File Edit Selection View Go Run Terminal Help OneHiddenLayerNN.ipynb - Machine_Learning_Introduction - Visual Studio Code [Admi...

OneHiddenLayerNN.ipynb

Trusted | Jupyter Server: local | Python 3.7.9 32-bit: Ic

单隐藏层的神经网络

[1]

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.linear_model import LogisticRegressionCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from math import sqrt
from datetime import datetime
```

一、定义数据集

[2]

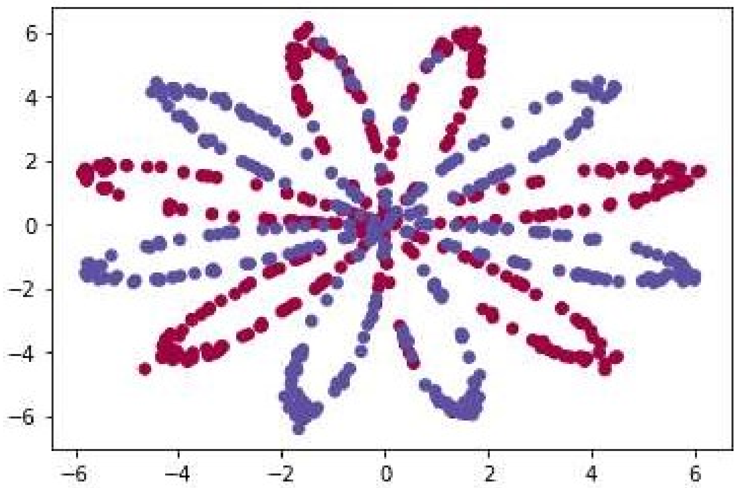
```python
def load_planar_dataset(m=1000):
    np.random.seed(12)
    # m = 1000 # number of samples
    t = 2 # number of type
    N = int(m/t) # number of samples per type
    D = 2 # dimensionality
    X = np.zeros((m, D))
    y = np.zeros((m, 1), dtype = 'uint8') # labels vector (0 for red, 1 for greed)
    r = 6 # maxium ray of the flower

    for i in range(t):
        idx = range(N*i, N*(i+1))
        theta = np.linspace(i*3.12, (i+1)*3.12, N) + np.random.randn(N)*0.2
        radius = r*np.sin(6*theta) + np.random.randn(N)*0.2
        X[idx] = np.c_[radius*np.sin(theta), radius*np.cos(theta)]
        y[idx] = i

    # X = X.T
    # y = y.T

    return X, y
```

二、导出数据并可视化

[3]

```
m = 700
X, y = load_planar_dataset(m=m)
df = pd.DataFrame(np.hstack((X, y)))
df.to_csv("./../dataAI/NeuralNetwork_Data1.txt", sep=",",
float_format="%.7f", index=False, header=None)
df.to_csv("./../dataAI/NeuralNetwork_Data1.csv", sep=",",
float_format="%.7f", index=False, header=["x1", "x2", "y"])
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Spectral)
```

```
<matplotlib.collections.PathCollection at 0x10efae30>
```



三、用Logistic Regression分类

1. 定义绘制决策边界函数

[4]

```python
def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
    x2_min, x2_max = X[:, 1].min(), X[:, 1].max()
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x1_min, x1_max, h), np.arange
(x2_min, x2_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, cmap=plt.cm.Spectral)
```

2. 线性逻辑回归分类

[5]

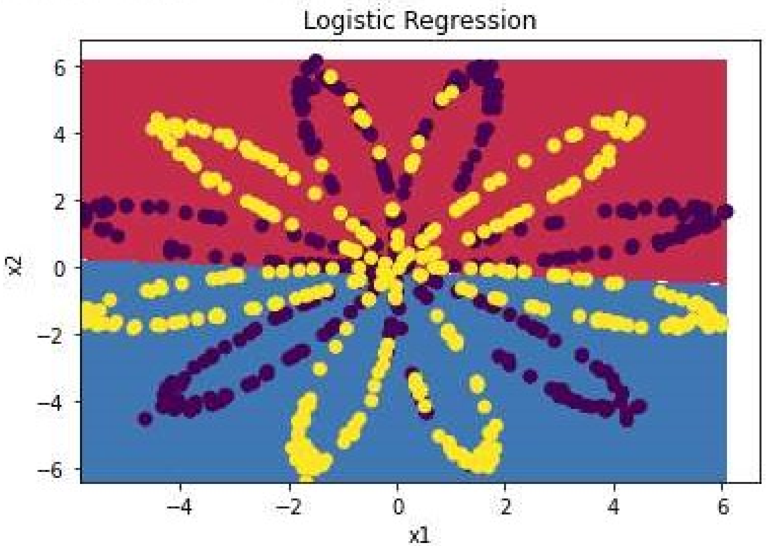
```
logi_reg = LogisticRegressionCV(cv=3)
logi_reg.fit(X, y.ravel())
```

```
LogisticRegressionCV(cv=3)
```

[6]

```
plot_decision_boundary(lambda x:logi_reg.predict(x), X, y)
plt.title("Logistic Regression")
plt.scatter(X[:, 0], X[:, 1], c=y)

y_predict = logi_reg.predict(X)
train_rmse_score = sqrt(mean_squared_error(y, y_predict))
train_r2_score = r2_score(y, y_predict)
print("所有数据集上得分: {:.7f} -- {:.7f}".format(train_rmse_score,
 train_r2_score))
```

```
所有数据集上得分: 0.6358347 -- -0.6171429
```

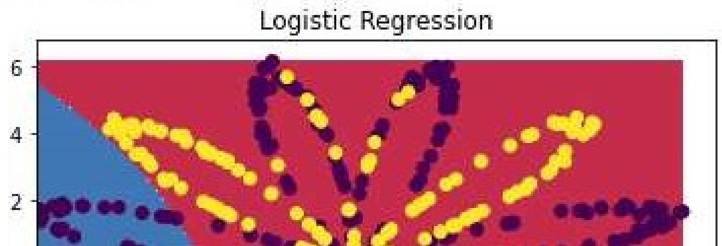


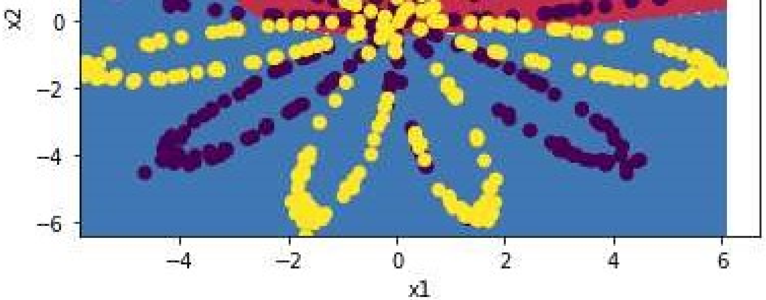
3. 多项式逻辑回归

[9]

```
poly_logi_reg = Pipeline([
            ("multi_feature", PolynomialFeatures(degree=3)),
            ("std_scaler", StandardScaler()),
            ("logi_reg", LogisticRegressionCV(cv=3))
        ])
poly_logi_reg.fit(X, y)

plot_decision_boundary(lambda x:poly_logi_reg.predict(x), X, y)
plt.title("Logistic Regression")

plt.scatter(X[:, 0], X[:, 1], c=y)

y_predict = poly_logi_reg.predict(X)
train_rmse_score = sqrt(mean_squared_error(y, y_predict))
train_r2_score = r2_score(y, y_predict)
print("所有数据集上得分: {:.7f} -- {:.7f}".format(train_rmse_score,
 train_r2_score))
```

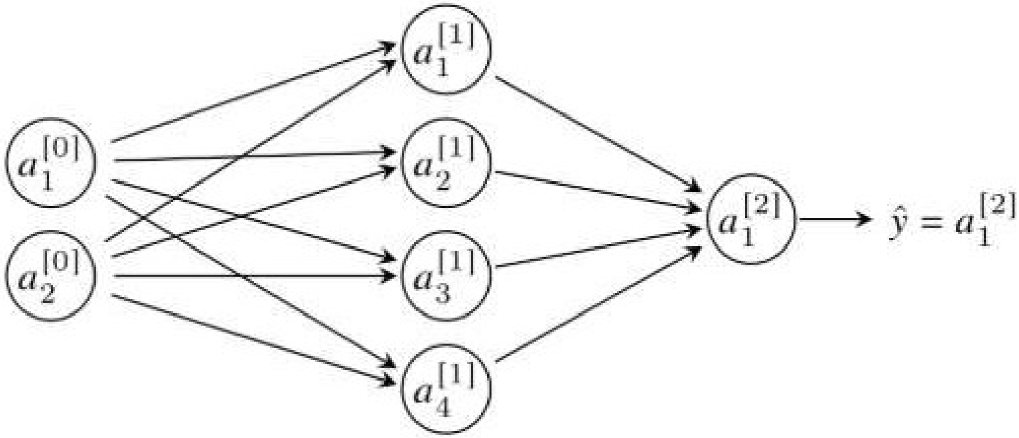
所有数据集上得分: 0.6633250 -- -0.7600000





四、Neural Network with One Hidden Layer

1. 2-Layer NN模型



2. NN算法步骤

2. NN算法步骤

- 确定结构，即输入层、隐藏层及输出层的单元数
- 初始化权矩阵$W^{[l]}$、$b^{[l]}, l=1,2,\cdots,L$
- 利用正向传播过程，计算预测值y_{pred}及损失函数$L(\hat{y},y)$
- 利用反向传播过程，计算导数$dW^{[2]}$、$db^{[2]}$、$dW^{[1]}$、$db^{[1]}$
- 利用梯度法更新参数W^2、$b^{[2]}$、$W^{[1]}$和$b^{[1]}$

3. 定义结构

[10]

```
# define neural network layer sizes
def neural_network_layer_sizes(X, Y):
    '''
    Arguments:
        X -- input dataset of shape(feature size, number of
examples)
        y -- labels of shape (output size, number of examples)

    Returns:
```

```
        n_x -- The size of input layer
        n_h -- The size of hidden layer
        n_y -- The size of output layer
    '''
    n_x = X.shape[0]
    n_h = 4
    n_y = Y.shape[0]

    return (n_x, n_h, n_y)
```

4. 初始化参数

[11]

```
# define initialize parameters
def initialize_parameters(n_x, n_h, n_y):
    '''
    Argument:
        n_x -- The size of input layer
        n_h -- The size of hidden layer
        n_y -- The size of output layer

    Returns:
        W1 -- weight matrix for layer 1 of shape (n_h, n_x)
        b1 -- bias vetor for layer 1 of shape(n_h, 1)
```

```
        W2 -- weight matrix for layer 2 of shape (n_y, n_h)
        b2 -- bias vetor for layer 2 of shape(n_y, 1)
    '''

    np.random.seed(12)

    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))

    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
```

5. 正向预测（Forward Propagation）

1）迭代公式

对于给定的样本$x^{(i)}$，正向过程如下：

- 正向计算过程：
  - 分量格式
$$\begin{cases} z^{[1](i)} &= W^{[1]}x^{(i)} + b^{[1]} \\ a^{[1](i)} &= g^{[1]}(z^{[1](i)}) \\ z^{[2](i)} &= W^{[2]}a^{[1](i)} + b^{[2]} \\ a^{[2](i)} &= \sigma(z^{[2](i)}) = \hat{y}^{(i)} \end{cases}$$
  - 向量格式
$$\begin{cases} Z^{[1]} &= W^{[1]}X^{+}b^{[1]} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= \sigma(Z^{[2]}) = \hat{y} \end{cases}$$
- 计算预测值：
$$y_p^{(i)} = \begin{cases} 1 & \text{if} \quad \hat{y}^{(i)} >= 0.5 \\ 0 & \text{otherwise} \end{cases}$$
- 成本函数：
$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right)$$

2）算法的实现

- 定义激活函数

[12]

```python
def sigmoid(Z):
    return 1/(1+np.exp(-Z))
```

[13]

```python
def ReLU(Z):
    return np.maximum(0, Z)
```

- 计算预测值

[14]

```python
def forward_propagation(X, parameters):
    """
    Arguments:
        X -- input data of size (n_x, m)
        parameters -- python dictionary containing the initialize
parameters

    Returns:
        A2 -- The sigmoid output of the second activation
```

```python
        cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # FP
    Z1 = np.dot(W1, X) + b1
    # A1 = np.tanh(Z1)
    A1 = ReLU(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)


    assert(A2.shape == (b2.shape[0], X.shape[1]))

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

    return A2, cache
```

- 成本函数

[15]

```python
def J(A2, y, parameters):
    m = y.shape[1]
```

```
    m = y.shape[1]

    cost = -1/m*np.sum(y*np.log(A2)+(1-y)*np.log(1-A2))
    cost = np.squeeze(cost)

    assert(isinstance(cost, float))

    return cost
```

6. 反向计算导数（Backward Propagation）

{}

1）迭代公式

- 分量格式 $$\begin{cases} dz^{[2](i)} = a^{[2](i)} - y^{(i)} \\ dW^{[2]} = dz^{[2](i)} a^{[1](i)T} \\ db^{[2]} = dz^{[2](i)} \\ dz^{[1](i)} = W^{[2]T} dz^{[2](i)} * g^{[1]'}(z^{[1]}) \\ dW^{[1]} = dz^{[1](i)} a^{[0](i)T} \ (a^{[0](i)} = x^{(i)}) \\ db^{[1]} = dz^{[1](i)} \end{cases}$$

- 向量格式 $$\begin{cases} dZ^{[2]} &= A^{[2]} - y \\ dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\ db^{[2]} &= \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \\ dZ^{[1]} &= W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]}) \\ dW^{[1]} &= \frac{1}{m} dZ^{[1](i)} A^{[0]T} \ (A^{[0]} = X) \\ db^{[1]} &= \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True) \end{cases}$$

- Tips:
  - To compute $dZ^{[1]}$, you'll need to compute $g^{[1]'}(Z^{[1]})$.
  - If $g^{[1]}(\cdot)$ is the tanh function and let $a = g^{[1]}$, then $g^{[1]'} = 1 - a^2$.
  - If $g^{[1]}(\cdot)$ is the sigmoid function, then $g^{[1]'} = a(1 - a)$, where $a = g^{[1]}$.
  - If $g^{[1]}(\cdot)$ is the ReLU function, then $g^{[1]'} = 1(x > 0), where a$ = g^{1}$.

2）代码实现

[16]

```
def backward_propagation(X, y, parameters, cache):
    """

    Arguments:
        X -- input data of size (n_x, m)
        y -- "true" label vector of shape(n_y, m)
```

```
        parameters -- python dictionary containing the initialize
parameters
        cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"

    Returns:
        grads -- python dictonary containing the gradients with
respect to different parameters
    """
    m = X.shape[1] # number of samples

    W1 = parameters["W1"]
    W2 = parameters["W2"]

    A1 = cache["A1"]
    A2 = cache["A2"]

    # BP
    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)

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

    return grads
```

7. 更新参数

General gradient descent rule: , where is the learning rate and represents a parameter.

[17]

```
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
    """

    # old paremeters
    W1 = parameters["W1"]
```

```python
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # gradient values
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]

    # update parameters
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

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

    return parameters
```

8. 算法整合

[18]

```python
def neural_network_model(X, y, n_h, learning_rate = 0.9
```

```
def neural_network_model(X, y, n_h, learning_rate = 0.5,
max_iterations = 10000, err_torlance = 1e-12, print_cost = False):
    """
    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 and then can be
used to predict.
    """
    tic = datetime.now()

    np.random.seed(12)
    # layer of neural network
    n_x = neural_network_layer_sizes(X, y)[0]
    n_y = neural_network_layer_sizes(X, y)[2]

    # initialize parameters
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
```

```
    cost_value_list = list(np.zeros((max_iterations, 1)))
    last_cost = 0
    # loop (using gradient descent)
    for i in range(max_iterations):
        # 1. Forward Propogation
        A2, cache = forward_propagation(X, parameters)
        # 2. Compute Cost Function
        cost = J(A2, y, parameters)
        # 3. Backward Propogation
        grads = backward_propagation(X, y, parameters, cache)
        # 4. Update Parameters
        parameters = update_parameters(parameters, grads,
learning_rate=learning_rate)
        # 5. Print Cost Values
        if print_cost and i % 1000 == 0:
            print
("\n==============================================")
            print("Cost value after iteration {:d}: {:f}".format
(i, cost))

        # 6. Exit Control
        cost_value_list[i] = cost
        if np.abs(cost-last_cost) < err_torlance:
            cost_value_list = cost_value_list[0:i]
            break
        last_cost = cost

    toc = datetime.now()
```

```
    toc = datetime.now()
    delta = toc - tic
    # print("NN Model Train Time: {:f} S.".format
(delta.total_seconds()))
    # return optimum parameters
    return parameters, cost_value_list, delta.total_seconds()
```

9. 模型测试

[19]

```
def nn_model_test(learning_rate=1.2):
    parameters, cost_list = neural_network_model(X.T, y.T, n_h=4,
learning_rate=learning_rate,  max_iterations=10000, err_torlance
= 1e-12, print_cost=True)
    print("W1 = " + str(parameters["W1"]))
    print("b1 = " + str(parameters["b1"]))
    print("W2 = " + str(parameters["W2"]))
    print("b2 = " + str(parameters["b2"]))

    plt.plot(range(len(cost_list)), cost_list)
```

[21]

```
nn_model_test(2.1)
```

```
============================================================
Cost value after iteration 0: 0.693077
```

```
==========================================================
Cost value after iteration 1000: 0.440976

==========================================================
Cost value after iteration 2000: 0.437611

==========================================================
Cost value after iteration 3000: 0.435636

==========================================================
Cost value after iteration 4000: 0.434290

==========================================================
```

10. 预测

[22]

```
def predict(X, parameters):
    """
    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)
    """



    A2, cache = forward_propagation(X, parameters)
    predictions = np.array(A2>=0.5, dtype=int)

    return predictions
```

[23]

```
def predict_test_case():
    np.random.seed(12)
    X_test = np.random.randn(2, 3)
    parameters = {'W1': np.array([[-0.00615039,  0.0169021 ],
        [-0.02311792,  0.03137121],
        [-0.0169217 , -0.01752545],
        [ 0.00935436, -0.05018221]]),
     'W2': np.array([[-0.0104319 , -0.04019007,  0.01607211, 
0.04440255]]),
     'b1': np.array([[ -8.97523455e-07],
        [  8.15562092e-06],
        [  6.04810633e-07],
        [ -2.54560700e-06]]),
     'b2': np.array([[  9.14954378e-05]])}
    return X_test, parameters
```

[24]

```
X_test, parameters_test = predict_test_case()
```

```
predictions = predict(X_test, parameters_test)
print("Predict mean: {:f}".format(np.mean(predictions)))
```

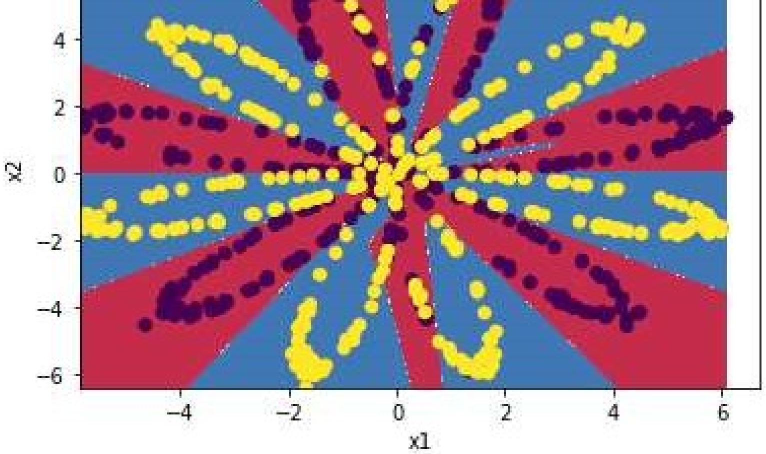
```
Predict mean: 0.666667
```

11. 绘制决策边界及模型评价

[25]

```
parameters, cost_list, training_time = neural_network_model(X.T,
y.T, n_h=17, learning_rate=5.2,  max_iterations=10000,
err_torlance = 1e-12, print_cost = False)

plot_decision_boundary(lambda x:predict(x.T, parameters), X, y)
plt.title("Neural Network Classfication")

plt.scatter(X[:, 0], X[:, 1], c=y)

y_predict = predict(X.T, parameters)
rmse_score = sqrt(mean_squared_error(y.T, y_predict))
accuracy_score = np.sum(np.array(y.T==y_predict, dtype=int)) / len
(y)
print("所有数据集上得分：{:.7f} -- {:.7f}，耗时{:f}秒！".format
(rmse_score, accuracy_score, training_time))
```

```
所有数据集上得分：0.2420153 -- 0.9414286，耗时6.830964秒！
```



12. 不同隐藏层节点的比较

[32]

```
plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 7, 10, 15, 20, 50, 78, 99]
for i, n_h in enumerate(hidden_layer_sizes):
    # 决策边界
    plt.subplot(len(hidden_layer_sizes), 2, i*2+1)
    plt.title('The Size Hidden Layer: %d' % n_h)
    parameters, cost_list, training_time = neural_network_model(X.T, y.T, n_h=n_h, learning_rate=0.9,  max_iterations=10000, err_torlance = 1e-12, print_cost = False)
    plot_decision_boundary(lambda x:predict(x.T, parameters), X, y)

    plt.scatter(X[:, 0], X[:, 1], c=y)
```

```
    # 函数值
    plt.subplot(len(hidden_layer_sizes), 2, i*2+2)
    plt.title('The Cost Function value descent')
    plt.plot(range(len(cost_list)), cost_list)



    y_predict = predict(X.T, parameters)
    rmse_score = sqrt(mean_squared_error(y.T, y_predict))
    accuracy_score = np.sum(np.array(y.T==y_predict, dtype=int)) /
len(y)
    print("模型（Hidden Units = {:d}）得分：{:.7f} -- {:.7f}，耗时
{:f}秒！".format(n_h, rmse_score, accuracy_score, training_time))
```

```
模型（Hidden Units = 1）得分：0.7071068 -- 0.5000000，耗时1.514948秒！
模型（Hidden Units = 2）得分：0.7071068 -- 0.5000000，耗时1.274590秒！
模型（Hidden Units = 3）得分：0.7071068 -- 0.5000000，耗时1.513126秒！
模型（Hidden Units = 4）得分：0.7071068 -- 0.5000000，耗时1.619667秒！
模型（Hidden Units = 5）得分：0.7071068 -- 0.5000000，耗时1.735398秒！
模型（Hidden Units = 7）得分：0.7071068 -- 0.5000000，耗时1.992669秒！
模型（Hidden Units = 10）得分：0.7071068 -- 0.5000000，耗时2.357605秒！
模型（Hidden Units = 15）得分：0.7071068 -- 0.5000000，耗时2.919191秒！
模型（Hidden Units = 20）得分：0.7071068 -- 0.5000000，耗时3.554785秒！
模型（Hidden Units = 50）得分：0.7071068 -- 0.5000000，耗时9.611463秒！
模型（Hidden Units = 78）得分：0.7071068 -- 0.5000000，耗时14.464670秒！
模型（Hidden Units = 99）得分：0.7071068 -- 0.5000000，耗时18.351490秒！
```