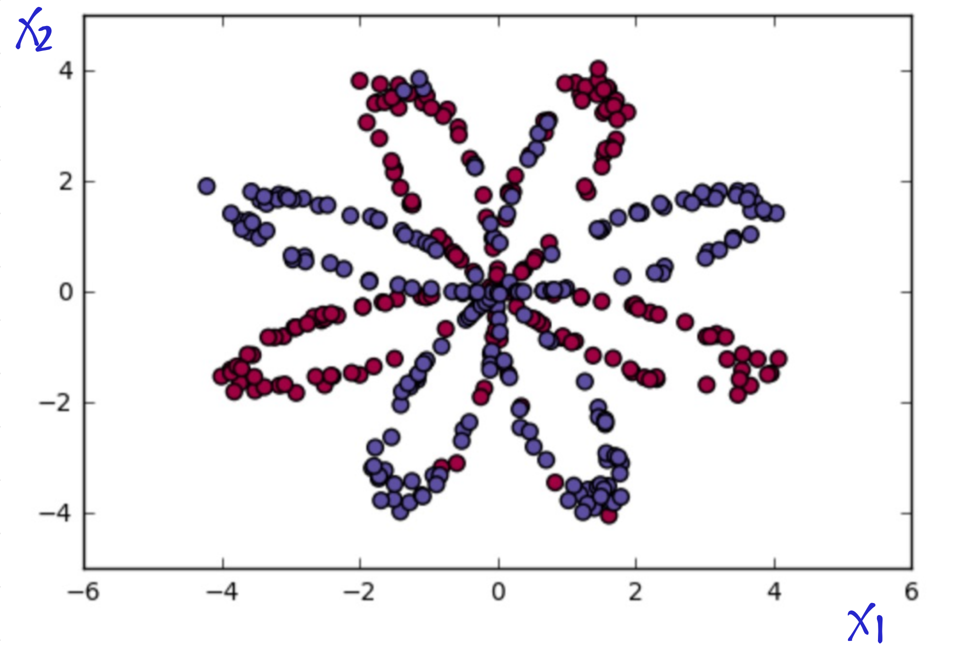
§ Planar data classification with a hidden layer

1. Dataset

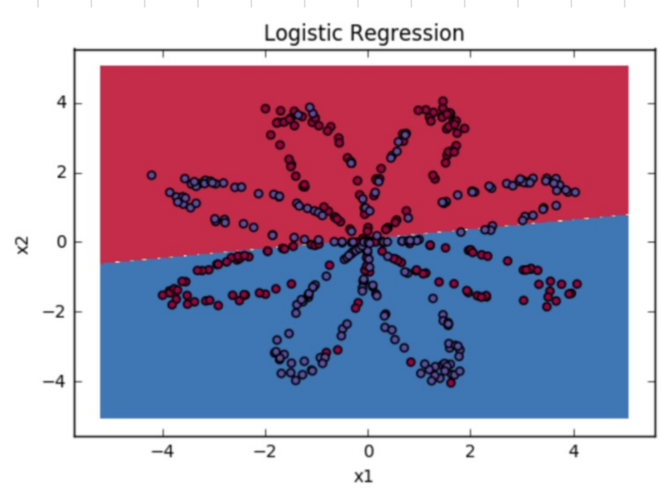
X, Y = load_planar_dataset( )

X (2x400) – a numpy-array contains your features (X_1, X_2)

Y (1x400) – a numpy-array contains your labels (red: 0, blue: 1)



2. Logistic Regression (Sample)

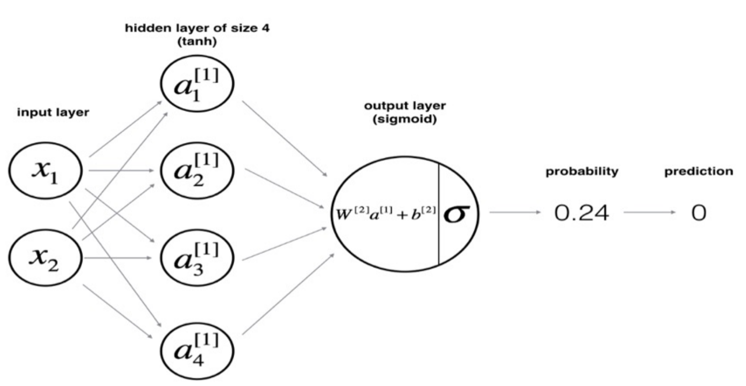


The logistic regression doesn't perform well since the dataset is not linearly separable.

Expected Output:

Accuracy	47%

3. Neural Network model



For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1](i)}$$
$$a^{[1](i)} = \tanh(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2](i)}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)})$$
$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[2](i)}\right) \right)$$

Reminder: The general methodology to build a Neural Network is to:
1. Define the neural network structure (# of input units, # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
    - Implement forward propagation
    - Compute loss
    - Implement backward propagation to get the gradients
    - Update parameters (gradient descent)

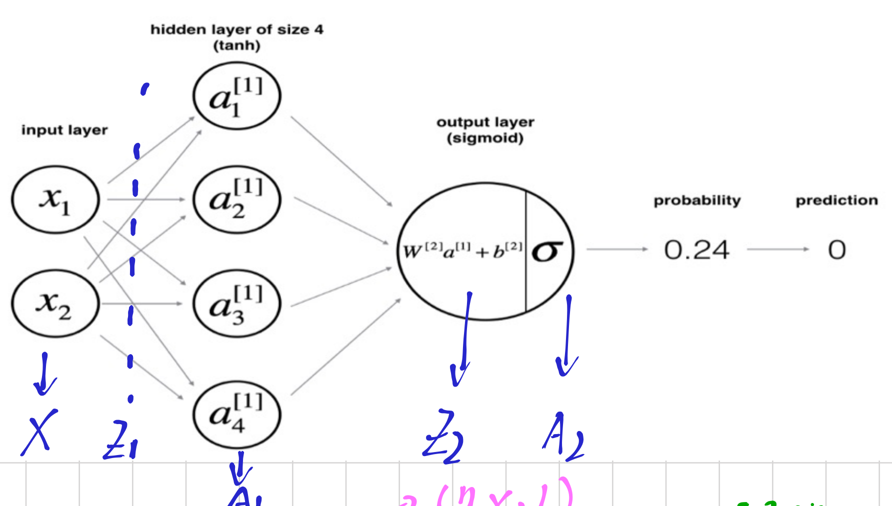
(1) define the neural network structure

n_x : the size of the input layer
n_h : the size of the hidden layer
n_y : the size of the output layer

(2) initialize the model's parameters

W1 = np.random.randn(n_h, n_x)*0.01 → weight matrix
b1 = np.zeros((n_h,1)) → bias vector
W2 = np.random.randn(n_y, n_h)*0.01 → weight matrix
b2 = np.zeros((n_y, 1)) → bias vector

(3) Loop



Z1 = np.dot(W1, X)+b1 → $z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1](i)}$
(n_h, n_x) (n_x, 1) (n_h, 1)

A1 = np.tanh(Z1) → $a^{[1](i)} = \tanh(z^{[1](i)})$
(n_h, 1)

Z2 = np.dot(W2, A1)+b2 → $z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2](i)}$
(n_y, n_h) (n_h, 1) (n_y, 1)

A2 = sigmoid(Z2) → $a^{[2](i)} = \sigma(z^{[2](i)})$
(n_y, 1)

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right)$$

```
logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1-A2), (1-Y))
cost = -1/m * np.sum(logprobs)
```

(n_y, 1) (n_y, 1) (n_y, 1)

P.S. Broadcasting

When operating on two arrays, Numpy compares their shapes element-wise.

Two dimensions are compatible when:

① they are equal, or
② one of them is 1

other → Value Error

e.g.

① A: 8 × 1 × 6 × 1
B: 7 × 1 × 5
result: 8 × 7 × 6 × 5

② A: 15 × 3 × 5
B: 15 × 1 × 5
result: 15 × 3 × 5

③ A: 8 × 4 × 3
B: 2 × 1
result: Error

not ①/②

Backward propagation

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]}a^{[1]^T}$	$dW^{[2]} = \frac{1}{m}dZ^{[2]}A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m}np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T}dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T}dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]}x^T$	$dW^{[1]} = \frac{1}{m}dZ^{[1]}X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m}np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Andrew Ng

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

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

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

(n_h, n_y) (n_y, 1) same

(n_h, 1)

(n_h, 1)

Broadcasting

(n_h, 1)

attention the difference etween np.dot() and np.multiply

Gradient rule

$\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$

(4) Integrate parts in nn_model

① Initialize parameters

parameters = initialize_parameters (n_x, n_h, n_y)

② Loop

for i in range (0, num_iterations):

i. Forward propagation

A2, cache = forward_propagation (X, parameters)

ii. Cost function

cost = compute_cost (A2, Y, parameters)

iii. Backpropagation

grads = backward_propagation (parameters, cache, X, Y)

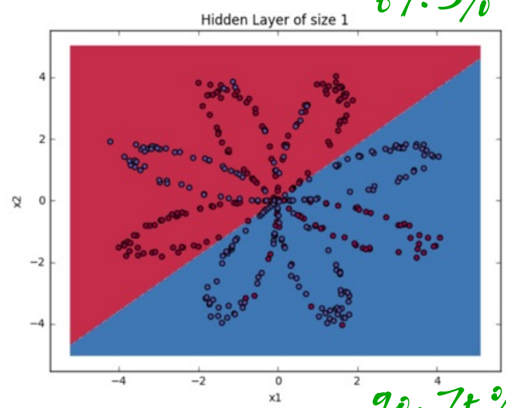
iv. Parameters update

parameters = update_parameters (parameters, grads)

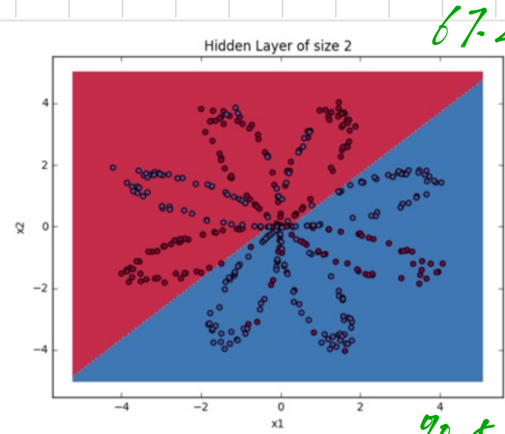
(5) Turning hidden layer size

change the hidden layer size from 1 → 2 → 3 → 4 → 5 → 20 → 50

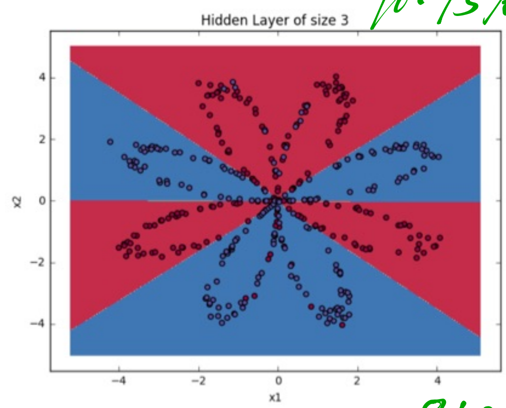
67.5%



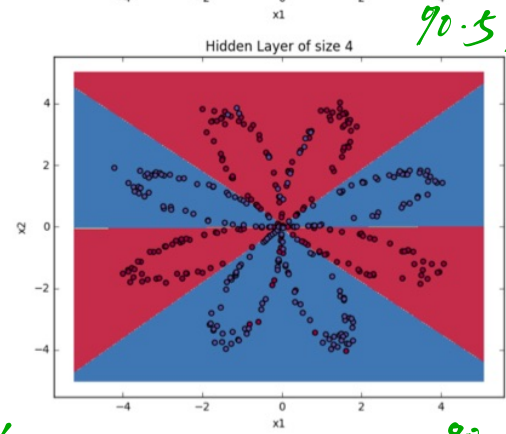
67.25%



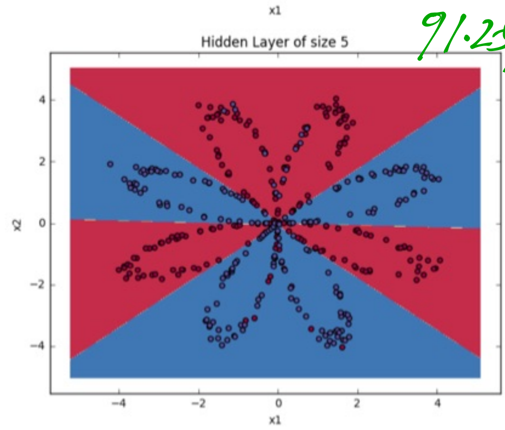
90.75%



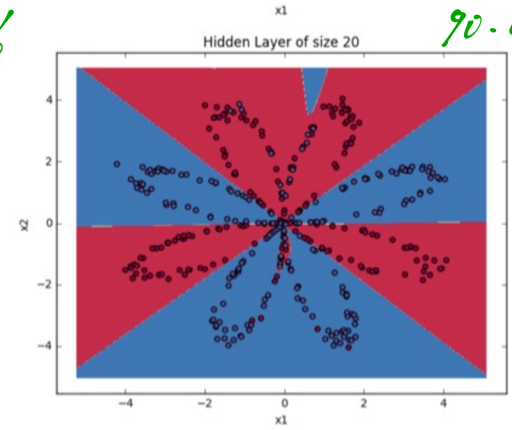
90.5%



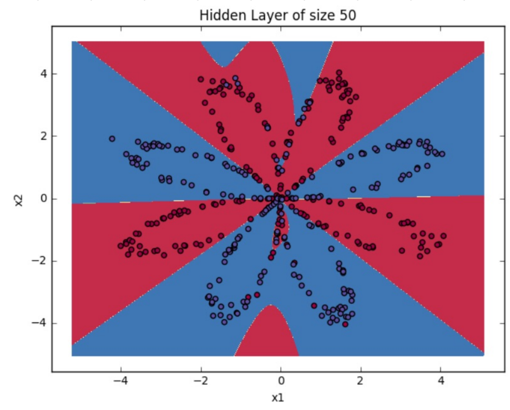
91.25%



90.0%



90.25%



Interpretation:
The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data.
The best hidden layer size seems to be around n_h = 5. Indeed, a value around here seems to fits the data well without also incurring noticable overfitting.
You will also learn later about regularization, which lets you use very large models (such as n_h = 50) without much overfitting.