 1./m*np.sum(dZ1, axis = 1, keepdims=True)

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads

```

No changes to update\_parameters().

```
# FUNCTION: update_parameters

def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Updates parameters using the gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients

    Returns:
    parameters -- python dictionary containing your updated parameters
    """
    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]

    # Retrieve each gradient from the dictionary "grads"
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]

    # Update rule for each parameter
    W1 = W1 - dW1 * learning_rate
    b1 = b1 - db1 * learning_rate
    W2 = W2 - dW2 * learning_rate
    b2 = b2 - db2 * learning_rate

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

No change to nn\_model().

```
# FUNCTION: nn_model

def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    """
    Arguments:
    X -- dataset of shape (n_x, number of examples)
    Y -- labels of shape (n_y, number of examples)
    n_h -- size of the hidden layer
    num_iterations -- Number of iterations in gradient descent loop
    print_cost -- if True, print the cost every 1000 iterations

    Returns:
    parameters -- parameters learnt by the model. They can then be used to predict.
    """
    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[1]

    # Initialize parameters,
```

```

# then retrieve W1, b1, W2, b2.
# Inputs: "n_x, n_h, n_y". Outputs = "W1, b1, W2, b2 in parameters".
parameters = initialize_parameters(n_x, n_h, n_y)
W1 = parameters["W1"]
W2 = parameters["W2"]
b1 = parameters["b1"]
b2 = parameters["b2"]

# Loop (gradient descent)

for i in range(0, num_iterations):

    # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
    A2, cache = forward_propagation(X, parameters)

    # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
    cost = compute_cost(A2, Y, parameters)

    # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
    grads = backward_propagation(parameters, cache, X, Y)

    # Gradient descent parameter update.
    # Inputs: "parameters, grads". Outputs: "parameters".
    parameters = update_parameters(parameters, grads)

    # Print the cost every 1000 iterations
    if print_cost and i % 10 == 0:
        print ("Cost after iteration %i: %f" %(i, cost))

return parameters

```

changes to predict() are highlighted.

```

# FUNCTION: predict

def predict(parameters, X):
    """
    Using the learned parameters, predicts a class for each example in X

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (n_x, m)

    Returns
    predictions -- vector of predictions
    """

    # Computes probabilities using forward propagation,
    # and classifies by argmax.

    A2, cache = forward_propagation(X, parameters)

    # for multiple classes
    # use argmax to find the digit: np.argmax(a, axis=0)
    predictions = np.argmax(A2, axis=0)

    return predictions

```

## Train the model and test its classification accuracy.

```
from sklearn.metrics import accuracy_score
# Build a model with a n_h-dimensional hidden layer
n_h=25
parameters = nn_model(X, Y, n_h, num_iterations = 100, print_cost=True)

predictions = predict(parameters, X)
accuracy_train = accuracy_score(train_labels.flatten(), predictions)
print('Accuracy of training: ', accuracy_train)

predictions = predict(parameters, X_test)
accuracy_test = accuracy_score(test_labels.flatten(), predictions)
print('Accuracy of test: ', accuracy_test)
```

```
Cost after iteration 0: 2.302137
Cost after iteration 10: 1.239301
Cost after iteration 20: 0.902871
Cost after iteration 30: 0.530826
Cost after iteration 40: 0.465360
Cost after iteration 50: 0.398585
Cost after iteration 60: 0.345182
Cost after iteration 70: 0.321645
Cost after iteration 80: 0.313386
Cost after iteration 90: 0.325701
Accuracy of training: 0.92115
Accuracy of test: 0.9254
```

## Summary and further reading

This chapter provides foundations of neural networks, including mathematical representation (notations), forward propagation, backward propagation, and parameter updating based on gradient descent. Section 5.7 describes the details of implementation of simple neural networks in Python from scratch.

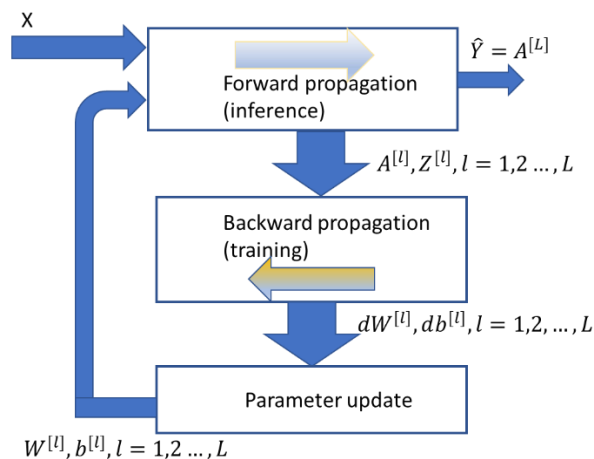


Fig. 5.13 Training and inference processes for neural networks

The basic framework for developing neural networks is illustrated in Fig.5.13. After the architecture of the network (e.g. size and activation functions) has been determined, we can start the **training** process with randomized initial parameters. In each iteration of



parameter updating process, the intermediate outputs of each layer during the forward propagation are stored for calculating the gradients of cost function with respect to parameters in backward propagation. The parameters are updated based on the gradients. The trained neural network can predict for new input  $X$  using forward propagation only. This prediction process is called **inference**.

Files: ([http://localhost:8889/tree/ch3\\_basic\\_nn](http://localhost:8889/tree/ch3_basic_nn))

C:\Users\weido\ch3\_basic\_nn\ch5\_ex1.ipynb, ch5\_mnist.ipynb,

C:/machine\_learning/NN\_nn\_overview/mnist\_train.csv

C:/machine\_learning/NN\_nn\_overview/mnist\_test.csv

### Further reading

[1] Christopher M. Bishop, "Pattern recognition and machine learning", chapter 5 Neural Networks.

### Exercises

1. Sigmoid function and  $\tanh(\cdot)$  are two commonly used activation functions for neural networks. They are defined as

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

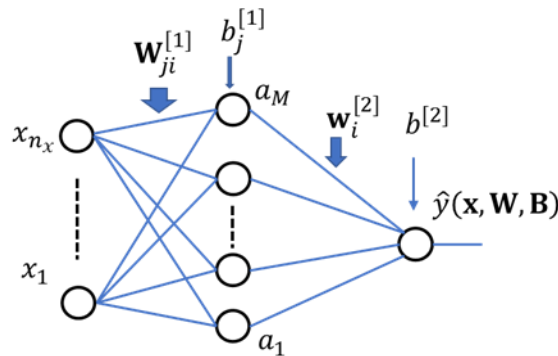
- 1) Prove that

$$\tanh(z) = 2\sigma(2z) - 1$$

- 2) Calculate derivatives of  $\sigma(z)$  and  $\tanh(z)$  in terms of sigmoid function  $\sigma(\cdot)$ .

2. Consider a two-layer neural network defined by (5.14), shown in Fig.5.4.

$$\hat{y}(\mathbf{x}; \mathbf{W}, \mathbf{B}) = \mathbf{f} \left( \sum_{j=1}^M w_j^{[2]} \mathbf{h} \left( \sum_{i=1}^{n_x} w_{ji}^{[1]} x_i + b_j^{[1]} \right) + b^{[2]} \right) \quad (5.14)$$



The hidden layer nonlinear activation function  $\mathbf{h}(\cdot)$  is given by sigmoid function

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Show that there exists an equivalent neural network, which computes exactly the same function, but with the hidden unit activation function given by

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

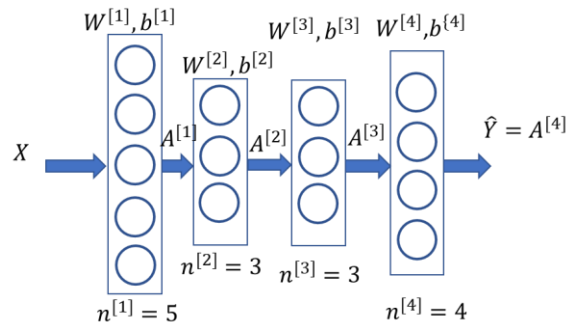
(Hint: first find the relation between  $\sigma(z)$  and  $\tanh(z)$ , and then show the relationship between the parameters between the two neural networks)

3. Prove equations (5.31) and (5.34). To understand the notations, you need to read the corresponding contexts.

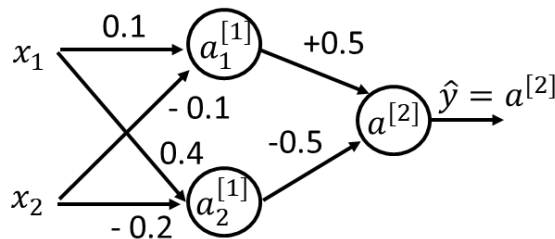
$$\frac{da_i}{dz_k} = \begin{cases} a_i(1 - a_i) & i = k \\ -a_i a_k & i \neq k \end{cases} \quad (5.31)$$

$$\frac{dL(\hat{y}, y)}{dz_i} = \hat{y}_i - y_i \quad (5.34)$$

4. The architecture of a neural network is illustrated in the figure below. The input examples have 100 features, and X includes 1000 examples. Thus, X is a matrix of shape (100, 1000). Please find the shapes of all parameters  $W^{[l]}, b^{[l]}, l = 1, 2, 3, 4$ , and the shapes of  $A^{[1]}, A^{[2]}, A^{[3]}, A^{[4]}$ .

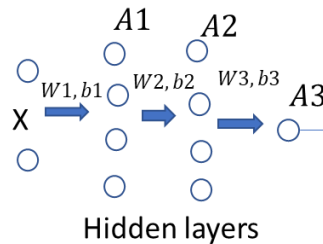


5. Consider a neural network below, with  $\tanh()$  function for the hidden layer activation and the sigmoid function for the output layer activation. The initial weights are shown along with the connections in the figure (all initial biases are zero). Suppose we have two training examples:  $(x_1, x_2) = (2, 3)$  with the label  $y = 1$ ; and  $(x_1, x_2) = (-1, -2)$  with the label  $y = 0$ , and use cross-entropy as the loss function.



By utilizing the Python code provided in this chapter, for the two examples, compute:

- 1) forward propagation (i.e. the outputs for the two examples).
  - 2) the loss.
  - 3) backward propagation (i.e. gradients).
  - 4) one-step updated parameters, given that the learning rate is 0.1.
  - 5) the loss, based on the updated parameters.
6. In section 5.7.1, we trained a neural network with two layers. The hidden layer has 4 units with  $\tanh()$  activations and the output layer has one unit with sigmoid activation. In this exercise, we will build and train a neural network (shown below) with three layers: two hidden layers with 4 units in each layer, and the output layer has one unit with sigmoid activation. The hidden units use  $\tanh()$  activation functions.



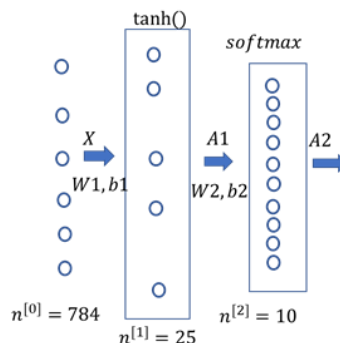
Choose an appropriate learning rate, print all trained parameters, plot the cost function versus iterations, plot training example scatterplot and decision boundary, and print the accuracy.

Do this project using one of the following datasets:

- 1) Noisy\_moons.  
`noisy_moons = sklearn.datasets.make_moons(n_samples=N, noise=.2)`
- 2) Noisy\_circles.  
`noisy_circles = sklearn.datasets.make_circles(n_samples=N, factor=.5, noise=.3)`

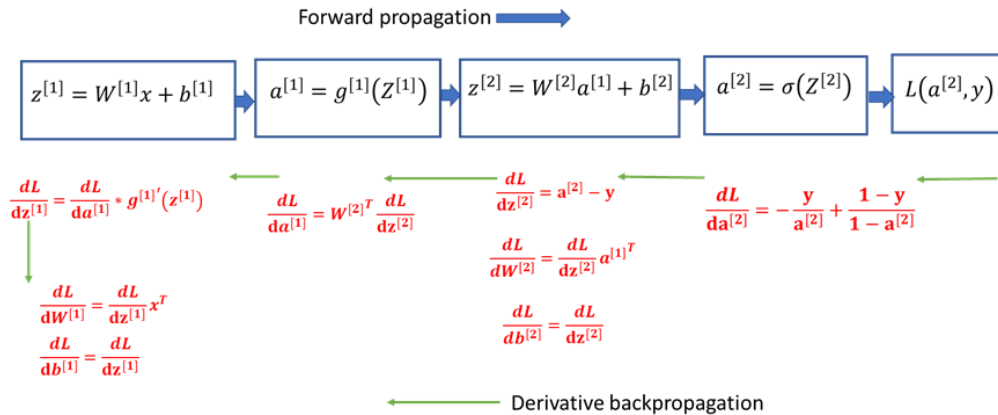
Compare this network with the one developed in section 5.7.1. which one is better? Justify your answer.

7. Develop a neural network illustrated in the figure below to recognize the handwritten digit using MNIST datasets `mnist_train.csv` and `mnist_test.csv`. (reference: Section 5.7.2)



- 1) Plot cost versus iteration index.

- 2) Find the accuracy of your trained neural network on mnist\_test.csv.
  - 3) Select a different size for the hidden layer, and repeat 1) and 2). Compare the results.
8. Consider the neural network in Fig.5.8 (below). The gradient descent algorithm can be described as follows.



Initialize the parameters and shapes:  $W^{[1]}$ : (4,3),  $b^{[1]}$ : (4,1),  $W^{[2]}$ : (1,4),  $b^{[2]}$ : (1,1)

Repeat the loop {

- 1) Forward propagation:

Equations

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

matrix shape verification

$$(4, m) = (4, 3) \times (3, m) + (4, 1)$$

$$(4, m)$$

$$(1, m) = (1, 4) \times (4, m) + (1, 1)$$

$$(1, m)$$

- 2) Back propagation:

$$dZ^{[2]} = A^{[2]} - Y \quad (1, m) = (1, m) - (1, m)$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \quad (1, 4) = (1, m) \times (4, m)^T$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \quad (1, 1)$$

$$dZ^{[1]} = (W^{[2]T} \cdot dZ^{[2]}) * g^{[1]'}(Z^{[1]}) \text{ (Note: * element-wise product)}$$

$$(4, m) = ((1, 4)^T \times (1, m)) * (4, m)$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T \quad (4, 3) = (4, m) \times (3, m)^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True) \quad (4, 1)$$

- 3) Parameter update:

$$W^{[1]} := W^{[1]} - \alpha \cdot dW^{[1]} \quad (4, 3)$$

$$b^{[1]} := b^{[1]} - \alpha \cdot db^{[1]} \quad (4, 1)$$

$$W^{[2]} := W^{[2]} - \alpha \cdot dW^{[2]} \quad (1, 4)$$

$$b^{[2]} := b^{[2]} - \alpha \cdot db^{[2]} \quad (1, 1)$$

}

It is noted that

$dZ^{[1]} = (W^{[2]T} \cdot dZ^{[2]}) * g^{[1]'}(Z^{[1]})$  depends on the activation function  $g^{[1]}(\cdot)$  in the hidden layer.

- 1) Show that if  $g^{[1]}(\cdot) = \tanh(\cdot)$ , the Python statement to calculate  $dZ^{[1]}$ , in `backward_propagation(parameters, cache, X, Y)`, is

$$dZ1 = \text{np.dot}(W2.T, dZ2) * (1 - \text{np.power}(A1, 2))$$

- 2) If sigmoid function  $\sigma(\cdot)$  is used for  $g^{[1]}(\cdot)$ , what is the Python statement to calculate  $dZ^{[1]}$ ?
- 3) Modify codes in Section 5.7 for using  $\sigma(\cdot)$  as the activation function in hidden layer. Compare the results with the case of  $\tanh(\cdot)$  activation function in hidden layer (i.e. Section 5.7).

Hint: modify the corresponding statements (relevant to  $g^{[1]}(\cdot)$ )

$A^{[1]} = g^{[1]}(Z^{[1]})$  in `forward_propagation(X, parameters)`

$dZ^{[1]} = (W^{[2]T} \cdot dZ^{[2]}) * g^{[1]'}(Z^{[1]})$  in `backward_propagation(parameters, cache, X, Y)`