Chapter 6

Basics of Neural Networks

6.1 Learning Objectives

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. In this chapter, we will present the basics of neural networks. The Learning objectives of this chapter include:

- To understand the extension of regression to neural networks
- To represent a neural network in mathematical equations and network diagram
- To be familiar with mathematical notations of variables and parameters in neural networks
- To perform forward propagation and derivative backpropagation
- To know various activation functions
- To train a neural network using gradient descent in a vectorized format
- To apply softmax activation function for multi-class classifications

6.2 From Regression to Neural Networks

In this section, instead of directly giving the mathematical representation of neural networks, we will move toward neural networks from 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 Chapters 3 and 4, respectively, are based on linear combinations of input features and take the form

$$\hat{y}(x;\, w) = f\left(\sum_{j=1}^{n_x} w_j x_j + w_0\right) = f\left(\sum_{j=0}^{n_x} w_j x_j\right) \tag{6.1}$$

where $x_j, j = 1,2,\ldots,n_x$ are input features, $x_0 = 1$, f(.) is a nonlinear activation function (e.g. sigmoid function) in the case of classification and is the identity in the case of linear regression. The semicolon in $\hat{y}(x;\, w)$ separates the variables and parameters. However, the resulting fitting curves in linear regression and the resulting decision boundaries in classification are 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,

$$\hat{y}(\mathbf{x};\, w) = \sum_{j=0}^{M} w_j \phi_j(\mathbf{x}) \tag{6.2}$$

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, $\{\phi_j(\mathbf{x}), j = 1,2,\ldots,M\}$ 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 $\phi_j(\mathbf{x})$ in (6.1),

$$\hat{y}(\mathbf{x};\, w) = f\left(\sum_{j=0}^{M} w_j \phi_j(\mathbf{x})\right) \tag{6.3}$$

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 (6.3) by making the basis functions 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 (6.1) to generate M basis functions, $\phi_j(\mathbf{x})$ for (6.3) (note that $\phi_0(\mathbf{x})$ is always equal to 1 so that w_0 is a bias)

$$z_j^{[1]} = \sum_{i=0}^{n_x} w_{ji}^{[1]} x_i \tag{6.4}$$

$$\phi_j(\mathbf{x}; w_{j:}^{[1]}) = a_j = h(z_j^{[1]}) \tag{6.5}$$

where j=1,2,…,M, and in the notation $w_{ji}^{[1]}$ and $z_j^{[1]}$, the superscript [1] indicates that the parameter is in the first layer of the network, *j* indicates that the parameter or the quantity is associated with the jth basic function, and *i* indicates that the parameter is the weight for input x_i. Thus $z_j^{[1]}$ *, j=1,2,…,M,* are the linear combinations of all input features. The M basis functions are generated by passing the M linear combinations $z_j^{[1]}$ through a differentiable nonlinear activation function h(.), respectively. The nonlinear function h(.) is typically chosen to be sigmoid function. The quantities a_j are known as *activations*, which are the outputs of activation functions. These activations, used as the basis functions in (6.3), are linearly combined to give

$$z^{[2]} = \sum_{j=1}^{M} w_j^{[2]} a_j + w_0^{[2]} \tag{6.6}$$

$$\hat{y}(\mathbf{x};\, W) = f(z^{[2]}) \tag{6.7}$$

Finally, we can combine these various steps (6.4)-(6.7) to give the overall network function

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

$$= \boldsymbol{f}\left(\textstyle\sum_{j=1}^{M} w_j^{[2]} \boldsymbol{h}\left(\sum_{i=1}^{n_x} w_{ji}^{[1]} x_i + b_j^{[1]}\right) + b^{[2]}\right) \tag{6.8}$$

where the set of all weight and bias parameters have been grouped into **W** and **B**, respectively. Although $w_{j0}^{[1]}$ and $w_0^{[2]}$ can be grouped into the weight set by introducing constant nodes $x_0 = 1$ and $a_0 = 1$, in practice it may be convenient to treat them as biases $b_j^{[1]}$ and $b^{[2]}$ separately. In (6.8), the outputs of h(.) serve a role of parameter-dependent (not fixed) basis functions in (6.3). Thus, the network is more powerful at the cost of more parameters to be learned. This network function can be represented in the form of a network diagram as shown in Fig.1. The diagram 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 (6.8) 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.

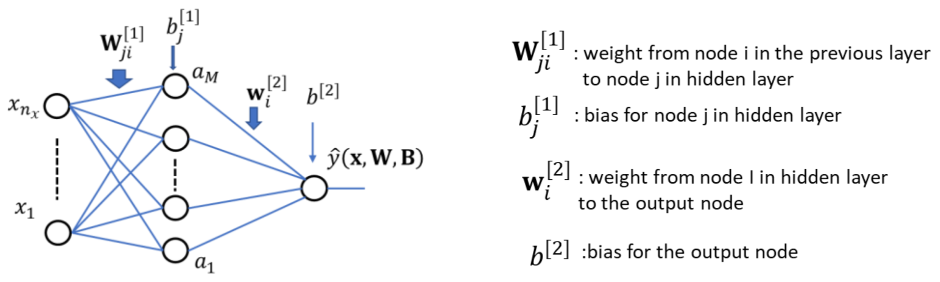

$\mathbf{W}_{ji}^{[1]}$
$b_j^{[1]}$
a_M
$\mathbf{w}_i^{[2]}$
$b^{[2]}$
x_{n_x}
$\hat{y}(\mathbf{x}, \mathbf{W}, \mathbf{B})$
x_1
a_1
$\mathbf{W}_{ji}^{[1]}$: weight from node i in the previous layer to node j in hidden layer
$b_j^{[1]}$: bias for node j in hidden layer
$\mathbf{w}_i^{[2]}$: weight from node I in hidden layer to the output node
$b^{[2]}$:bias for the output node

Fig.1 Diagram for the two-layer neural network corresponding to (6.8)

We can extend the neural network in Fig.1 by adding additional hidden layers and/or more nodes in hidden layers. 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(.) and f(.) 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 generate 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.

6.3 A simplest Neural Network: a Logistic Regression Unit

A neural network generally comprises multiple layers of nodes, known as *neurons*. In our context, the relationship between the neurons in machine learning and the neurons in biological literatures is not our concern. As shown in the previous section, 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 texts. 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 $\mathbf{x} \in \mathbb{R}^{\boldsymbol{n_x}}$, 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 parameter vector $W \in \mathbb{R}^{n_x}$ and $b \in \mathbb{R}$ from the training dataset X and Y, and then estimate the label y for a new value of **x** as

$$\hat{y} = p(y = 1|\mathbf{x}) = \sigma(z) = \sigma(W^T\mathbf{x} + b) \tag{6.9}$$

where $z = \sum_{j=1}^{n_x} w_j x_j + b$ is a linear combination of input features with a bias b,
$\sigma(\,)$ is the sigmoid activation function, shown in Fig.2, defined by

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{6.10}$$

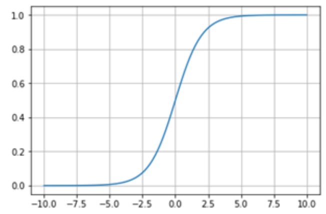


Fig. 2 Sigmoid function

To learn an optimal parameters *W* and *b*, we first define a cost function, and then minimize the cost function over *W* and *b* space. The cost function of logistic regression, for a single data example (**x**,*y*), is defined as

$$L(\hat{y}, y) = -y\ln(\hat{y}) - (1-y)\ln(1-\hat{y}) \tag{6.11}$$

Thus, the average cost function of *m* data samples is given by

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ln\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\ln\left(1-\hat{y}^{(i)}\right)\right] \tag{6.12}$$

where,

$$\hat{y}^{(i)} = \sigma(W^T x^{(i)} + b) \tag{6.13}$$

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

Repeat {

$$W := W - \alpha \frac{dJ(W,b)}{dW} \tag{6.14.a}$$

$$b := b - \alpha \frac{dJ(W,b)}{db} \tag{6.14.b}$$

}

The logistic regression can be viewed as a special neural network with a single neuron, illustrated in Fig.3. Multiple logistic regression units can be used to construct a multi-layer neural network, shown in Fig.4, where each circle represents a logistic regression unit. We will discuss the formal representation of neural networks later.

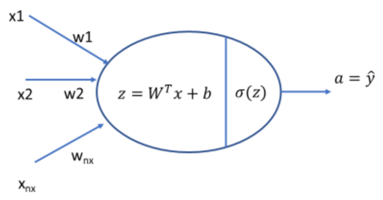


Fig.3 a logistic regression unit

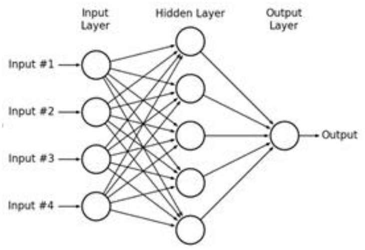


Fig.4 an example of multi-layer neural network

6.4 Derivative Computation: Backward Propagation

To implement the gradient descent algorithm efficiently for multiple-layer neural networks, we introduce computation graph: an analysis tool of derivative computation. For instance, we consider a function *J(a,b,c)=3(a+bc)*. Let *u=bc*, *v=a+u*, *J=3v*, then the computation of *J(a,b,c)* can be illustrated by the graph in Fig.5. The red numbers show a set of example values calculated at all locations.

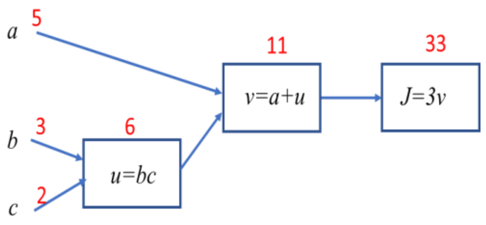


Fig.5 Forward propagation

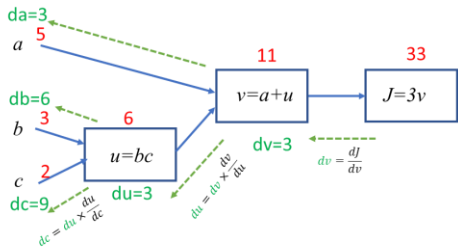


Fig.6 Derivative computation through Backward propagation

To compute a derivative, a small disturbance is introduced to the corresponding variable, and then the change of the function value is calculated. The derivative can be estimated by the ratio of the function value change and the variable disturbance. For example, to compute $\frac{dJ}{dv}$, we set v=11.001, and compute the corresponding J=33.003. Thus $\frac{dJ}{dv} = \frac{0.003}{0.001} = 3$. For simplicity, $\frac{dJ}{dv}$ will be denoted by dv later on, if J is the final output of the function. For example, $da \triangleq \frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da}$. We set a=5.001, and then get v=11.001 and J=33.003. Thus $da \triangleq \frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da} = \frac{0.003}{0.001}\frac{0.001}{0.001} = 3$. The derivatives of J with respect to all variables are calculated and labeled in Fig.6. The green dash arrows show the sequence of derivatives to be computed by following derivative chain rule in a backward direction. For instance, to compute $\frac{dJ}{dc}$, we can compute it as $\frac{dJ}{dv}\frac{dv}{du}\frac{du}{dc} = 9$.

Now consider a logistic regression with $n_x=2$ (two features in **x**). Hence the cost function of the logistic regression for one training example can be represented by $L(a, y)$,

$$z = W^T\mathbf{x} + b \tag{6.15.a}$$

$$\hat{y} = a = \sigma(z) \tag{6.15.b}$$

$$L(a, y) = -y\ln(a) - (1-y)\ln(1-a) \tag{6.15.c}$$

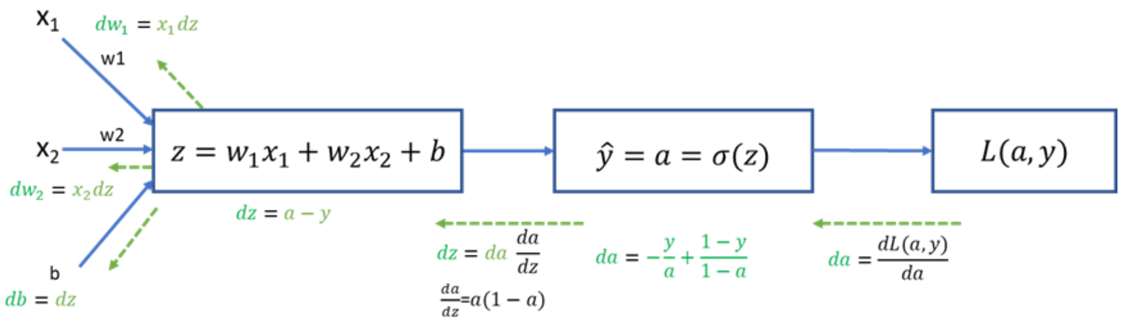


Fig.7 Computation of partial derivatives of cost function *L(a,y)* with respect to parameters. The notation *dx* means $\frac{d}{dx}L$.

The computation graph for the cost function is shown in Fig.7. The derivatives of cost function with respect to w_1, w_2 *and b* can be computed by computing *da*, *dz* first in a backpropagation style. This computation process is based on chain rule of derivative. As the result, the parameters *W* and *b* can be updated as

$$dw_1 = x_1dz = x_1(a-y) \tag{6.16.a}$$

$$dw_2 = x_2dz = x_2(a-y) \tag{6.16.b}$$

$$db = dz = (a-y) \tag{6.16.c}$$

$$w_1 \coloneqq w_1 - \alpha dw_1 \tag{6.17.a}$$

$$w_2 \coloneqq w_2 - \alpha dw_2 \tag{6.17.b}$$

$$b \coloneqq b - \alpha db \tag{6.17.c}$$

The cost function of *m* training examples is

$$J(W, b) = \frac{1}{m}\sum_{i=1}^{m} L\left(a^{(i)}, y^{(i)}\right) \tag{6.18.a}$$

$$a^{(i)} = \sigma\left(z^{(i)}\right) = \sigma\left(W^T x^{(i)} + b\right) \tag{6.18.b}$$

Thus, the derivatives of *J* with respect to *W* and *b* can be computed by

$$\frac{\partial J(W,b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} dw_1^{(i)} \tag{6.19.a}$$

$$\frac{\partial J(W,b)}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m} dw_2^{(i)} \tag{6.19.b}$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)} \tag{6.19.c}$$

where

$$dw_1{}^{(i)} = x_1{}^{(i)} dz^{(i)} = x_1{}^{(i)}(a^{(i)} - y^{(i)}) \tag{6.19.d}$$

$$dw_2{}^{(i)} = x_2{}^{(i)} dz^{(i)} = x_2{}^{(i)}(a^{(i)} - y^{(i)}) \tag{6.19.e}$$

$$dz^{(i)} = a^{(i)} - y^{(i)} \tag{6.19.f}$$

Thus, the parameters can be updated as

$$w_1 \coloneqq w_1 - \alpha\frac{\partial J(W,b)}{\partial w_1} = w_1 - \alpha\frac{1}{m}\sum_{i=1}^{m} x_1{}^{(i)}(a^{(i)} - y^{(i)}) \tag{6.20.a}$$

$$w_2 \coloneqq w_2 - \alpha\frac{\partial J(W,b)}{\partial w_2} = w_2 - \alpha\frac{1}{m}\sum_{i=1}^{m} x_2{}^{(i)}(a^{(i)} - y^{(i)}) \tag{6.20.b}$$

$$b \coloneqq b - \alpha\frac{\partial J(W,b)}{\partial b} = b - \alpha\frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \tag{6.20.c}$$

Now we can calculate the derivatives of J(W,b) with respect to all parameters, for a general situation where the number of features is n_x and the number of examples is *m*. Using vector operations in Python, we can avoid loop operations, and thus achieve very efficient computations. The training dataset consists of *m* data examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$. The *m* training data can be represented by a $n_x \times m$ matrix X. The *m* labels can be denoted by a row vector

$$Y = [y^{(1)}, y^{(2)} \ldots, y^{(m)}] \in \mathbb{R}^{1\times m}$$

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

The algorithm can be vectorized as:

```
Repeat:
{                                                   Shape for each intermediate variable (tensor)
    $Z = W^T X + b = np.dot(W.T, X) + b$                Z:  (1, m), row vector, np.dot() numpy function
    $A = \sigma(Z)$                                     A: (1,m), row vector
    $dZ = A - Y$                                        dZ: (1,m), row vector
    $dW = \frac{1}{m} X (dZ)^T$                          dW: ($n_x$,1), column vector
    $db = \frac{1}{m} np.sum(dZ)$                        db: (1,1), a scalar, np.sum() numpy function
    $W \coloneqq W - \alpha dW$                          W: ($n_x$,1), column vector
    $b \coloneqq b - \alpha db$                          b: (1,1), a scalar
}
```

6.5 Neural Network Representation

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

In this chapter, we will use an example shown in Fig.8 to demonstrate the fundamental concepts of neural networks though a typical neural network in practice is much larger. Fig.8 shows a neural network that has two layers: hidden layer with 4 units and output layer with one unit. The important notations are described as follows. First, $a_j^{[i]}$ denotes the output of the *j*th node in layer *i*, and also serves 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 for layer *i+1*. The input layer is defined as layer 0. Therefore,

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

The parameters associated with layer *i* are denoted by

$$W^{[i]} = \begin{bmatrix} ---W_1^{[i]^T}--- \\ ---W_2^{[i]^T}--- \\ \vdots \\ ---W_{n^{[i]}}^{[i]^T}--- \end{bmatrix}_{(n^{[i]},n^{[i-1]})} \qquad b^{[i]} = \begin{bmatrix} b_1^{[i]} \\ b_2^{[i]} \\ \vdots \\ b_{n^{[i]}}^{[i]} \end{bmatrix}$$

where $W_j^{[i]}$ represents a weight vector for node *j* in layer *i* and its element $W_j^{[i]}(k)$ 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 $\left(n^{[i]}, n^{[i-1]}\right)$, and $n^{[i]}$ is the number of nodes in layer *i*. $b_j^{[i]}$is the bias to the node *j* in layer *i*. In general, the superscripts in square brackets indicate the layer number.

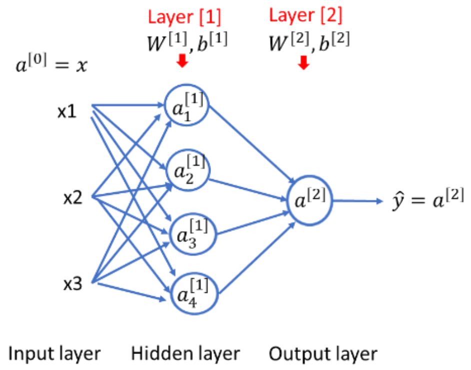


Fig.8 A 2-layer neural network

Therefore, given input vector **x**, the forward propagation can be described by a set of equations:

1) hidden layer

$$z^{[1]} = W^{[1]}\mathbf{x} + b^{[1]} \tag{6.21.a}$$

$$a^{[1]} = \sigma\left(z^{[1]}\right) \tag{6.21.b}$$

2) output layer

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \tag{6.21.c}$$

$$a^{[2]} = \sigma\left(z^{[2]}\right) \tag{6.21.d}$$

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 **x**. (for example, $n_x = 3$ for Fig.8)

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

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

1) hidden layer

$$Z^{[1]} = W^{[1]}X + b^{[1]} \tag{6.22.a}$$

$$A^{[1]} = \sigma\left(Z^{[1]}\right) \tag{6.22.b}$$

2) output layer

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \tag{6.22.c}$$

$$A^{[2]} = \sigma\left(Z^{[2]}\right) \tag{6.22.d}$$

Please note that the operator "+" in (6.22.a) and (6.22.c) imposes broadcasting function since the shape of the two operands, e.g. $W^{[1]}X$ and $b^{[1]}$, 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 $b^{[1]}$ is *(4,1)*. The broadcasting "+" means that $b^{[1]}$ is added to each column of $W^{[1]}X$. It is helpful for a reader to verify the shape of each variable in (6.22.a)-(6.22.c).

6.6 Activation Functions

The activation function defined in (6.10) is the sigmoid function. In fact, there are other commonly used activation functions. The choice of activation function for each layer is very important. An activation function is used to introduce non-linearity to a network. This allows us to model a class label / score that varies non-linearly with independent variables. Non-linear means the output cannot be replicated from a linear combination of inputs, this allows the model to learn complex mappings from the available data, and thus the network becomes a universal approximator, whereas, a model which uses a linear function (i.e. no activation function) is unable to make sense of complicated data, such as, speech, videos, etc. and is effective for only a single layer.

Another important aspect of the activation function is that it should be differentiable. This is required when we backpropagate through our network and compute gradients, and thus tune our weights accordingly. The non-linear functions are continuous and map the input into the range (0,1), (−1,1), etc. In a neural network, it is possible for some neurons to have linear activation functions, but they must be accompanied by neurons with non-linear activation functions in some other part of the same network. Although any non-linear function can be used as an activation function, in practice, only a small fraction of these are used.

Sigmoid

$$a_j^{[i]} = \sigma(z) = \frac{1}{1+exp(-z)} \tag{6.23}$$

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

Bipolar Sigmoid

$$a_j^{[i]} = \sigma(z) = \frac{1-exp\ (-z)}{1+exp(-z)} \tag{6.24}$$

The sigmoid function can be scaled to have any range of output values, depending upon the problem. When the range is from −1 to 1, it is called a bipolar sigmoid.

Tanh

$$a_j^{[i]} = tanh(z) = \frac{exp(z)-exp(-z)}{exp(z)+exp(-z)} = 2\sigma(2z) - 1 \tag{6.25}$$

The tanh non-linearity compresses the input in the range (−1,1). It provides an output which is zero-centered. So, large negative values are mapped to negative outputs, similarly, zero-valued inputs are mapped to near zero outputs. Also, the gradients for tanh are steeper than sigmoid, but it suffers from the vanishing gradient problem.

Arctan

$$a_j^{[i]} = tan^{-1}(z) \tag{6.26}$$

This activation function maps the input values in the range (−π/2, π/2). Its derivative converges quadratically against zero for large input values. On the other hand, in the sigmoid activation function, the derivative converges exponentially against zero, which can cause problems during back-propagation. Its graph is slightly flatter than tanh, so it has a better tendency to differentiate between similar input values.

Binary Step

$$a_j^{[i]} = f(z) = \begin{cases} 0 & z < 0 \\ 1 & z > 0 \end{cases} \tag{6.27}$$

A binary step function is generally used in the Perceptron linear classifier. It thresholds the input values to 1 and 0, if they are greater or less than zero, respectively. This activation function is useful when the input pattern can only belong to one of two groups i.e. binary classification.

ReLU (Rectified linear unit)

$$a_j^{[i]} = f(z) = maximum(0, z) \tag{6.28}$$

A rectified linear unit has the output 0 if its input is less than or equal to 0, otherwise, its output is equal to its input. It is also more biologically accurate. 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.

However, being non-differentiable at 0, ReLU neurons have a tendency to become inactive for all inputs i.e. they die out. This can be caused by high learning rates, and can thus reduce the model's learning capacity. This is commonly referred to as the "Dying ReLU" problem.

Leaky ReLU

$$a_j^{[i]} = f(z) = maximum(\alpha z, z) \tag{6.29}$$

The non-differentiability at zero problem can be solved by allowing a small value to flow when the input is less than or equal to 0, which thus overcomes the "Dying ReLU" problem. It has proved to give better results for some problems. α is typically equal to 0.01.

Smooth ReLU

$$a_j^{[i]} = f(z) = \log(1 + \exp(z)) \tag{6.30}$$

This activation function also overcomes the "Dying ReLU" problem by making itself differentiable everywhere and causes less saturation overall.

Softmax

$$a_j^{[i]} = \frac{exp\left(z_j^{[i]}\right)}{\sum_k z_k^{[i]}} \tag{6.31}$$

The softmax function's output tells us the probabilities that any of the classes are true, so it produces values in the range (0,1). It highlights the largest values and tries to suppress values which are below the maximum value, its resulting values always sum to 1. This function is widely used in multiple classification logistic regression models.

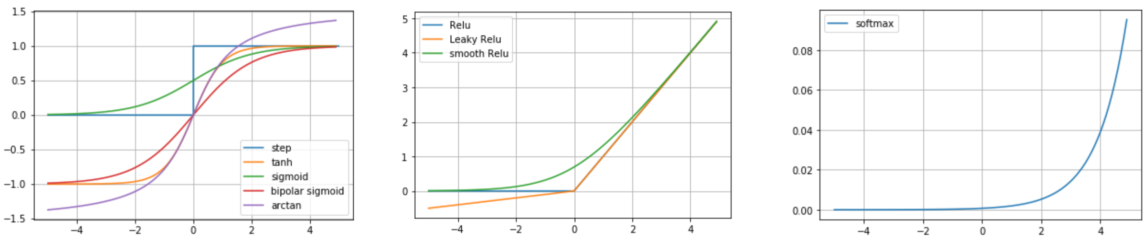
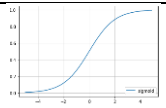
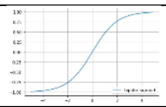
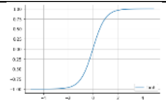
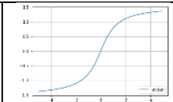
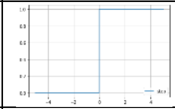
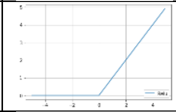
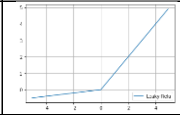
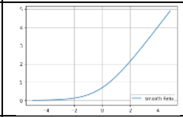
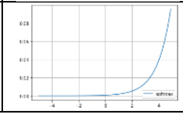


Fig.9 Comparison of activation functions

Summary of activation functions

The comparison of activation functions is visualized in Fig.9. Their derivatives are listed the following table. The derivative of an activation function will be used in backward propagation.

Name	**Plot**	**Equation**	**derivative**
Sigmoid		$\sigma(z) = \dfrac{1}{1 + exp(-z)}$	$\sigma'(z) = \sigma(z)(1 - \sigma(z))$
Bipolar sigmoid		$f(z) = \dfrac{1 - exp(-z)}{1 + exp(-z)}$	$f'(z) = \dfrac{2e^{-z}}{(1 + e^{-z})^2}$
tanh		$f(z) = tanh(z) = \dfrac{exp(z) - exp(-z)}{exp(z) + exp(-z)} = 2\sigma(2z) - 1$	$f'(z) = 1 - f(z)^2$

arctan		$f(z) = tan^{-1}(z)$	$f'(z) = \frac{1}{1+z^2}$
Binary step		$f(z) = \begin{cases} 0 & z < 0 \\ 1 & z > 0 \end{cases}$	0 if z≠ 0
ReLU		$f(z) = \max(0, x)$	$f'(z) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$
Leaky ReLU		$f(z) = max(\alpha z, z)$	$f'(z) = \begin{cases} \alpha & x < 0 \\ 1 & x > 0 \end{cases}$
Smooth ReLU (softplus)		$f(z) = \log(1 + \exp(z))$	$f'(z) = \frac{1}{1+e^{-z}}$
Softmax		$f(z^{[i]}) = \frac{exp\left(z_j^{[i]}\right)}{\sum_k z_k^{[i]}}$	

6.7 Network Training: Gradient Descent

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.8. In addition to input layer, 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.8, 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 $\left(n^{[i]}, n^{[i-1]}\right)$. Thus $W^{[1]}$ shape: (4,3). $W^{[2]}$ shape: (1,4).
- $b^{[i]}$ is the bias of layer *i*. The shape of $b^{[i]}$ is $\left(n^{[i]}, 1\right)$. Thus $b^{[1]}$ shape: (4,1), $b^{[2]}$ shape: (1,1).

The cost function is given by

$$J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right) = \frac{1}{m}\sum_{i=1}^{m} L\left(a^{[2]^{(i)}}, y^{(i)}\right) \tag{6.32}$$

where the loss function for one example is

$$L\left(a^{[2]}, y\right) = -\left(y \cdot lna^{[2]} + (1-y)ln\left(1 - a^{[2]}\right)\right) \tag{6.33}$$

Note that the superscript *(i)* refers the *ith* example in the dataset. The training dataset is denoted by

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

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

The gradient descent can be partitioned into three parts: 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 are computed with respect to the parameters W and b, in a backward direction. In the end, the parameters are updated based on the derivatives calculated in the backward propagation.

The forward propagation and backward propagation are shown in Fig.10. The forward propagation is implemented by just following the equations (6.21) and (6.33). The backward propagation requires a series of derivative computations. The derivatives of the loss function with respect to parameters can be calculated in the following sequence by applying chain rule.

$$da^{[2]} \triangleq \frac{d}{da^{[2]}} L\left(a^{[2]}, y\right) = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \tag{6.34}$$

$$\begin{aligned} dz^{[2]} \triangleq \frac{d}{dz^{[2]}} L\left(a^{[2]}, y\right) &= da^{[2]} \cdot \frac{d}{dz^{[2]}}\left(a^{[2]}\right) \\ &= \left(-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right) \cdot \sigma\left(z^{[2]}\right)\left(1 - \sigma\left(z^{[2]}\right)\right) = a^{[2]} - y \end{aligned} \tag{6.35}$$

$$dW^{[2]} \triangleq \frac{d}{dW^{[2]}} L\left(a^{[2]}, y\right) = dz^{[2]} \cdot \frac{d}{dW^{[2]}}\left(z^{[2]}\right) = dz^{[2]} \cdot \left(a^{[1]}\right)^T \tag{5.36}$$

$$db^{[2]} \triangleq \frac{d}{db^{[2]}} L\left(a^{[2]}, y\right) = dz^{[2]} \cdot \frac{d}{db^{[2]}}\left(z^{[2]}\right) = dz^{[2]} \tag{6.37}$$

$$da^{[1]} \triangleq \frac{d}{da^{[1]}} L\left(a^{[2]}, y\right) = dz^{[2]} \cdot \frac{d}{da^{[1]}}\left(z^{[2]}\right) = \left(W^{[2]}\right)^T \cdot dz^{[2]} \tag{6.38}$$

$$dz^{[1]} \triangleq \frac{d}{dz^{[1]}} L\left(a^{[2]}, y\right) = da^{[1]} \cdot \frac{d}{dz^{[1]}}\left(a^{[1]}\right) = \left(W^{[2]}\right)^T \cdot dz^{[2]} * g^{[1]\prime}\left(z^{[1]}\right) \tag{6.39}$$

Note: * indicates element-wise multiplication.

$$dW^{[1]} \triangleq \frac{d}{dW^{[1]}} L\left(a^{[2]}, y\right) = dz^{[1]} \cdot \frac{d}{dW^{[1]}}\left(z^{[1]}\right) = dz^{[1]} \cdot (x)^T \tag{6.40}$$

$$db^{[1]} \triangleq \frac{d}{db^{[1]}} L\left(a^{[2]}, y\right) = dz^{[1]} \cdot \frac{d}{db^{[1]}}\left(z^{[1]}\right) = dz^{[1]} \tag{6.41}$$

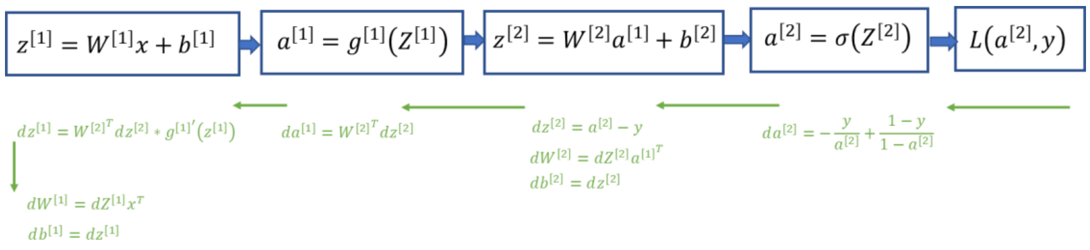

$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = g^{[1]}(Z^{[1]})$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(Z^{[2]})$
$L(a^{[2]}, y)$
$da^{[2]} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$
$dz^{[2]} = a^{[2]} - y$
$dW^{[2]} = dZ^{[2]} a^{[1]T}$
$db^{[2]} = dz^{[2]}$
$da^{[1]} = W^{[2]T} dz^{[2]}$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$
$dW^{[1]} = dZ^{[1]} x^T$
$db^{[1]} = dz^{[1]}$

Fig.10 Data forward propagation and derivative backward propagation

Fig. 10 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 be 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. When considering the cost function for *m* data examples in (6.32), we can develop the following vectorized gradient descent algorithm:

```
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                                   matrix shape verification
      $Z^{[1]} = W^{[1]}X + b^{[1]}$                  $(4,m) = (4,3) \times (3,m) + (4,1)$
      $A^{[1]} = g^{[1]}(Z^{[1]})$                     $(4,m)$
      $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$              $(1,m) = (1,4) \times (4,m) + (1,1)$
      $A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$      $(1,m)$
2)    Back propagation:

      $dZ^{[2]} = A^{[2]} - Y$                         $(1,m) = (1,m) - (1,m)$
      $dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$          $(1,4) = (1,m) \times (4,m)^T$
      $db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$          $(1,1)$
      $dZ^{[1]} = \left(W^{[2]^T} \cdot dZ^{[2]}\right) * g^{[1]'}(Z^{[1]})$ (Note: * element-wise product)
                                                  $(4,m) = ((1,4)^T \times (1,m)) * (4,m)$
      $dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$                $(4,3) = (4,m) \times (3,m)^T$
      $db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$          $(4,1)$
3)    Parameter update:

      $W^{[1]} := W^{[1]} - \alpha \cdot dW^{[1]}$         $(4,3)$
      $b^{[1]} := b^{[1]} - \alpha \cdot db^{[1]}$         $(4,1)$
      $W^{[2]} := W^{[2]} - \alpha \cdot dW^{[2]}$         $(1,4)$
      $b^{[2]} := b^{[2]} - \alpha \cdot db^{[2]}$         $(1,1)$
}
```

To make the gradient descent algorithm work, we should 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.

6.8 Multi-class Classification: Softmax Regression

In this section, we will discuss how to develop a neural network for a multi-class classification problem. Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to 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 $y^{(i)} \in \{1,\ldots,C\}$ where C is the number of classes. The recognition of hand-written digits is a good example for C=10. In order for a neural network to perform a multiple-class classification task, we only need to modify a neural network with 2-class classification by replacing the output layer activation (e.g. sigmoid) function with softmax activation. The softmax activation delivers a vector that represent the probabilities of all classes given an input *x*.

6.8.1 From logistic regression to Softmax

In logistic regression, we had a training set $\{(x^{(1)},y^{(1)}),\ldots,(x^{(m)},y^{(m)})\}$ of *m* labeled examples, where the input features are $x^{(i)} \in \mathbb{R}^n$. In the binary classification setting, the labels were $y^{(i)} \in \{0,1\}$. Our hypothesis took the form:

$$\hat{y}^{(i)} = h_\theta(x^{(i)}) = \frac{1}{1+exp(-\theta^T x^{(i)})} \tag{5.47}$$

and the model parameters θ were trained to minimize the cost function

$$J(\theta) = -\left[\sum_{i=1}^{m} y^{\{i\}} log\hat{y}^{(i)} + (1-y^{\{i\}}) log(1-\hat{y}^{(i)})\right] \tag{5.48}$$

In the softmax regression setting, we are interested in multi-class classification (as opposed to only binary classification), and so the label y can take on K different values, rather than only two. Thus, in our training set $\{(x^{(1)},y^{(1)}),\ldots,(x^{(m)},y^{(m)})\}$, we now have that $y^{(i)} \in \{1,2,\ldots,C\}$. (Note that our convention will be to index the classes starting from 1, rather than from 0.) For example, in the MNIST digit recognition task, we would have C=10 different classes.

Given a test input *x*, we want our hypothesis to estimate the probability that *P(y=k|x)* for each value of k=1,…,C. i.e., we want to estimate the probability of the class label taking on each of the C different possible values. Thus, our hypothesis will output a C-dimensional vector (whose elements sum to 1) giving us our C estimated probabilities, as shown in Fig.11. Concretely, our hypothesis $h_\theta(x)$ takes the form:

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=C|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{C} exp\left({\theta^{(j)}}^T x\right)} \begin{bmatrix} exp\left({\theta^{(1)}}^T x\right) \\ exp\left({\theta^{(2)}}^T x\right) \\ \vdots \\ exp\left({\theta^{(C)}}^T x\right) \end{bmatrix} \tag{6.49}$$

Here $\theta^{(1)}, \theta^{(2)}, \ldots \theta^{(C)}$ are the parameters of our model, where $\theta^{(j)}$ represents the weights to node *j* in the output layer. Notice that the term $\sum_{j=1}^{C} exp\left({\theta^{(j)}}^T x\right)$ normalizes the distribution, so that the outputs of nodes in the output layer sum to one.

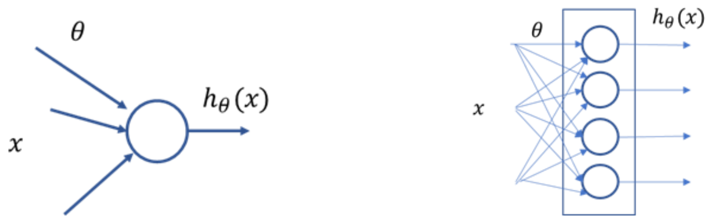


Fig. 11 Softmax regression is a generalization of logistic regression (left: logistic regression, x and θ are vectors, and $h_\theta(x)$ is a scalar, right: softmax regression for 4-class classification, x is a vector, θ is a matrix, and $h_\theta(x)$ is a vector)

For convenience, we will also write θ to denote all the parameters of our model. When we implement softmax regression, it is usually convenient to represent θ as a n-by-C matrix obtained by concatenating $\theta^{(1)}, \theta^{(2)}, \dots \theta^{(C)}$ into columns, so that

$$\theta = [\theta^{(1)} \quad \theta^{(2)} \quad \dots \ \theta^{(C)}] \tag{6.50}$$

6.8.2 Relationship between logistic regression and softmax regression

Softmax regression will reduce to logistic regression when C (the number of classes) is 2. Concretely, when C=2, the softmax regression hypothesis outputs

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \end{bmatrix} = \frac{1}{exp\left({\theta^{(1)}}^T x\right) + exp\left({\theta^{(2)}}^T x\right)} \begin{bmatrix} exp\left({\theta^{(1)}}^T x\right) \\ exp\left({\theta^{(2)}}^T x\right) \end{bmatrix} \tag{6.51}$$

We can divide the numerator and denominator of $h_\theta(x)$ by $exp\left({\theta^{(2)}}^T x\right)$ simultaneously, giving us

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \end{bmatrix} = \frac{1}{1+exp(-(\theta^{(2)}-\theta^{(1)})^T x)} \begin{bmatrix} exp(-(\theta^{(2)}-\theta^{(1)})^T x) \\ 1 \end{bmatrix} \tag{6.52}$$

Thus, replacing $\theta^{(2)} - \theta^{(1)}$ with a single parameter vector θ′, we find that softmax regression predicts the probability of (y=2) one of the classes as $\frac{1}{1+exp(-(\theta')^T x)}$, which is a sigmoid function, and that of (y=1) the other class as $1 - \frac{1}{1+exp(-(\theta')^T x)}$. When we use the softmax for C=2, one of the elements in $h_\theta(x)$ is the same as logistic regression output while the other is redundant, because they sum to one.

6.8.3 Softmax layer in neural network

If we want to design a multi-layer neural network for a multi-class classification, we can use a softmax layer as the output layer. Fig.12 shows a neural network with a 4-unit softmax layer as the output layer.

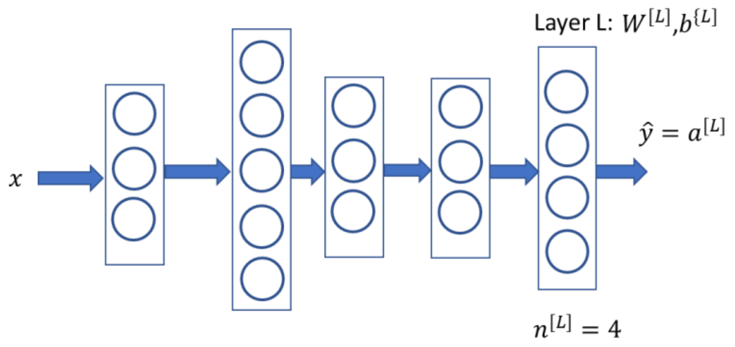


Fig.12 a 4-unit softmax layer for 4-class classification task

The output layer, softmax layer, can be described as

$$z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]} \tag{6.53}$$

Please note that $W^{[L]}$ is equivalent to θ^T in (6.50). For the particular setting in Fig.12, $W^{[L]}$ shape is (4,3), $a^{[L-1]}$shape is (3,1), $b^{[L]}$ shape is (4,1). The softmax activation function calculates the output $a^{[L]}$ as

$$t = exp\left(z^{[L]}\right) \tag{6.54.a}$$

$$a^{[L]} = g^{[L]}\left(z^{[L]}\right) = \frac{t}{\sum_{j=1}^{C} t_j} \tag{6.54.b}$$

where t_j is the element of vector *t*.

```
def softmax(x):
    """
    Compute the sigmoid of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- softmax(x)
    """
    t=np.exp(x)
    s = t/np.sum(t, axis=0)

    return s
```

Now let's consider the loss function for one example, say (*x*,*y*). y is the label in one-hot code format, i.e., for class j, only the jth element is 1 and other elements are all zero. For example,

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

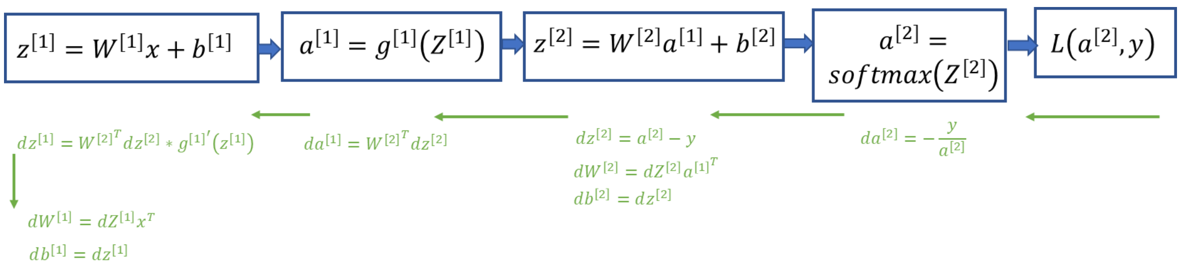
The estimated y is denoted by $\hat{y} = a^{[L]}$ and the loss function is

$$L(\hat{y}, y) = -\sum_{j=1}^{C} y_j \ln\left(\hat{y}_j\right) \tag{6.55.a}$$

For *m* examples, the cost function can be calculated by

$$J\left(\hat{Y}, Y\right) = \frac{-1}{m}\sum\left(Y * \ln\left(A^{[L]}\right)\right) \tag{6.55.b}$$

Note: in (6.55.b), * 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 $\left(n^{[L]}, m\right)$ given or derived from training dataset. $A^{[L]}$ is the outputs of the neural network for m examples, with the same shape of Y.



$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = g^{[1]}(Z^{[1]})$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = softmax(Z^{[2]})$
$L(a^{[2]}, y)$
$dz^{[1]} = W^{[2]^T} dz^{[2]} * g^{[1]'}(z^{[1]})$
$da^{[1]} = W^{[2]^T} dz^{[2]}$
$dz^{[2]} = a^{[2]} - y$
$dW^{[2]} = dZ^{[2]} a^{[1]^T}$
$db^{[2]} = dz^{[2]}$
$da^{[2]} = -\frac{y}{a^{[2]}}$
$dW^{[1]} = dZ^{[1]} x^T$
$db^{[1]} = dz^{[1]}$

Fig.13 Backpropagation of softmax layer for a two-layer neural network (L=2)

The derivative backpropagation is illustrated in Fig. 13 with L=2. It is easy to verify the following equations,

$$da^{[2]} = \frac{dL(\hat{y},y)}{da^{[2]}} = -\frac{y}{a^{[2]}} \tag{6.56.a}$$

$$\frac{\partial a_i^{[2]}}{\partial z_k^{[2]}} = \begin{cases} a_i^{[2]}\left(1 - a_i^{[2]}\right) & i = k \\ -a_i^{[2]} y_k & i \neq k \end{cases} \tag{6.56.b}$$

$$dz_k^{[2]} = \frac{\partial L}{\partial z_k^{[2]}} = \sum_{i=1}^{C} \frac{dL(\hat{y},y)}{da_i^{[2]}} \frac{\partial a_i^{[2]}}{\partial z_k^{[2]}} = -y_k + a_k^{[2]} \sum_{i=1}^{C} y_i = a_k^{[2]} - y_k \tag{6.56.c}$$

Thus,

$$dz^{[2]} = a^{[2]} - y \tag{6.56.d}$$

which is the same as logistic regression (6.35), except that the dimensions of the variables are different. Therefore, the way to build a neural network for multiple-class is pretty 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 examples in a 4-class task is

$$Y_{orig} = [0,3,1,2]$$

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.

In summary, based on the neural networks we implemented before for binary classifications, we modify them by replacing sigmoid activation with softmax activation function (6.54) in the output layer, for forward propagation. The backward propagation does not need to change. The resulting neural network will work for C-class classification tasks, including C=2. The output of the output layer is a vector indicating the probabilities of C classes given input *x*. The predicted class is the class associated with the largest element in the output vector.

6.9 Practice in Python

In this section, we will guide a reader to implement a neural network in python from scratch by an example (note: this example is an assignment from coursera.org online course "neural networks and deep learning" by Andrew Ng). In this example, we are going to build a single hidden layer neural network for a 2-class classification task. For visualization, the dataset has two features with a label. The network architecture is shown in Fig.14. 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.

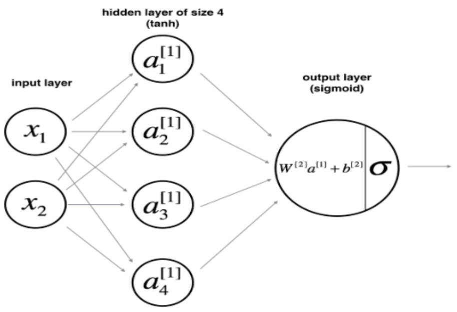


Fig.14 NN model

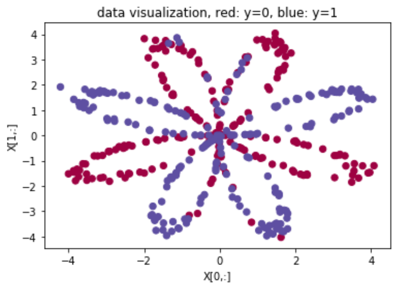


Fig.15 Dataset: *m*=400, X (2,400), Y(1,400)

6.9.1 Import package and load dataset

Let's first import all the packages that we will need. planar_utils provide various useful functions used in this example. The complete code for planar_utils is saved in file planar_utils.py, and provided at the end of this section. The loaded data is visualized in Fig.15.

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset, load_extra_datasets


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

X, Y = load_planar_dataset()
# Visualize the data:
plt.scatter(X[0, :], X[1, :],c=Y[0,:], s=40, cmap=plt.cm.Spectral)
plt.xlabel("X[0,:]")
plt.ylabel("X[1,:]")
plt.title("data visualization, red: y=0, blue: y=1")
plt.show()
```

planar_utils.py:

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

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)


def sigmoid(x):
    """
    Compute the sigmoid of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- sigmoid(x)
    """
    s = 1/(1+np.exp(-x))
    return s

def load_planar_dataset():
    np.random.seed(1)
    m = 400 # number of examples
    N = int(m/2) # number of points per class
    D = 2 # dimensionality
    X = np.zeros((m,D)) # data matrix where each row is a single example
    Y = np.zeros((m,1), dtype='uint8') # labels vector (0 for red, 1 for blue)
    a = 4 # maximum ray of the flower

    for j in range(2):
        ix = range(N*j,N*(j+1))
        t = np.linspace(j*3.12,(j+1)*3.12,N) + np.random.randn(N)*0.2 # theta
        r = a*np.sin(4*t) + np.random.randn(N)*0.2 # radius
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        Y[ix] = j

    X = X.T
    Y = Y.T

    return X, Y

def load_extra_datasets():
    N = 200
    noisy_circles = sklearn.datasets.make_circles(n_samples=N, factor=.5, noise=.3)
    noisy_moons = sklearn.datasets.make_moons(n_samples=N, noise=.2)
    blobs = sklearn.datasets.make_blobs(n_samples=N, random_state=5, n_features=2, centers=6)

    gaussian_quantiles = sklearn.datasets.make_gaussian_quantiles(mean=None, cov=0.5, n_samples=N, n_features=2, n_classes=2, shuffle=True, random_state=None)
    no_structure = np.random.rand(N, 2), np.random.rand(N, 2)

    return noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure
```

6.9.2 Develop the NN model

In this section, we will present the functions (models) which will be used to build the neural network.

Define the size of NN. The size of input layer, n_x, is determined by the shape of dataset. The size of hidden layer is pre-defined by the designer as a constant (we choose 4). The size of output layer is determined by the shape of label Y.

```python
def layer_sizes(X, Y):
    """
    Arguments:
    X -- input dataset of shape (input size, number of examples)
    Y -- labels of shape (output size, number of examples)

    Returns:
    n_x -- the size of the input layer
    n_h -- the size of the hidden layer
    n_y -- the size of the output layer
    """
    ### START CODE HERE ### (≈ 3 lines of code)
    n_x = X.shape[0] # size of input layer
    n_h = 4
    n_y = Y.shape[0] # size of output layer
    ### END CODE HERE ###
    return (n_x, n_h, n_y)
```

Parameter initialization. The shapes of parameters W1, b1, W2 and b2 are defined by the 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*.

```python
# 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 your 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) # we set up a seed so that your output matches ours although the initial
ization is random.

    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y,1))
    ### END CODE HERE ###

    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
```

Loop. The purpose of the loop is to update the parameters to reduce the cost function. In the loop, 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 (output of initialization function)

    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    ### END CODE HERE ###

    # Implement Forward Propagation to calculate A2 (probabilities)
    ### START CODE HERE ### (≈ 4 lines of code)
    Z1 = np.dot(W1, X) + b1
    A1 =  np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    ### END CODE HERE ###

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

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

    return A2, cache
```

The following function is to **compute the cost**. This function is not required to learn the model, but it outputs the values of cost function in different iterations, thus visualizes the learning process.

```
# FUNCTION: compute_cost

def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)

    Arguments:
    A2 -- The sigmoid output of the second activation, of shape (1, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)
    parameters -- python dictionary containing your parameters W1, b1, W2 and b2

    Returns:
    cost -- cross-entropy cost given equation (13)
    """

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

    # Compute the cross-entropy cost
    ### START CODE HERE ### (≈ 2 lines of code)
    logprobs = np.multiply(np.log(A2),Y) +  np.multiply(np.log(1-A2), (1-Y))
    cost = -1/m*np.sum(logprobs)
    ### END CODE HERE ###

    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

```
# FUNCTION: backward_propagation

def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (2, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

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

    # First, retrieve W1 and W2 from the dictionary "parameters".
    ### START CODE HERE ### (≈ 2 lines of code)
    W1 = parameters["W1"]
    W2 = parameters["W2"]

    ### END CODE HERE ###

    # Retrieve also A1 and A2 from dictionary "cache".
    ### START CODE HERE ### (≈ 2 lines of code)
    A1 = cache["A1"]
    A2 =  cache["A2"]
    ### END CODE HERE ###

    # Backward propagation: calculate dW1, db1, dW2, db2.
    ### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on slide above)
    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)
    ### END CODE HERE ###

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

    return grads
```

Parameter update

```
#  FUNCTION: update_parameters

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

    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"
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]
    ### END CODE HERE ###

    # Retrieve each gradient from the dictionary "grads"
    ### START CODE HERE ### (≈ 4 lines of code)
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]
    ## END CODE HERE ###

    # Update rule for each parameter
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = W1 - dW1 * learning_rate
    b1 = b1 - db1 * learning_rate
    W2 = W2 - dW2 * learning_rate
    b2 = b2 - db2 * learning_rate
    ### END CODE HERE ###

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

    return parameters
```

Build the NN model by putting all previous functions together.

```
# FUNCTION: nn_model

def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    """
```

```python
    Arguments:
    X -- dataset of shape (2, number of examples)
    Y -- labels of shape (1, 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)[2]

    # Initialize parameters, then retrieve W1, b1, W2, b2. Inputs: "n_x, n_h, n_y". Outputs = "W1, b1, W2, b2, parameters".
    ### START CODE HERE ### (≈ 5 lines of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]
    ### END CODE HERE ###

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        ### START CODE HERE ### (≈ 4 lines of code)
        # 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)

        ### END CODE HERE ###

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

    return parameters
```

Predict the label ŷ for x, based on the trained model.

```python
# 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 of our model (red: 0 / blue: 1)
    """

    # Computes probabilities using forward propagation, and classifies to 0/1 using 0.5 as the
threshold.
    ### START CODE HERE ### (≈ 2 lines of code)
    A2, cache = forward_propagation(X, parameters)
    predictions = A2 > 0.5
    ### END CODE HERE ###

    return predictions
```

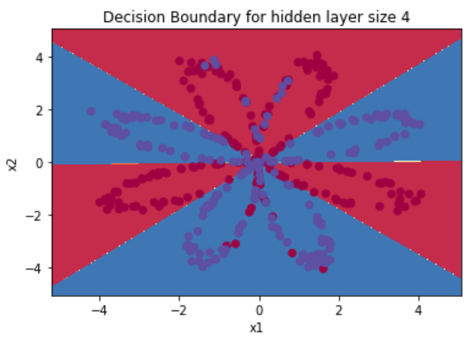
6.9.3 Top-level codes (put all together)

Train the model by calling the function nn_model. The dataset serves the inputs to the function nn_model. The performance is evaluated by predict function.

```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 4, 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(4))

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

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

the output of the above code is:

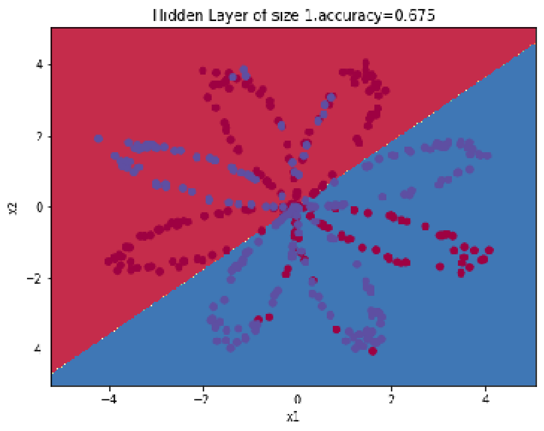
```
Cost after iteration 0: 0.693048
Cost after iteration 1000: 0.288083
Cost after iteration 2000: 0.254385
Cost after iteration 3000: 0.233864
Cost after iteration 4000: 0.226792
Cost after iteration 5000: 0.222644
Cost after iteration 6000: 0.219731
Cost after iteration 7000: 0.217504
Cost after iteration 8000: 0.219440
Cost after iteration 9000: 0.218553
Accuracy of nn:  0.9075
parameters are  {'W1': array([[  0.14446295,  -9.68518983],
       [-11.1325378 ,   3.33887837],
       [-11.46577593, -13.41879291],
       [  9.25561539, -10.19788118]]), 'b1': array([[ 0.01581017],
       [-0.40865387],
       [-0.0649429 ],
       [ 0.01701161]]), 'W2': array([[-11.98548519,   3.44414249,   6.15132104,  10.521138  ]]), 'b2': array([[-0.0660254]])}
```

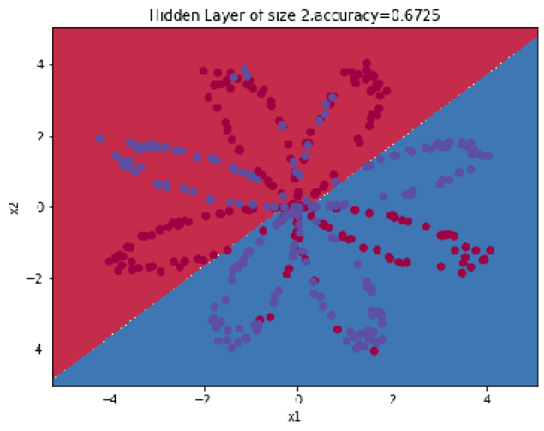


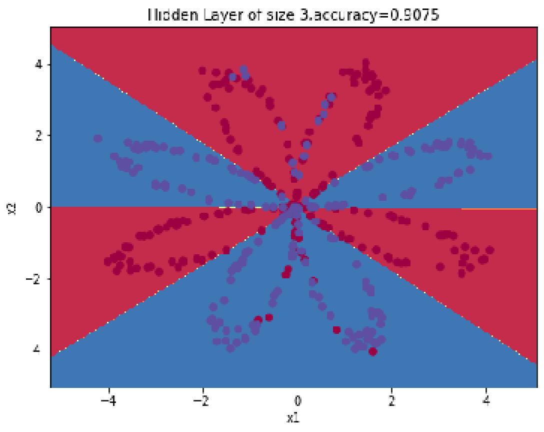
Try different sizes of the hidden layer. The decision boundary for each case is shown in the next page.

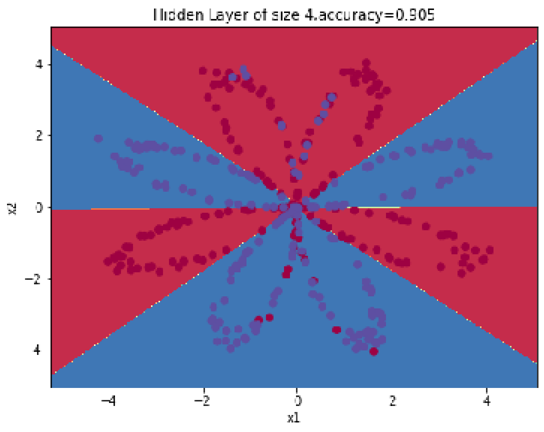
```python
# This may take about 2 minutes to run

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i+1)
    #plt.title('Hidden Layer of size %d, accuracy=' % n_h, accuracy=)
    parameters = nn_model(X, Y, n_h, num_iterations = 5000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = accuracy_score((Y.T).flatten(),predictions.T)
    plt.title('Hidden Layer of size %d,' % n_h+'accuracy='+str(accuracy))
    print ("Accuracy for {} hidden units: {} ".format(n_h, accuracy))
plt.show()
```

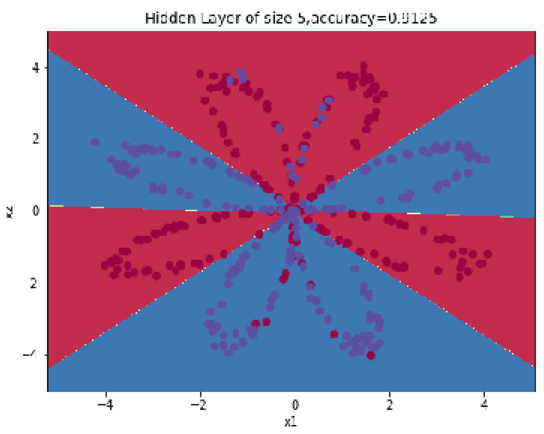
```
Accuracy for 1 hidden units: 0.675
Accuracy for 2 hidden units: 0.6725
Accuracy for 3 hidden units: 0.9075
Accuracy for 4 hidden units: 0.905
Accuracy for 5 hidden units: 0.9125
Accuracy for 20 hidden units: 0.905
Accuracy for 50 hidden units: 0.9075
```

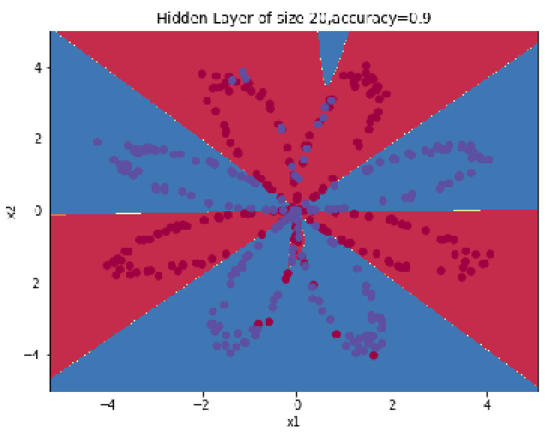


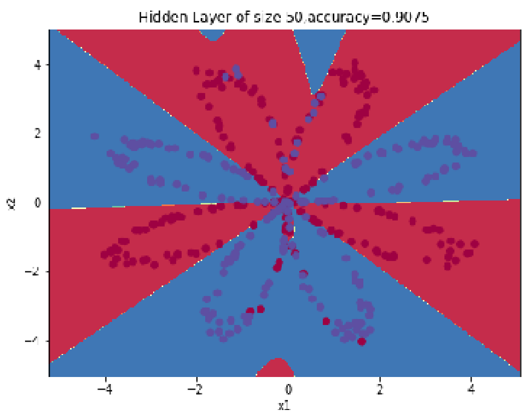








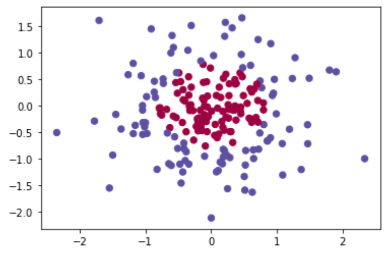


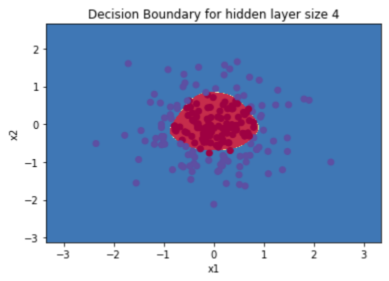


Train the NN model using a different dataset

```
# Datasets
noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure = load_extra_datasets(
)

datasets = {"noisy_circles": noisy_circles,
            "noisy_moons": noisy_moons,
            "blobs": blobs,
            "gaussian_quantiles": gaussian_quantiles}

### START CODE HERE ### (choose your dataset)
dataset = "gaussian_quantiles"
### END CODE HERE ###

X, Y = datasets[dataset]
X, Y = X.T, Y.reshape(1, Y.shape[0])

# make blobs binary
if dataset == "blobs":
    Y = Y%2

# Visualize the data

plt.scatter(X[0, :], X[1, :], c=Y[0,:], s=40, cmap=plt.cm.Spectral);

plt.show()

```

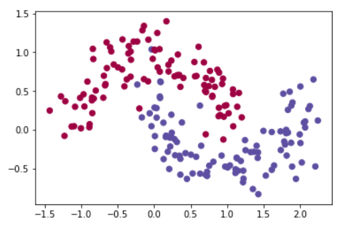
```python
    parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000, print_cost=True)

    # Plot the decision boundary
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    plt.title("Decision Boundary for hidden layer size " + str(4))

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

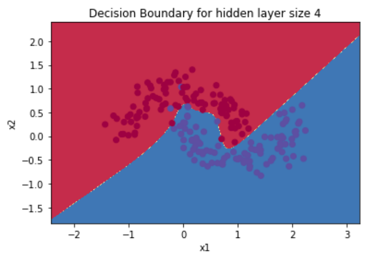




```
Cost after iteration 0: 0.693130
Cost after iteration 1000: 0.102746
Cost after iteration 2000: 0.081207
Cost after iteration 3000: 0.073476
Cost after iteration 4000: 0.081129
Cost after iteration 5000: 0.082204
Cost after iteration 6000: 0.074486
Cost after iteration 7000: 0.069134
Cost after iteration 8000: 0.076900
Cost after iteration 9000: 0.650636
Accuracy of nn:  0.98
```

If “noisy_moons” is selected as the dataset, then the results are shown below:





```
Cost after iteration 0: 0.692992
Cost after iteration 1000: 0.167137
Cost after iteration 2000: 0.069092
Cost after iteration 3000: 0.063716
Cost after iteration 4000: 0.061312
Cost after iteration 5000: 0.059669
Cost after iteration 6000: 0.058364
Cost after iteration 7000: 0.057279
Cost after iteration 8000: 0.056359
Cost after iteration 9000: 0.055569
Accuracy of nn:  0.985
```

6.9.4 Considerations for M-class tasks

In the previous sections, we have demonstrated how to develop and train a two-layer neural network for binary classification in Python programming. 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 using the 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.

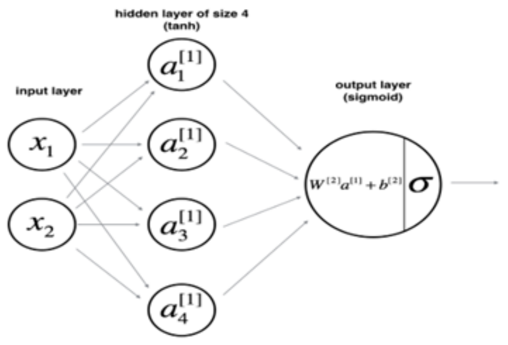
Before working on multi-class tasks, it is helpful to summarize the model we developed in the previous sections. The model is defined by

```
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,

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

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



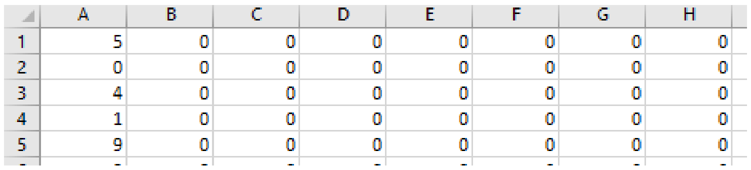
The model implemented consists of two input features ($n_x = 2$), one hidden layer with 4 units($n_h = 4$), and the output layer with single unit. The hidden layer uses Tanh( ) as activation functions and the output layer uses sigmoid function for binary classification.

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. Additionally, the black and white images from NIST were size-normalized and centered to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. This database is well liked for training and testing in the field of machine learning and image processing. It is a remixed subset of the original NIST datasets. One half of the 60,000 training images consist of images from NIST's testing dataset and the other half from NIST's training set. The 10,000 images from the testing set are similarly assembled.

Data preparation

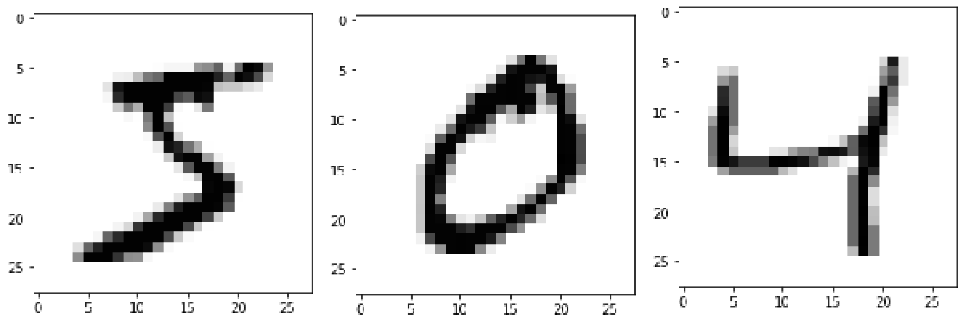
The images from the data set have the size 28 x 28. They are saved in the csv data files mnist_train.csv and mnist_test.csv. Every line of these files consists of 785 numbers between 0 and 255. The first number of each line is the label, i.e. the digit which is depicted in the image. The following 784 numbers are the pixels of the 28 x 28 image. A part of mnist_train.csv opened by Excel is shown below (a line 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

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.

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

fac = 0.99/255
train_imgs = np.asfarray(train_data[:, 1:])*fac+0.01
test_imgs = np.asfarray(test_data[:, 1:])*fac+0.01

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

for i in range(10):
    img = train_imgs[i].reshape((28,28))
    plt.imshow(img, cmap="Greys")
    plt.show()
```



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)

```
# 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)
```

Modification of the previous work

Now, we will highlight the modifications on the previously developed neural network so that the resulting network can recognize the handwritten digits based on the MNIST dataset.

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.16. The input layer and output layer sizes are automatically determined by the shape of X and Y. Thus n_h is assigned to 25.

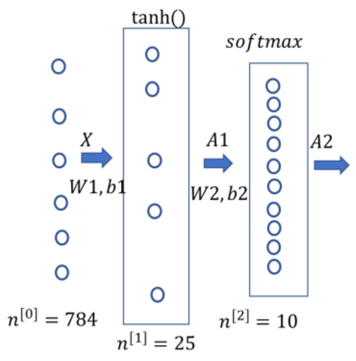


Fig.16 a two-layer neural network for handwritten digit recognition

2) Output layer: we use softmax activation function to replace sigmoid function in the output layer.

```
def softmax(x):
    """
    Compute the sigmoid of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- softmax(x)
    """
    t=np.exp(x)
    s = t/np.sum(t, axis=0)

    return s
```

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 calculated the accuracy.

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

Notes: files for this chapter, C:/machine_learning/NN_nn_overview/: assign_week3.py, planar_utils.py, mnist.py, mnist_train.csv, mnist_train.csv. assign_3_layer.py.

Summary

This chapter provides foundations of neural networks, including mathematical representation (notations), forward propagation, backward propagation, and parameter updating based on gradient descent. Section 6.9 describes the details of implementation of simple neural networks in Python from scratch.

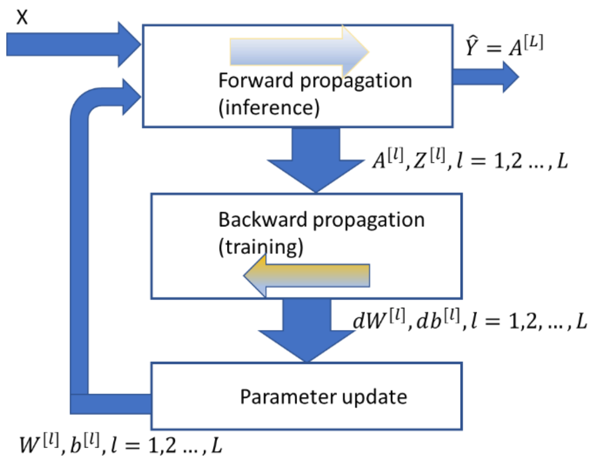

X
Forward propagation (inference)
$\hat{Y} = A^{[L]}$
$A^{[l]}, Z^{[l]}, l = 1,2 \dots, L$
Backward propagation (training)
$dW^{[l]}, db^{[l]}, l = 1,2, \dots, L$
Parameter update
$W^{[l]}, b^{[l]}, l = 1,2 \dots, L$

Fig. 17 Training and inference processes for neural networks

The basic framework for developing neural networks is illustrated in Fig. 17. After the architecture of the network (e.g. size and activation functions) has be 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 to be used to calculate the gradients of cost function with respect to parameters in backward propagation. The parameters are updated based on the gradients. A trained neural network can predict for new input X using forward propagation only. This prediction process is called **inference**.

References

[1] "Neural networks and deep learning" online course at www.coursera.org

[2] Christopher M. Bishop, "Pattern recognition and machine learning".

Exercises

1. Sigmoid function and tanh( ) are two commonly used activation functions for neural networks.

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

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

5) Prove that

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

6) Represent derivatives of $\sigma(z)$ and $tanh(z)$ in terms of sigmoid function $\sigma(\quad)$.

2. Consider a two-layer neural network of the form (6.8), shown in Fig.1. The hidden nonlinear activation functions h( ) are given by sigmoid functions

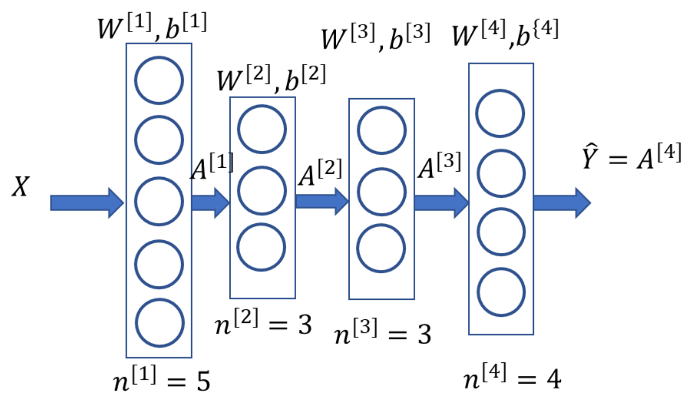
$$\sigma(z) = \frac{1}{1 + exp(-z)}$$

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

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

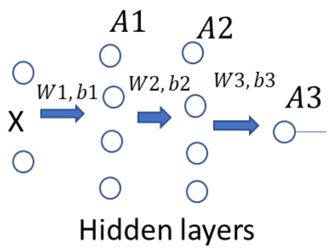
(Hint: first find the relation between $\sigma(z)$ and $tanh(z)$, and then show the relationship between the parameters between the two neural networks)

3. 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 $A^{[1]}, A^{[2]}, A^{[3]}, A^{[4]}$.



4. Run all the codes presented in Sections 6.9.1, 6.9.2, and 6.9.3, and make sure you get the similar results and understand the details of the codes.

5. In sections 6.9.2 and 6.9.3, we trained a neural network with two layers: hidden layer with 4 units and output layer with one sigmoid unit, to classify some generated patterns. In the exercise, we will train a neural network (shown below) with three layers: two hidden layers with 4 units in each layer, and output layer with one sigmoid unit, to classify the same generated patterns. The hidden units use tanh() activation functions.

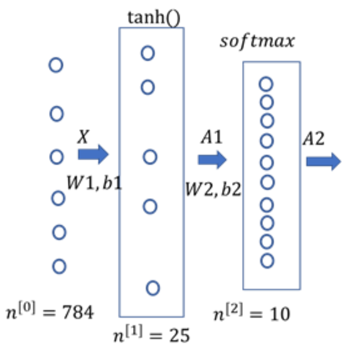


Identify the learning rate, print all trained parameters, plot the cost function versus iterations, plot training example scatterplot and decision boundary, and print the accuracy. Repeat this on four different datasets:

1) Planar dataset.
2) Noisy_circles.
3) Noisy_moons.
4) Gaussian_quantiles

Compare this network with the one developed in section 6.9.3. which one is better? Justify your answer.

6. Before doing this exercise, read Section 6.9.4 *Considerations for M-class tasks* carefully. Develop a neural network illustrated in the figure below to recognize the handwritten digit using MNIST datasets mnist_train.csv and mnist_test.csv.



1) Plot cost versus iteration index.
2) Find the accuracy of your trained neural network on mnist_test.csv.
3) Select a different number of units in hidden layer, and repeat 1) and 2). Compare the results.

7. Consider the neural network in Fig.11. 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:

<u>Equations</u>	<u>matrix shape verification</u>
$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(4,m) = (4,3) \times (3,m) + (4,1)$
$A^{[1]} = g^{[1]}(Z^{[1]})$	$(4,m)$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(1,m) = (1,4) \times (4,m) + (1,1)$
$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$	$(1,m)$

2) Back propagation:

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

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

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

$dZ^{[1]} = \left(W^{[2]T} \cdot dZ^{[2]}\right) * g^{[1]'}(Z^{[1]})$ (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

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

1) Show that if $g^{[1]}(\ )$=tanh( ), the Python statement to calculate $dZ^{[1]}$, line 36 in backward_propagation(parameters, cache, X, Y), is

```
dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
```

2) If sigmoid function $\sigma(\ )$ is used for $g^{[1]}(\ )$, what is the Python statement to calculate $dZ^{[1]}$?

3) Run codes in Sections 6.9.1 through 6.9.3 using $\sigma(\ )$ for activation function in hidden layer. Compare the results with those presented in Sections 6.9.1 through 6.9.3.

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

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

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