Deep Learning in Python

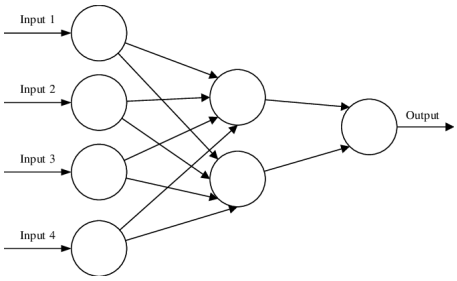
Contents

1 Neural Networks and Deep Learning

1.1 Logistic Regression as a Neural Network

1.1.1 Introduction to Logistic Regression

A single **neural network** can be built off of an input (x), an activation layer known as a *neuron*, and producing an output (y). A larger neural network is then formed by taking many of the single neurons and stacking them together. Each *feature* $(x_1, x_2, ..., x_n)$ can be used as an input to the activation layers to produce our output y. For the below example, we say that the layer in the middle is *densely connected* since every feature is in input.



To store an image, your computer stores three different matrices corresponding to the red, green, and blue channel (RGB values). So if your input image is 64x64 pixels, you will have three 64x64 matrices. To unroll these values into a **feature vector**, we will add values from all 3 vectors into a single x vector, where n_x is the number of features in the vector (in this case, 12288).

In **binary classification**, our goal is to learn a classifier that can input an image represented by feature vector x and predict whether the corresponding label y is a 1 or 0 (1 for cat, 0 for non cat).

Here is some common notation we will be using:

- Single training example: (x,y) where $x \in \mathbb{R}^{n_x}$, y $\in \{0, 1\}$
- M training examples: $\{(x^1, y^1), ..., (x^m, y^m)\}$
- Matrix X: $\begin{bmatrix} | & | & ... & | \\ x^1 & x^2 & ... & x^m \\ | & | & ... & | \end{bmatrix}$ where rows = n_x, columns = m. In Python, X.shape = (n_x, m).
- Matrix Y: $\begin{bmatrix} y^1, y^2, ..., y^m \end{bmatrix}$ where Y.shape = (1, m)

Given x, we want an estimate known as $\hat{y}$ = P(y=1|x) given the following parameters: $x \in \mathbb{R}^{n_x}$, $w \in \mathbb{R}^{n_x}$, and $b \in \mathbb{R}$. We want out output to be $0 \leq \hat{y} \leq 1$, so we will use the **sigmoid function** to find our output, which will be:

$$\hat{y} = \sigma(z) \text{ where } z = w^T x + b \text{ and } \sigma(z) = \frac{1}{1 + e^{-z}}$$

If z is large, then $\sigma(z)$ will be very close to 1. But if z is a large negative number, then $\sigma(z)$ will be very close to 0. So given $\{(x^1, y^1), ..., (x^m, y^m)\}$ we want $\hat{y}^i \approx y^i$

1.1.2 Logistic Regression Cost Function

We will associate x^i, y^i, and z^i with the i^{th} training example of our data. We will need to define a **loss function**, with respect to a single training example, to measure how good our output ($\hat{y}$) is when the true label is y. Since we are using gradient descent, we will define the following loss function:

$$\mathcal{L}(\hat{y}, y) = -(y\, log(\hat{y}) + (1 - y)\, log(1 - \hat{y}))$$

We want this loss function to be as small as possible. Lets look at the two cases:

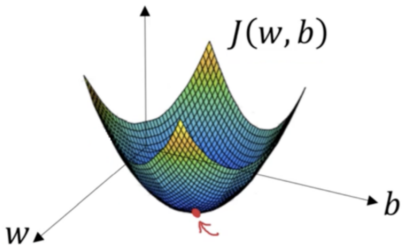
If y=1: $\mathcal{L}(\hat{y}, y) = -y\, log(\hat{y})$ and we want this to be as small as possible ($\hat{y}$ large).
If y=0: $\mathcal{L}(\hat{y}, y) = -\, log(1 - \hat{y})$ and we want this to be large ($\hat{y}$ small).

To train the parameters w and b, we need to define a **cost function**, which measures how well your doing on an entire training set (cost of the parameters). We will define this as:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^i, y^i) = -\frac{1}{m}\sum_{i=1}^{m}[(y^i\, log(\hat{y}^i) + (1-y^i)\, log(1-\hat{y}^i))]$$

1.1.3 Gradient Descent

We want to find the values of w and b that will *minimize* the cost function J(w,b). Our cost function that we defined is convex (only one minimum). So we will initialize (w,b) to a random value, typically zero, and will take steps downhill in the steepest direction it can. Eventually it will converge to a minimum value and find our parameter values.



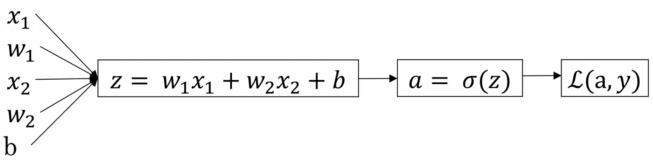
Gradient descent will repeatedly update the value of w and b with the formula:

$$w = w - \alpha\frac{\partial J(w,b)}{\partial w}\ ,\ b = b - \alpha\frac{\partial J(w,b)}{\partial b}$$

where α is our learning rate that we set multiplied by the partial derivative (since there are two variables) of the cost function with respect to the given parameter.

We want to modify out w and b parameters in order to reduce the loss when performing gradient descent on our Logistic Regression. We can set up a computation graph to find the derivatives through **backpropagation**. We will do this for a *single* training example, lets remind ourselves of our equations and the graph:

$z = w^T x + b$
$\hat{y} = a = \sigma(z)$
$\mathcal{L}(a,y) = -(y\, log(a) + (1-y)\, log(1-a))$



The first back step is to compute "da" $= \frac{\partial\mathcal{L}(a,y)}{\partial a} = \frac{-y}{a} - \frac{1-y}{1-a}$

Next we step back again and compute "dz" $= \frac{\partial\mathcal{L}(a,y)}{\partial z} = \frac{\partial\mathcal{L}}{\partial a} * \frac{\partial a}{\partial z} = a(1-a) * (\frac{-y}{a} + \frac{1-y}{1-a}) = a - y$

The final step back is to find how much to change our w_1, w_2, and b values. We can do this by:

- Calculating: $\frac{\partial\mathcal{L}}{\partial w_1}$ = "dw_1" $= x_1 * dz$, "dw_2" $= x_2 * dz$, and "db" $= dz$
- Then update our variables: $w_1 = w_1 - \alpha * dw_1$, $w_2 = w_2 - \alpha * dw_2$, and $b = b - \alpha * dz$

Now we want to **perform gradient descent on m examples**. This will use the cost function (not the loss function like we did on a single example). We will write sudo-code for Python that implements this m example gradient descent (assume that $n_x = 2$):

Initialize: J=0, dw_1=0, dw_2=0, db=0
for i=1 to m:
$z^i = w^T x^i + b$
$z^i = \sigma(z^i)$
$J+ = -[(y^i\, log(a^i) + (1-y^i)\, log(1-a^i))]$
$dz^i = a^i - y^i$
$dw_1+ = x_1^i * dz^i$
$dw_2+ = x_2^i * dz^i$
$db+ = dz^i$

After the loop, we then take the average and update our varaibles:

J /= m
dw_1 /= m
dw_2 /= m
db /= m
$w_1 = w_1 - \alpha * dw_1$
$w_2 = w_2 - \alpha * dw_2$
$b = b - \alpha * db$

1.1.4 Vectorizing Logistic Regression

We often find ourselves training on big data sets and we need our code to run as fast as possible. This is where **vectorization** comes into play. One example is when we want to calculate $z = w^T x + b$. We can use *np.dot(w, x) + b* in Python rather than a *for loop* to decrease our run time by a significant amount. The takeaway from this section is that in deep learning, we want to avoid for loops and use vectorized code (such as NumPy methods) to save time when using large data sets.

Recall that we had defined a matrix, X = $\begin{bmatrix} | & | & ... & | \\ x^1 & x^2 & ... & x^m \\ | & | & ... & | \end{bmatrix}$, that contains the training data.

In order to vectorize finding $z^i = w^T x^i + b$ for m training examples, we can instead create a vector of these z values, denoted by Z.

$$Z \ = \ [z^1, \ z^2, ..., \ z^m] = w^T X + [b, \ b, \ ..., \ b] = [w^t x^1 + b, \ w^t x^2 + b, \ ..., \ w^t x^m + b]$$

We will also want to vectorize our sigmoid function, which we will see in our programming assignment where we find A = $[a^1, a^2, ..., a^m] = \sigma(z)$. These are the forward propogations.

Next we will want to vectorize the remaining steps in order to speed up our code. Lets begin with $dz^i = a^i - y^i$. Instead of looping through each training example, we can use vectors we already created, A = $[a^1, a^2, ..., a^m]$ and Y = $[y^1, y^2, ..., y^m]$. We can define:

$$dZ = A - Y = [a^1 - y^1, a^2 - y^2, ..., a^m - y^m] = [dz^1, dz^2, ..., dz^m]$$

Now we want to vectorize dw for all training examples. We know that $dw^i += x^i_m * dz^i$ must be updated fo each example and then divided by the total number of examples (m), so we define this as:

$$dw = \frac{1}{m} X \, dz^T = \frac{1}{m} \begin{bmatrix} | & | & ... & | \\ x^1 & x^2 & ... & x^m \\ | & | & ... & | \end{bmatrix} \begin{bmatrix} dz^1 \\ | \\ dz^m \end{bmatrix} = \frac{1}{m} [x^1 dz^1 + ... + x^m dz^m]$$

Finally, we see db is the sum of dz^i divided by the total number of examples (m), we can write this as:

$$db \ = \ \frac{1}{m} \sum_{i=1}^{m} dz^i = np.sum(dZ)$$

Note that the below steps are only one step of gradient descent, you would still need a for loop to perform multiple steps. The new vectorized gradient descent can now be written as:

$$Z = w^T X + b$$

$$A = \sigma(z)$$

$$dZ = A - Y$$

$$dw = \frac{1}{m} X \, dz^t$$

$$db = \frac{1}{m} \ np.sum(dZ)$$

$$w = w - \alpha dw$$

$$b = b - \alpha db$$

Some quick notes on **broadcasting** in Python:

- When computing a mathematical operation on an (m,n) matrix with either a (1,n) or (m,1) vector, Python with automatically manipulate it to convert it into an (m,n) matrix to match.
- When computing a mathematical operation on a row vector (m,1) with a real number, Python will copy the number m times in order to match the sizes of the two vectors.
- To simplify your code, don't use "rank 1" arrays (m,). Instead use either column vectors (m,1) or row vectors (1,m) to avoid any errors. You can use assert statements to ensure they are the correct dimensions and reshape() to change any dimensions needed.

Lets take an in depth look at the math behind the **Logistic Regression cost function**:

Remember that $\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$. We interpret $\hat{y} = P(y = 1 \mid x)$ such that:

If y=1 : P(y|x) = $\hat{y}$

If y=0 : P(y|x) = 1-$\hat{y}$

We know that P(y|x) = $\hat{y}^y * (1 - \hat{y})^{(1-y)}$ and by taking the log of this, we get:

$$log(p(y|x)) \; = \; log(\hat{y}^y * (1 - \hat{y})^{(1-y)})$$

$$log(p(y|x)) \; = \; ylog(\hat{y}) + (1 - y)log(1 - \hat{y})$$

We know that the above equation for log(p(y|x)) can be denoted as -$\mathcal{L}(\hat{y}, y)$, which is our cost function. This is only for one example, but we need to find this for m examples. We will use **maximum likelihood estimation** to find the parameters that maximize this equation:

$$log(p(m\ examples)) \; = \; log(\prod_{i=1}^{n} p(y^i|x^i))$$

$$log(p(m\ examples)) \; = \; \sum_{i=1}^{m} log(p(y^i|x^i))$$

$$log(p(m\ examples)) \; = \; -\sum_{i=1}^{m} \mathcal{L}(\hat{y}^i, y^i)$$

This justifies out cost function, and because now we want to minimize the cost we drop the negative sign and to make sure our quantities are scaled, we add the $\frac{1}{m}$. Note that minimizing the loss below corresponds to maximizing log(p(y|x)).

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^i, y^i) = \frac{1}{m} \sum_{i=1}^{m} [(y^i \, log(\hat{y}^i) \; + \; (1 - y^i) \, log(1 - \hat{y}^i))]$$

1.1.5 Programming Assignment

Problem Statement: You are given a dataset ("data.h5") containing:

- A training set of m_train images labeled as cat (y=1) or non-cat (y=0).
- A test set of m_test images labeled as cat or non-cat.
- Each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB). Thus, each image is square (height = num_px) and (width = num_px).

You will build a simple image-recognition algorithm that can correctly classify pictures as cat or non-cat. We added "_orig" at the end of image datasets (train and test) because we are going to preprocess them. After preprocessing, we will end up with train_set_x and test_set_x.

```
# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()
```

Lets find the dimensions of our data. Remember that train_set_x_orig is a numpy-array of shape (m_train, num_px, num_px, 3).

```
m_train = train_set_x_orig.shape[0] # 209 examples
m_test = test_set_x_orig.shape[0] # 50 examples
num_px = train_set_x_orig.shape[1] # 64
```

Note that each image is of size (64, 64, 3), the training set has shape (209, 64, 64, 3) with corresponding labels (1, 209), and the test set has shape (50, 64, 64, 3) with corresponding labels (1,50).

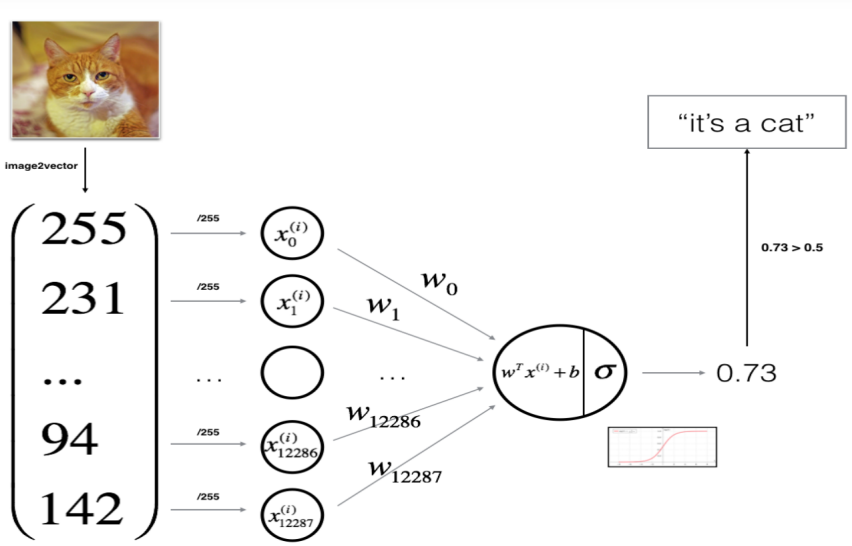
We now want to reshape the training and test data sets so that images of size (num_px, num_px, 3) are **flattened** into single vectors of shape (num_px*num_px*3, 1). To flatten a matrix X of shape (a,b,c,d) to a matrix X_flatten of shape (b*c*d, a) we use: X_flatten = X.reshape(X.shape[0], -1).T

```
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T

train_set_x_flatten.shape # (12288, 209)
test_set_x_flatten.shape # (1288, 50)
```

Now we will **standardize** our dataset. One common practice is to substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (since the range for the vector values of an image are [0, 255]).

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```



Above is a depication of how our Logistic Regression Neural Network will operate (refer to the previous section for an explanation of the mathematical expressions). We will carry out the following steps:

- Initialize the parameters of the model.
- Learn the parameters for the model by minimizing the cost.
- Use the learned parameters to make predictions (on the test set).
- Analyse the results and conclude.

Lets begin building the parts of our algorithm. There are 3 main steps for **building a Neural Network**:

1. Define the model structure (such as number of input features).
2. Initialize the model's parameters.
3. Loop:
   - Calculate current loss (forward propagation).
   - Calculate current gradient (backward propagation).
   - Update parameters (gradient descent).

```python
def sigmoid(z):
    # Compute sigmoid of z = w.T * x + b
    s = 1 / (1+np.exp(-z))
    return s

def initialize_with_zeros(dim):
    # Creates a vector of zeros of shape (dim, 1) for w and initializes b=0.
    w = np.zeros((dim,1))
    b = 0
    return w, b
```

Now that your parameters are initialized, you can do the **forward/backward propagation** steps for learning the parameters. The steps for forward propagation are:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^1, a^2, ..., a^{m-1}, a^m)$
- You calculate the cost function: $J = -\frac{1}{m}\sum_{i=1}^{m} y^i \log(a^i) + (1 - y^i)\log(1 - a^i)$

The steps for back propagation are:

- Compute $\frac{\partial J}{\partial w} = \frac{1}{m} X(A - Y)^T$
- Compute $\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^i - y^i)$

```python
def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation explained above
    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, m)

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b
    """

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T, X) + b) # compute activation
    cost = -(1/m)*np.sum(Y*np.log(A) + (1-Y)*np.log(1-A)) # compute cost

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = (1/m) * np.dot(X, (A-Y).T)
    db = (1/m) * np.sum(A-Y)

    cost = np.squeeze(cost)
    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

Now that we have intialized our parameters and can compute a cost function and its gradient, we can create an **optimize function** to update the parameters using gradient descent. The goal is to learn w and b by minimizing the cost function J . For a parameter θ , the update rule is $\theta = \theta - \alpha d\theta$, where α is the learning rate.

We can also create a **predict** function with our learned parameters to make predictions for a dataset X. We will calculate $\hat{Y}$ = A, and then convert entries to 0 or 1 based on probabilities.

```
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
  """
  Arguments:
  w -- weights, a numpy array of size (num_px * num_px * 3, 1)
  b -- bias, a scalar
  X -- data of shape (num_px * num_px * 3, number of examples)
  Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, m)
  num_iterations -- number of iterations of the optimization loop
  learning_rate -- learning rate of the gradient descent update rule
  print_cost -- True to print the loss every 100 steps

  Returns:
  params - dictionary containing the weights w and bias b
  grads - the gradients of the weights and bias with respect to the cost function
  costs - list of all the costs computed during the optimization (graphing)
  """
  costs = []

  for i in range(num_iterations):
    grads, cost = propagate(w, b, X, Y)

    dw = grads["dw"] # Retrieve derivatives from grads
    db = grads["db"] # Retrieve derivatives from grads

    w = w - learning_rate * dw # update rule
    b = b - learning_rate * db # update rule

    if i % 100 == 0: # Record the costs
      costs.append(cost)

    if print_cost and i % 100 == 0: # Print the cost every 100 training iterations
      print ("Cost after iteration %i: %f" %(i, cost))

  params = {"w": w, "b": b}
  grads = {"dw": dw, "db": db}
  return params, grads, costs

def predict(w, b, X):
'''
Predict whether the label is 0 or 1 using learned  parameters (w, b)

Arguments:
w -- weights, a numpy array of size (num_px * num_px * 3, 1)
b -- bias, a scalar
X -- data of size (num_px * num_px * 3, number of examples)

Returns a numpy array (vector) containing all predictions (0/1) for the examples in X
'''
m = X.shape[1]
Y_prediction = np.zeros((1,m))
w = w.reshape(X.shape[0], 1)

# Compute "A" predicting the probabilities of a cat being present in the picture
A = sigmoid(np.dot(w.T, X) + b)

for i in range(A.shape[1]): # Convert prob. A[0,i] to actual predictions p[0,i]
  Y_prediction[0,i] = A[0,i] > 0.5

assert(Y_prediction.shape == (1, m))

return Y_prediction
```

The final step is to **merge all functions into a model** by putting together the previous parts in the correct order.

```
def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
          learning_rate = 0.5, print_cost = False):
    """
    Arguments:
    X_train -- training set represented by a np array (num_px * num_px * 3, m_train)
    Y_train -- training labels represented by a np array (1, m_train)
    X_test -- test set represented by a np array (num_px * num_px * 3, m_test)
    Y_test -- test labels represented by a np array (1, m_test)
    num_iterations -- hyperparameter used to optimize the parameters
    learning_rate -- hyperparameter used in the update rule of optimize()
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """
    w, b = initialize_with_zeros(X_train.shape[0]) # initialize parameters with zeros

    # Gradient descent
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations,
                                        learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    Y_prediction_test = predict(w, b, X_test) # Predict test set examples
    Y_prediction_train = predict(w, b, X_train) # Predict test set examples

    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train -
                                                             Y_train)) * 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test -
                                                            Y_test)) * 100))
    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d

d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000,
          learning_rate = 0.005, print_cost = True)
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

From our model, we can see that the training accuracy is 99% while the test accuracy is 70%, which is a cause of overfitting. We can also see that the cost is decreasing per onehundred iterations (meaning the parameters are being leaned). We can continue to idecrease the cost by increasing the number of iterations, but this will also cause more offerfitting to occur. Given the small dataset and the fact that Logistic Regression is a linear classifier, overall it is not a bad simple model. In the future, we will learn to avoid overfitting and increase the test accuracy.

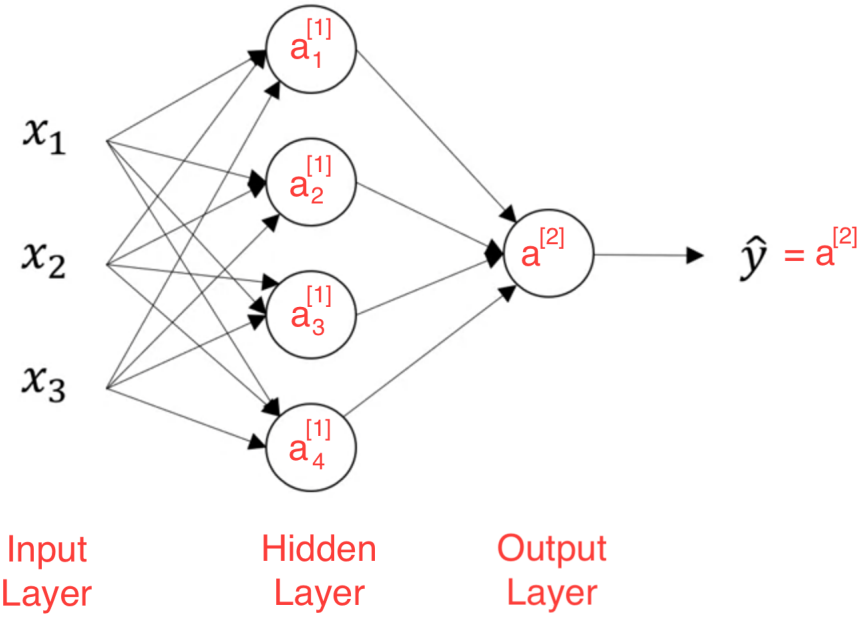
1.2 Shallow Neural Network

1.2.1 Overview and Representation

Previously, we only had a single sigmoid function in our Logistic Regression model. Now, we will stack multiple sigmoids ontop of one another, followed by then feeding these into another sigmoid to create a Nerual Network. Some new notation we are introducing:

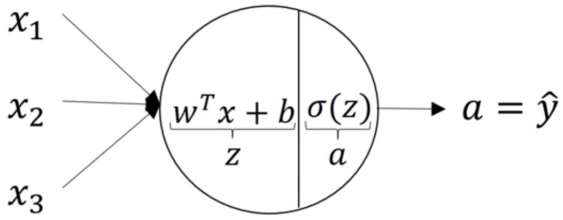
- [#] will refer to quanities associated with a given layer.
- (i) will refer to the ith training example (similar to the previous section).
- a$^{[0]}$ will be the activation layer (same as our vector x that has the input features).
- a$^{[1]}$ will be the values for our hidden layer.
- a$^{[2]}$ will be the output layer values (our $\hat{y}$).
- a$_i^{[l]}$ will be $[l]$ = layer, i = node in the layer.

We will be focusing on a **Two Layer Neural Network**, which has an input layer, one hidden layer, and an output layer. We don't count the input layer as an "official" layer. The *hidden layer* will have parameters $w^{[1]}$ and $b^{[1]}$, while the output layer has parameters $w^{[2]}$ and $b^{[2]}$ associated with it.



1.2.2 Computing a Neural Network Output

Each node in our hidden layer will take all of the input values from x, compute z, and input this value into an activation function (sigmoid). Since each node has to compute z, we will vectorize this process by using matrix multiplication in python. This gives us the following equation to find our vector $z^{[1]}$ and our activation vector $a^{[1]}$ for the hidden layer.



$$z^{[1]} = \begin{bmatrix} - w_1^{[1]T} - \\ - w_2^{[1]T} - \\ - w_3^{[1]T} - \\ - w_4^{[1]T} - \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}, \quad \text{also let } a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

Note: Let the matrix with the w values be denoted as $W^{[1]}$ and the vector holding b values be $b^{[1]}$.

For the **hidden layer**, this gives us the general formulas (remember x = $a^{[0]}$):

$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$ with shapes: $z^{[1]} = (4,1),\ W^{[1]} = (4,3),\ a^{[0]} = (3,1),\ b^{[1]} = (4,1)$

$a^{[1]} = \sigma(z^{[1]})$ with shapes: $a^{[1]} = (4,1),\ z^{[1]} = (4,1)$

For the **output layer**, this gives us the following formulas (ouput $a^{[1]}$ used as input "x"):

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ with shapes: $z^{[2]} = (1,1),\ W^{[2]} = (1,4),\ a^{[1]} = (4,1),\ b^{[2]} = (1,1)$

$a^{[2]} = \sigma(z^{[2]})$ with shapes: $a^{[2]} = (1,1),\ z^{[2]} = (1,1)$

1.2.3 Vectorizing Across Multiple Examples

When we want to compute predictions for all of our training examples (not just a single example like we did above), we need to vectorize a method in order to compute all of these at once. Some new notation we will use:

- ex: $a^{[2](i)}$ refers to the layer 2 value for the i^{th} training example.

We will define the following matrices to work with n_x training examples where X = m training examples, $Z^{[1]}$ = is all of the $z^{[1]}$ values for m training example, and $A^{[1]}$ = all of our $a^{[1]}$ values for m training examples. Note that the format is the same for $Z^{[2]}$ and $A^{[2]}$ with their corresponding values.

$$X = \begin{bmatrix} | & | & \dots & | \\ x^1 & x^2 & \dots & x^m \\ | & | & \dots & | \end{bmatrix} \quad Z^{[1]} = \begin{bmatrix} | & | & \dots & | \\ z^{[1](1)} & z^{[1](2)} & \dots & z^{[1](m)} \\ | & | & \dots & | \end{bmatrix} \quad A^{[1]} = \begin{bmatrix} | & | & \dots & | \\ a^{[1](1)} & a^{[1](2)} & \dots & a^{[1](m)} \\ | & | & \dots & | \end{bmatrix}$$

This gives us the following **vectorized formulas** to find all predicted values for m examples:

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

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

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

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

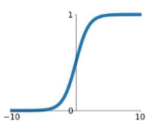
1.2.4 Activation Functions

Previously, we have used the sigmoid function as our activation function. However, there are much better functions that we can use instead, denoted by a general symbol g. For a Two Layer Neural Network, we can denote the activation function for the **hidden layer** as $g^{[1]}(z^{[1]})$ and the **output layer** as $g^{[2]}(z^{[2]})$.

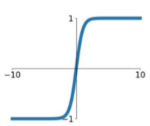
One exception is when you are doing binary classification to use a sigmoid function for the output layer. This is because a a sigmoid function will output a value between 0 and 1, which works perfectly since our true labels (y) will either be a 0 or 1.

Another option for an activation function is tanh(z). This is often much more useful than the sigmoid function, and will output a value in the range [-1,1]. A downfall to both the sigmoid and tanh function are that when z is very large or small, the gradient is near 0 and can cause gradient descent to slow. The most commonly used activation function is the ReLU function (or the Leaky ReLU function to avoid having a 0 gradient for negative numbers).

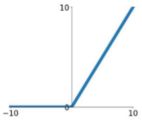
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



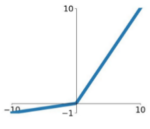
tanh
$\tanh(x)$



ReLU
$\max(0, x)$



Leaky ReLU
$\max(0.1x, x)$



1.2.5 Gradient Descent for Neural Networks

A Neural Network with one hidden layer will have the following:

- Parameters: $W^{[1]}$ with shape $(n^{[1]}, n^{[0]})$.
  $b^{[1]}$ with shape $(n^{[1]}, 1)$.
  $W^{[2]}$ with shape $(n^{[2]}, n^{[1]})$.
  $b^{[2]}$ with shape $(n^{[2]}, 1)$.
- $n^{[0]}$ input features (known as n_x), $n^{[1]}$ hidden layer units, and $n^{[2]}$ output units.
- Cost Function: J($W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$) $= \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$
- Gradient descent (repeat the following steps):
  1) Compute predicts $(\hat{y}^{(i)}, ..., \hat{y}^{(m)})$
  2) Compute $dW^{[1]} = \frac{dJ}{dw^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ... and similary for $dW^{[2]}$ and $db^{[2]}$
  3) Compute $W^{[1]} = W^{[1]} - \alpha\, dw^{[1]}$
  4) Compute $b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$
  5) Compute $W^{[2]} = W^{[2]} - \alpha\, dw^{[2]}$
  6) Compute $b^{[2]} = b^{[2]} - \alpha\, db^{[2]}$

Recall the formulas for **forward propagation** (where g is the generalized activation function):

$Z^{[1]} = W^{[1]} X + b^{[1]}$
$A^{[1]} = g^{[1]}(Z^{[1]})$
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
$A^{[2]} = g^{[2]}(Z^{[2]})$

This means that we can perform **back propagation** (gradient descent) with the following steps:

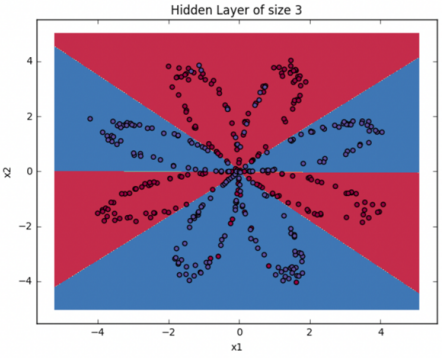
$dz^{[2]} = A^{[2]} - Y$
$dW^{[2]} = \frac{1}{m}\, dz^{[2]}\, A^{[1]T}$
$db^{[2]} = \frac{1}{m}\, np.sum(dz^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T}\, dz^{[2]} * g^{[1]'}(Z^{[1]})$, which is an element-wise product of two $(n^{[1]}, m)$ matrices.
$dW^{[1]} = \frac{1}{m}\, dz^{[1]}\, X^T$
$db^{[1]} = \frac{1}{m}\, np.sum(dz^{[1]}, axis = 1, keepdims = True)$, which is an $(n^{[1]}, 1$ vector).

For a Neural Network, we want to **initialize the weights** to random values instead of zeros (initializing to zero will cause all of the calculations to be symmetric). To randomly initialize our weights:

- Set $W^{[1]}$ = np.random.randn((2,2)) * 0.01 to create small Gaussian random variables.
- Initialize $b^{[1]}$ = np.zero((2,1)) because b does not have the symmetry problem that w can have.
- Similarly, we can do the same for $W^{[2]}$ and $b^{[2]}$.

1.2.6 Programming Assignment

For this, you will generate red and blue points to form a flower. You will then fit a neural network to correctly classify the points. You will try different layers and see the results.



We will learn:

- 2-class classification NN with one hidden layer.
- Use units with a non-linear activation function (ex: tanh).
- Compute the cross entropy loss.
- Implement forward and backward propagation.

FIrst, lets **import the packages and dataset** that we will be working with. Also, it will be helpful to visualize the data using matplotlib (notice how it is a flower with two different colored points). For our data, the red corresponds to y=0 and the blue corresponds to y=1. For our data, we have:

- A numpy-array (matrix) X that contains your features (x1, x2)
- A numpy-array (vector) Y that contains your labels (red:0, blue:1). sc

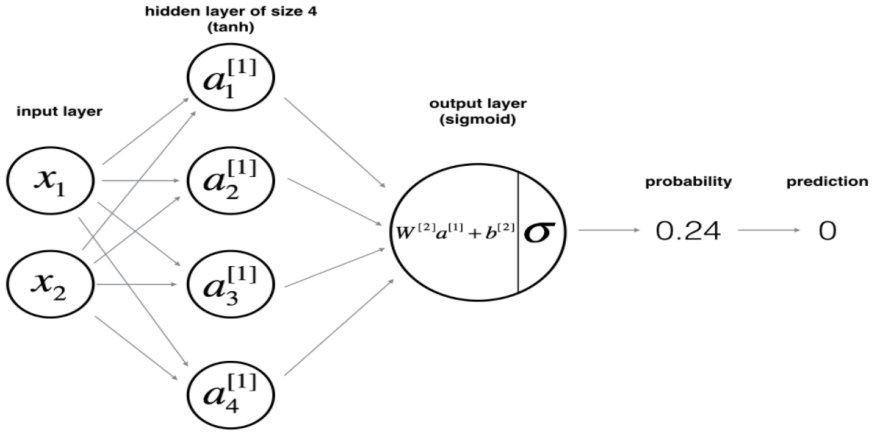
```
import numpy as np
import matplotlib.pyplot as plt
from testCases_v2 import * # test examples to test correctness of functions
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset,
                         load_extra_datasets

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

X, Y = load_planar_dataset()

shape_X = X.shape # (2, 400)
shape_Y = Y.shape # (1, 400)
m = X.shape[1] # 400
```

We are going to build a Neural Network model with one hidden layer. Lets take a look at the **setup** for our model and the **mathematical equations** that correspond to them. Note that we will use the *tanh* activation function for the hidden layer, and the *sigmoid* activation function for the output layer since it is binary classification.



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

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \tanh(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$

$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)})$$

$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

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

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

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)

You often build helper functions to compute steps 1-3 and then merge them into one function we call 'nn_model()'. Once you've built 'nn_model()' and learnt the right parameters, you can make predictions on new data.

```
# Step 1: Define the Neural Network structure
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)
    """
    n_x = X.shape[0] # size of input layer
    n_h = 4 # size of the hidden layer
    n_y = Y.shape[0] # size of output layer

    return (n_x, n_h, n_y)

# Step 2: Initialize the model's parameters (random init)
def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(2) # match example output (but initialization is random).

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

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

# Step 3 (part 1): Implement forward propagation
def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- dict containing output of initialization function
    """
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

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

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

Now that you have computed $A^{[2]}$, which contains $a^{[2](i)}$ for every example, you can **compute the cost** function as follows:

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

There's many ways to implement cross-entropy loss. In Python, we implement $-\sum_{i=0}^{m} y^{(i)}\log(a^{[2](i)})$ as:

logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)

Note that if you use 'np.multiply' followed by 'np.sum' the end result will be a type 'float', whereas if you use 'np.dot', the result will be a 2D numpy array. We can use 'np.squeeze()' to remove redundant dimensions (in the case of single float, this will be reduced to a zero-dimension array). We can cast the array as a type 'float' using 'float()'.

```
# Step 3 (part 2): Compute the cost
def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation above

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

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

    cost = float(np.squeeze(cost))  # makes sure cost is the dimension we expect
    return cost
```

Now that we have the cache computed from forward propagation, we can now implement **backward propagation**. Remeber that we will use the six vectorized equations we previously found, which are:

$$dz^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m}\,dz^{[2]}\,A^{[1]T}$$

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

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

$$dW^{[1]} = \frac{1}{m}\,dz^{[1]}\,X^{T}$$

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

Quick Note: to compute dZ1 you'll need to compute $g^{[1]'}(Z^{[1]})$. Since $g^{[1]}()$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]'}(z) = 1-a^2$. So you can compute $g^{[1]'}(Z^{[1]})$ using '(1 - np.power(A1, 2))' in Python.

```python
# Step 3 (part 3): Backward propagation to get gradients
def backward_propagation(parameters, cache, X, Y):
    """
    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)
    """
    m = X.shape[1]

    W1 = parameters['W1']
    W2 = parameters['W2']
    A1 = cache['A1']
    A2 = cache['A2']

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

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

Now that we have the gradients, we can implement the **update rule**. Remeber that the general rule is $\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$ where α is the learning rate and θ represents a parameter.

```python
# Step 3 (part 4): Update parameters using gradient descent
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
    """
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']

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

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

Now that we have completed each step in the general methodology to build a Neural Network, we can create a function to put together each of these helper functions. We call this our **Neural Network Model** function.

```python
def nn_model(X, Y, n_h, num_iterations = 10000, 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. 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
    parameters = initialize_parameters(n_x, n_h, n_y)

    # Loop (gradient descent)
    for i in range(0, num_iterations):
        # Forward propagation. Outputs: "A2, cache".
        A2, cache = forward_propagation(X, parameters)

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

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

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

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

    return parameters
```

Now that we can build a complete model, we can make predictions using forward propagation:

$$y_{prediction} = \text{activation} > 0.5 = \begin{cases} 1 & \text{if } activation > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

```python
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)
    """
    # Computes probabilities using forward propagation (threshold of 0.5)
    A2, cache = forward_propagation(X, parameters)
    predictions = (A2 > 0.5) # vector of predict 0/1 values

    return predictions
```

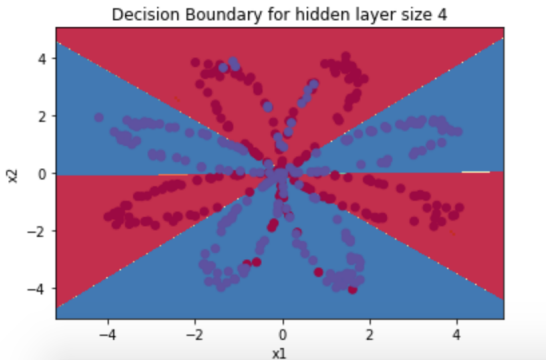
Now that we have a way to create a model and make predictions, we can use this on our planar dataset. We will use a hidden layer of size 4 for our Neural Network.

```python
# 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))

# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))
                              /float(Y.size)*100) + '%') # 90%
```

```
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: 0.219471
Cost after iteration 9000: 0.218612
<matplotlib.text.Text at 0x7f31307ead68>
```

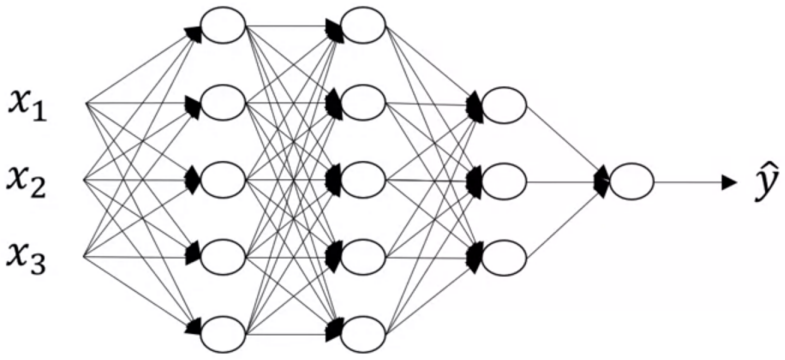


We can see that per 1000 iterations, the cost continued to decrease. The decision boundries were quite accurate as well. Accuracy is really high (90%) compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision boundaries, unlike logistic regression.

If we were to run the Neural Network with many different hidden layer sizes. We can see that around an n_h=5 would give us the highest accuracy (around 91%). The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data. We will also learn later about regularization, which lets you use very large models (such as n_h = 50) without much overfitting.

1.3 Deep L-Layer Neural Network

Previously, we have been using a shallow Neural Network with one hidden layer. Now we will focus on a **deep Neural Network** with multiple hidden layers, with the below exampling being a 4 layer Neural Network with 3 hidden layers. Lets take a look at a diagram and some new notation we will use:



L = number of layers in the network.
$n^{[l]}$ = number of units (neurons) in layer l.
$a^{[l]}$ = activations in layer l, where $a^{[l]} = g^{[l]}(z^{[}l])$
$W^{[l]}$ = the weights for corresponding $z^{[l]}$.
$b^{[l]}$ = the corresponding b values for layer l.
$a^{[0]}$ = x (the input features for our model).
$a^{[L]}$ = the predicted outputs for our model ($\hat{y}$).

We will begin with computing forward propagation for a **single training example** (x). Note that l denotes a given layer, a represents the input from the previous layer, and g is our activation function:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

Now lets look at the **vectorized** formulas for computing forward propagation. Remember that the capital letter denotes a matrix that holds all of the values for m training examples:

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

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

Note that for a deep Neural Network, we will have to use a for loop to **iterate** over $l = 1, ...,$ L. Using a for loop for propagation is the only time we are allowed to since there is no other way.

It is very important that our **matrix dimensions** are correct in order for our outputs to line up with one another. The general shape for each of the variables is as follows:

For a single training example:	For m training examples:
$z^{[l]} = (n^{[l]}, 1)$	$Z^{[l]} = (n^{[l]}, m)$
$W^{[l]} = (n^{[l]}, n^{[l-1]})$	$W^{[l]} = (n^{[l]}, n^{[l-1]})$
$a^{[l]} = (n^{[l-1]}, 1)$	$A^{[l]} = (n^{[l-1]}, m)$
$b^{[l]} = (n^{[l]}, 1)$	$b^{[l]} = (n^{[l]}, 1)$

1.3.1 Forward and Backward Propagation

We will begin with **forward propagation** for a layer l:

Input: $a^{[l-1]}$

Output: $a^{[l]}$, cache $(z^{[l]})$

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

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

Next we want to compute **backward propagation** for a layer l:

Input: $da^{[l]}$

Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$$

$$dW^{[l]} = \frac{1}{m}\, dZ^{[l]}\, A^{[l-1]T}$$

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

$$dA^{[l-1]} = W^{[l]T}\, dZ^{[l]}$$

We initialize forward propagation with x (our training examples). But for backwards propagation, we intialize with $da^{[l]} = (-\frac{y}{a} + \frac{1-y}{1-a})$ for a single layer l. But the vectorized version we intialize with:

$$dA^{[l]} = \left[(-\tfrac{y^{(1)}}{a^{(1)}} + \tfrac{1-y^{(1)}}{1-a^{(1)}}), ..., (-\tfrac{y^{(m)}}{a^{(m)}} + \tfrac{1-y^{(m)}}{1-a^{(m)}})\right]$$

A quick note on **parametes vs. hyperparameters**:

The parameters for our model are W and b. The hyperparameters can be things such as learning rate (α), number of iterations, number of hidden layers (L), hidden units, activation function, etc. The hyperparameters help to control the final values of W and b.

Deep learning is a very empirical process. We may try an idea for a hyperparameter, implement it in our code, and observe the results of the experiment through iterating. It is a trial and error process to find the best hyperparameters for our models.

1.3.2 Programming Assignment