

Deep Learning in Python

Contents

1	Neural Networks and Deep Learning	1
1.1	Logistic Regression as a Neural Network	1
1.1.1	Introduction to Logistic Regression	1
1.1.2	Logistic Regression Cost Function	1
1.1.3	Gradient Descent	2
1.1.4	Vectorizing Logistic Regression	3
1.1.5	Programming Assignment	4
1.2	Shallow Neural Network	9
1.2.1	Overview and Representation	9
1.2.2	Computing a Neural Network Output	9
1.2.3	Vectorizing Across Multiple Examples	10
1.2.4	Activation Functions	10
1.2.5	Gradient Descent for Neural Networks	11
1.2.6	Programming Assignment	11
1.3	Deep L-Layer Neural Network	17
1.3.1	Forward and Backward Propagation	18
1.3.2	Programming Assignment (Building)	19
1.3.3	Programming Assignment 2 (Classifier)	26
2	Improving Deep Neural Networks	32
2.1	Regularizing & Optimizing Your Neural Network	32
2.1.1	Bias/Variance	32
2.1.2	Regularization	32
2.1.3	Dropout Regularization	33
2.1.4	Normalizing Inputs	34
2.1.5	Weight Initialization	34
2.1.6	Gradient Checking	35
2.1.7	Programming Assignment (Initialization)	36
2.1.8	Programming Assignment (Regularization)	39
2.1.9	Programming Assignment (Gradient Checking)	45
2.2	Optimization Algorithms	49
2.2.1	Mini-Batch Gradient Descent	49
2.2.2	Exponentially Weighted Averages	49
2.2.3	Gradient Descent with Momentum	50
2.2.4	RMSprop	50
2.2.5	Adam	50
2.2.6	Learning Rate Decay	51
2.2.7	Programming Assignment (Optimization)	51
2.3	Hyperparameter Tuning, Batch Normalization, and Frameworks	59
2.3.1	Tuning process	59
2.3.2	Batch Normalization	59
2.3.3	Multi-Class Classification (Softmax Regression)	60
2.3.4	Programming Frameworks (TensorFlow)	61
2.3.5	Programming Assignment (TensorFlow 1)	62

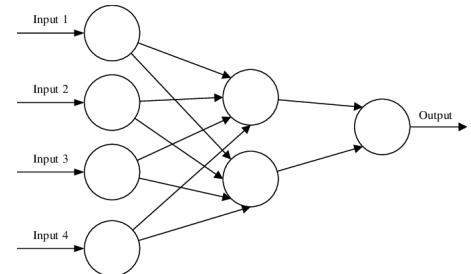
3 Structuring Machine Learning Projects	68
3.1 Machine Learning Strategy 1	68
3.1.1 Evaluation Metrics	68
3.1.2 Setting up the Dev and Test sets	68
3.1.3 Comparing to Human-Level Performance	69
3.2 Machine Learning Strategy 2	69
3.2.1 Error Analysis	69

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, \dots, 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: $\{(x^1, y^1), \dots, (x^m, y^m)\}$
- Matrix X :
$$\begin{bmatrix} | & | & \dots & | \\ x^1 & x^2 & \dots & x^m \\ | & | & \dots & | \end{bmatrix}$$
 where rows = n_x , columns = m . In Python, $X.shape = (n_x, m)$.
- Matrix Y :
$$\begin{bmatrix} y^1 & y^2 & \dots & y^m \end{bmatrix}$$
 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 our 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 $\{(x^1, y^1), \dots, (x^m, y^m)\}$ 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:

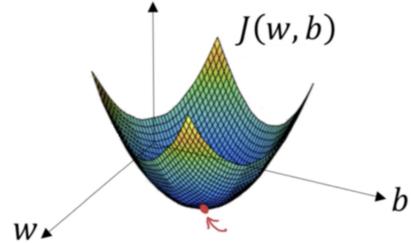
- If $y=1$: $\mathcal{L}(\hat{y}, y) = -y \log(\hat{y})$ and we want this to be as small as possible (\hat{y} large).
- If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y})$ and we want this to be large (\hat{y} small).

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.



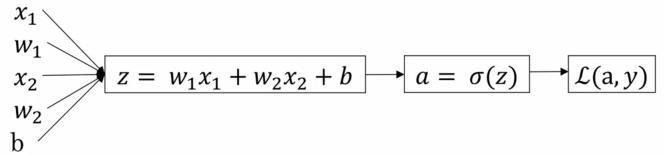
Gradient descent will repeatedly update the value of w and b with the formula:

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

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:

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



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:

$$\begin{aligned} z^i &= w^T x^i + b \\ z^i &= \sigma(z^i) \\ J+ &= -[(y^i \log(a^i) + (1 - y^i) \log(1 - a^i))] \\ dz^i &= a^i - y^i \\ dw_1+ &= x_1^i * dz^i \\ dw_2+ &= x_2^i * dz^i \\ db+ &= dz^i \end{aligned}$$

$$\begin{aligned} J /&= m \\ dw_1 /&= m \\ dw_2 /&= m \\ db /&= m \\ w_1 &= w_1 - \alpha * dw_1 \\ w_2 &= w_2 - \alpha * dw_2 \\ b &= b - \alpha * db \end{aligned}$$

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

$$\begin{aligned} J /&= m \\ dw_1 /&= m \\ dw_2 /&= m \\ db /&= m \\ w_1 &= w_1 - \alpha * dw_1 \\ w_2 &= w_2 - \alpha * dw_2 \\ b &= b - \alpha * db \end{aligned}$$

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, $X = \begin{bmatrix} | & | & \dots & | \\ x^1 & x^2 & \dots & x^m \\ | & | & \dots & | \end{bmatrix}$, 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 .

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

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

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, $A = [a^1, a^2, \dots, a^m]$ and $Y = [y^1, y^2, \dots, y^m]$. We can define:

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

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

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

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

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

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

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:

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

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

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

This justifies our 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 $(\text{num_px}, \text{num_px}, 3)$ 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.

```
1 # Loading the data (cat/non-cat)
2 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).

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

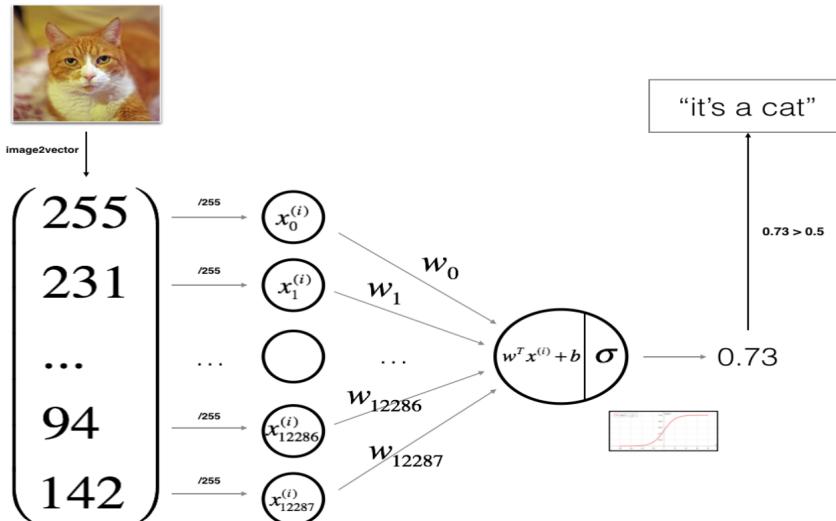
Note that each image is of size (64, 64, 3), the training set has shape (209, 64, 64, 3) with corresponding labels (1, 209), and the test set has shape (50, 64, 64, 3) with corresponding labels (1,50).

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`

```
1 train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
2 test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T
3
4 train_set_x_flatten.shape # (12288, 209)
5 test_set_x_flatten.shape # (1288, 50)
```

Now we will **standardize** our dataset. One common practice is to subtract 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]).

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



Above is a depiction 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).
- Analyze 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).

```

1 def sigmoid(z):
2     # Compute sigmoid of z = w.T * x + b
3     s = 1 / (1+np.exp(-z))
4     return s
5
6 def initialize_with_zeros(dim):
7     # Creates a vector of zeros of shape (dim, 1) for w and initializes b=0.
8     w = np.zeros((dim,1))
9     b = 0
10    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 $A = \sigma(w^T X + b) = (a^1, a^2, \dots, a^{m-1}, a^m)$
- 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 $\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^m (a^i - y^i)$

```

1 def propagate(w, b, X, Y):
2     """
3         Implement the cost function and its gradient for the propagation explained above
4         Arguments:
5             w -- weights, a numpy array of size (num_px * num_px * 3, 1)
6             b -- bias, a scalar
7             X -- data of size (num_px * num_px * 3, number of examples)
8             Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, m)
9
10            Return:
11            cost -- negative log-likelihood cost for logistic regression
12            dw -- gradient of the loss with respect to w, thus same shape as w
13            db -- gradient of the loss with respect to b, thus same shape as b
14            """
15
16            m = X.shape[1]
17
18            # FORWARD PROPAGATION (FROM X TO COST)
19            A = sigmoid(np.dot(w.T, X) + b) # compute activation
20            cost = -(1/m)*np.sum(Y*np.log(A) + (1-Y)*np.log(1-A)) # compute cost
21
22            # BACKWARD PROPAGATION (TO FIND GRAD)
23            dw = (1/m) * np.dot(X, (A-Y).T)
24            db = (1/m) * np.sum(A-Y)
25
26            cost = np.squeeze(cost)
27            grads = {"dw": dw,
28                      "db": db}
29
30            return grads, cost

```

Now that we have initialized 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.

```

1 def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
2 """
3     Arguments:
4         w -- weights, a numpy array of size (num_px * num_px * 3, 1)
5         b -- bias, a scalar
6         X -- data of shape (num_px * num_px * 3, number of examples)
7         Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, m)
8         num_iterations -- number of iterations of the optimization loop
9         learning_rate -- learning rate of the gradient descent update rule
10        print_cost -- True to print the loss every 100 steps
11
12    Returns:
13        params - dictionary containing the weights w and bias b
14        grads - the gradients of the weights and bias with respect to the cost function
15        costs - list of all the costs computed during the optimization (graphing)
16    """
17    costs = []
18
19    for i in range(num_iterations):
20        grads, cost = propagate(w, b, X, Y)
21
22        dw = grads["dw"] # Retrieve derivatives from grads
23        db = grads["db"] # Retrieve derivatives from grads
24
25        w = w - learning_rate * dw # update rule
26        b = b - learning_rate * db # update rule
27
28        if i % 100 == 0: # Record the costs
29            costs.append(cost)
30
31        if print_cost and i % 100 == 0: # Print the cost every 100 training iterations
32            print ("Cost after iteration %i: %f" %(i, cost))
33
34    params = {"w": w, "b": b}
35    grads = {"dw": dw, "db": db}
36    return params, grads, costs
37
38 def predict(w, b, X):
39     """
40     Predict whether the label is 0 or 1 using learned parameters (w, b)
41
42     Arguments:
43         w -- weights, a numpy array of size (num_px * num_px * 3, 1)
44         b -- bias, a scalar
45         X -- data of size (num_px * num_px * 3, number of examples)
46
47     Returns a numpy array (vector) containing all predictions (0/1) for the examples in X
48     """
49     m = X.shape[1]
50     Y_prediction = np.zeros((1,m))
51     w = w.reshape(X.shape[0], 1)
52
53     # Compute "A" predicting the probabilities of a cat being present in the picture
54     A = sigmoid(np.dot(w.T, X) + b)
55
56     for i in range(A.shape[1]): # Convert prob. A[0,i] to actual predictions p[0,i]
57         Y_prediction[0,i] = A[0,i] > 0.5
58
59     assert(Y_prediction.shape == (1, m))
60
61     return Y_prediction

```

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

```

1  def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
2      learning_rate = 0.5, print_cost = False):
3      """
4          Arguments:
5              X_train -- training set represented by a np array (num_px * num_px * 3, m_train)
6              Y_train -- training labels represented by a np array (1, m_train)
7              X_test -- test set represented by a np array (num_px * num_px * 3, m_test)
8              Y_test -- test labels represented by a np array (1, m_test)
9              num_iterations -- hyperparameter used to optimize the parameters
10             learning_rate -- hyperparameter used in the update rule of optimize()
11             print_cost -- Set to true to print the cost every 100 iterations
12
13     Returns:
14     d -- dictionary containing information about the model.
15     """
16     w, b = initialize_with_zeros(X_train.shape[0]) # initialize parameters with zeros
17
18     # Gradient descent
19     parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations,
20                                         learning_rate, print_cost)
21
22     # Retrieve parameters w and b from dictionary "parameters"
23     w = parameters["w"]
24     b = parameters["b"]
25
26     Y_prediction_test = predict(w, b, X_test) # Predict test set examples
27     Y_prediction_train = predict(w, b, X_train) # Predict test set examples
28
29     # Print train/test Errors
30     print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train -
31                                                 Y_train)) * 100))
32     print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test -
33                                                 Y_test)) * 100))
34
35     d = {"costs": costs,
36           "Y_prediction_test": Y_prediction_test,
37           "Y_prediction_train" : Y_prediction_train,
38           "w" : w,
39           "b" : b,
40           "learning_rate" : learning_rate,
41           "num_iterations": num_iterations}
42
43     return d
44
45 d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000,
46           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 one hundred iterations (meaning the parameters are being leaned). We can continue to decrease the cost by increasing the number of iterations, but this will also cause more overfitting 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.

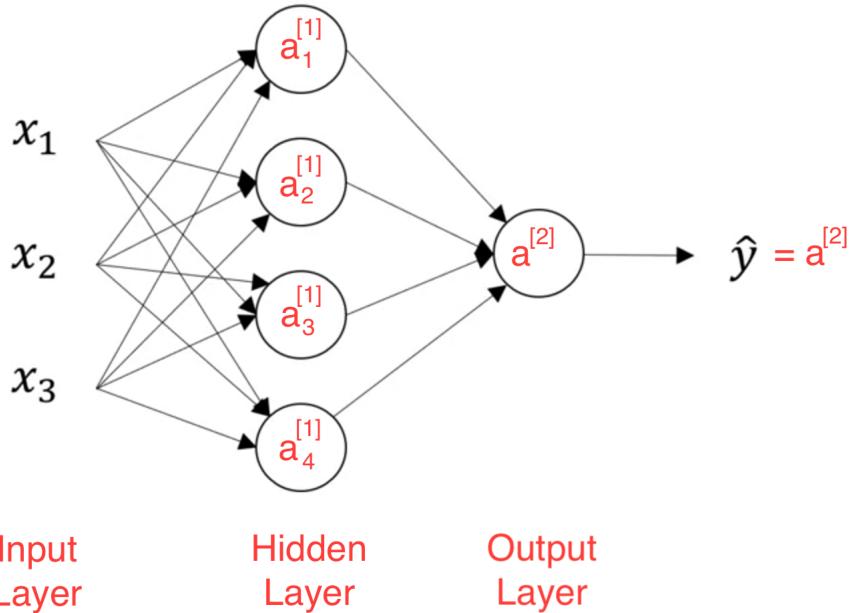
1.2 Shallow Neural Network

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 on top of one another, followed by then feeding these into another sigmoid to create a Neural Network. Some new notation we are introducing:

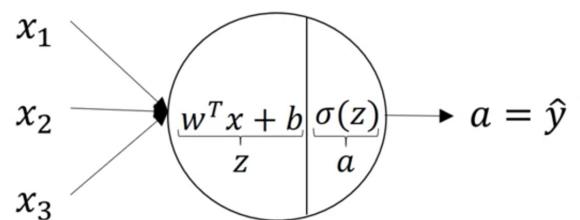
- $[\#]$ will refer to quantities associated with a given layer.
- (i) will refer to the i^{th} training example (similar to the previous section).
- $a^{[0]}$ will be the activation layer (same as our vector x that has the input features).
- $a^{[1]}$ will be the values for our hidden layer.
- $a^{[2]}$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

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.



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

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 $x = a^{[0]}$):

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

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

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

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

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

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.

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

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

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

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

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

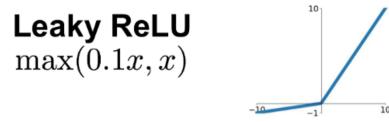
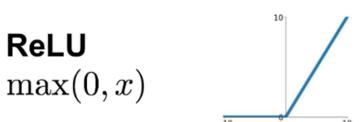
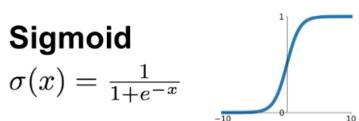
$$A^{[2]} = \sigma(Z^{[2]})$$

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 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).



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 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 $(\hat{y}^{(1)}, \dots, \hat{y}^{(m)})$
 - 2) Compute $dW^{[1]} = \frac{dJ}{dw^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ... and similarly 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:

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

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

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

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(Z^{[1]}), \text{ which is an element-wise product of two } (n^{[1]}, m) \text{ matrices.}$$

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

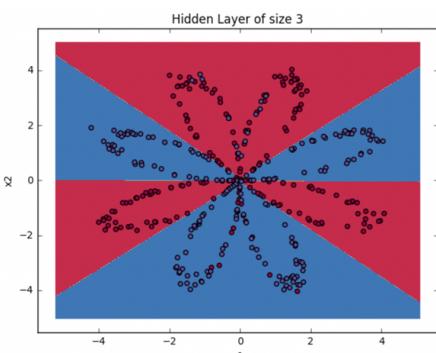
$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True), \text{ which is an } (n^{[1]}, 1 \text{ vector}).$$

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 $b^{[1]} = np.zeros((2,1))$ 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:

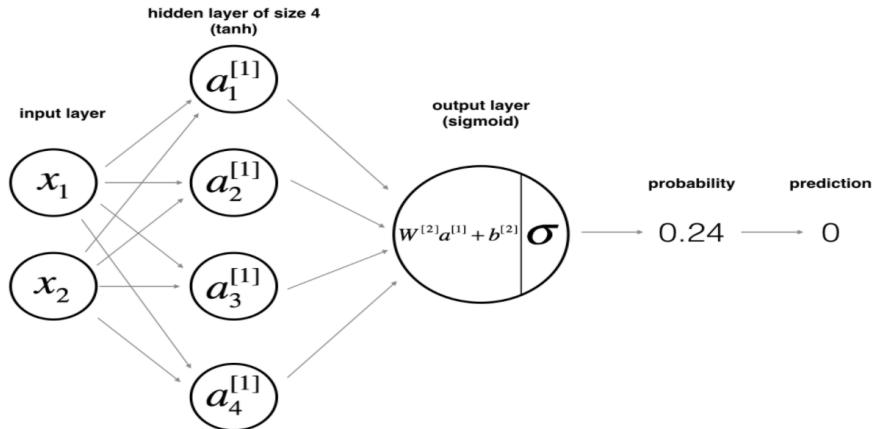
- A NumPy-array (matrix) X that contains your features (x_1, x_2)
- A NumPy-array (vector) Y that contains your labels (red:0, blue:1). sc

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from testCases_v2 import * # test examples to test correctness of functions
4 import sklearn
5 import sklearn.datasets
6 import sklearn.linear_model
7 from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset,
8                         load_extra_datasets
9
10 np.random.seed(1) # set a seed so that the results are consistent
11
12 X, Y = load_planar_dataset()
13
14 shape_X = X.shape # (2, 400)
15 shape_Y = Y.shape # (1, 400)
16 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 $x^{(i)}$:

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

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

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

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

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

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

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

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.

```
1  # Step 1: Define the Neural Network structure
2  def layer_sizes(X, Y):
3      """
4          Arguments:
5              X -- input dataset of shape (input size, number of examples)
6              Y -- labels of shape (output size, number of examples)
7      """
8
9      n_x = X.shape[0] # size of input layer
10     n_h = 4 # size of the hidden layer
11     n_y = Y.shape[0] # size of output layer
12
13     return (n_x, n_h, n_y)
14
15
16  # Step 2: Initialize the model's parameters (random init)
17  def initialize_parameters(n_x, n_h, n_y):
18      np.random.seed(2) # match example output (but initialization is random).
19
20      W1 = np.random.randn(n_h, n_x) * 0.01 # weight matrix of shape (n_h, n_x)
21      b1 = np.zeros((n_h, 1)) # bias vector of shape (n_h, 1)
22      W2 = np.random.randn(n_y, n_h) * 0.01 # weight matrix of shape (n_y, n_h)
23      b2 = np.zeros((n_y, 1)) # bias vector of shape (n_y, 1)
24
25      parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
26      return parameters
27
28
29  # Step 3 (part 1): Implement forward propagation
30  def forward_propagation(X, parameters):
31      """
32          Argument:
33              X -- input data of size (n_x, m)
34              parameters -- dict containing output of initialization function
35      """
36
37      W1 = parameters['W1']
38      b1 = parameters['b1']
39      W2 = parameters['W2']
40      b2 = parameters['b2']
41
42      # Implement Forward Propagation to calculate A2 (probabilities)
43      Z1 = np.dot(W1,X) + b1
44      A1 = np.tanh(Z1)
45      Z2 = np.dot(W2, A1) + b2
46      A2 = sigmoid(Z2) # output of second activation function
47
48      cache = {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2}
49      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:

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

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()’.

```
1 # Step 3 (part 2): Compute the cost
2 def compute_cost(A2, Y, parameters):
3     """
4         Computes the cross-entropy cost given in equation above
5
6     Arguments:
7     A2 -- The sigmoid output of the second activation, of shape (1, m)
8     Y -- "true" labels vector of shape (1, number of examples)
9     parameters -- python dictionary containing your parameters W1, b1, W2 and b2
10    """
11    m = Y.shape[1] # number of example
12
13    # Compute the cross-entropy cost
14    logprobs = np.multiply(np.log(A2), Y) + np.multiply((1 - Y), np.log(1-A2))
15    cost = -(1/m)*np.sum(logprobs)
16
17    cost = float(np.squeeze(cost)) # makes sure cost is the dimension we expect
18    return cost
```

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

$$\begin{aligned} dz^{[2]} &= A^{[2]} - Y \\ dW^{[2]} &= \frac{1}{m} dz^{[2]} A^{[1]T} \\ db^{[2]} &= \frac{1}{m} np.sum(dz^{[2]}, axis=1, keepdims=True) \\ dz^{[1]} &= W^{[2]T} dz^{[2]} * g^{[1]'}(Z^{[1]}) \\ dW^{[1]} &= \frac{1}{m} dz^{[1]} X^T \\ db^{[1]} &= \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True) \end{aligned}$$

Quick Note: to compute dZ_1 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.

```

1 # Step 3 (part 3): Backward propagation to get gradients
2 def backward_propagation(parameters, cache, X, Y):
3     """
4     Arguments:
5     parameters -- python dictionary containing our parameters
6     cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
7     X -- input data of shape (2, number of examples)
8     Y -- "true" labels vector of shape (1, number of examples)
9     """
10    m = X.shape[1]
11
12    W1 = parameters['W1']
13    W2 = parameters['W2']
14    A1 = cache['A1']
15    A2 = cache['A2']
16
17    # Backward propagation
18    dZ2 = A2 - Y
19    dW2 = (1/m)*np.dot(dZ2, A1.T)
20    db2 = (1/m)*np.sum(dZ2, axis=1, keepdims=True)
21    dZ1 = np.dot(W2.T, dZ2) * (1-np.power(A1, 2))
22    dW1 = (1/m)*np.dot(dZ1, X.T)
23    db1 = (1/m)*np.sum(dZ1, axis=1, keepdims=True)
24
25    grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}
26    return grads

```

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

```

1 # Step 3 (part 4): Update parameters using gradient descent
2 def update_parameters(parameters, grads, learning_rate = 1.2):
3     """
4     Updates parameters using the gradient descent update rule given above
5
6     Arguments:
7     parameters -- python dictionary containing your parameters
8     grads -- python dictionary containing your gradients
9     """
10    W1 = parameters['W1']
11    b1 = parameters['b1']
12    W2 = parameters['W2']
13    b2 = parameters['b2']
14
15    dW1 = grads['dW1']
16    db1 = grads['db1']
17    dW2 = grads['dW2']
18    db2 = grads['db2']
19
20    # Update rule for each parameter
21    W1 = W1 - learning_rate * dW1
22    b1 = b1 - learning_rate * db1
23    W2 = W2 - learning_rate * dW2
24    b2 = b2 - learning_rate * db2
25
26    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
27    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.

```

1 def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
2     """
3         Arguments:
4             X -- dataset of shape (2, number of examples)
5             Y -- labels of shape (1, number of examples)
6             n_h -- size of the hidden layer
7             num_iterations -- Number of iterations in gradient descent loop
8             print_cost -- if True, print the cost every 1000 iterations
9
10    Returns:
11        parameters -- parameters learnt by the model. They can then be used to predict.
12        """
13    np.random.seed(3)
14    n_x = layer_sizes(X, Y)[0]
15    n_y = layer_sizes(X, Y)[2]
16
17    # Initialize parameters
18    parameters = initialize_parameters(n_x, n_h, n_y)
19
20    # Loop (gradient descent)
21    for i in range(0, num_iterations):
22        # Forward propagation. Outputs: "A2, cache".
23        A2, cache = forward_propagation(X, parameters)
24
25        # Cost function. Outputs: "cost".
26        cost = compute_cost(A2, Y, parameters)
27
28        # Backpropagation. Outputs: "grads".
29        grads = backward_propagation(parameters, cache, X, Y)
30
31        # Gradient descent parameter update. Outputs: "parameters".
32        parameters = update_parameters(parameters, grads)
33
34        # Print the cost every 1000 iterations
35        if print_cost and i % 1000 == 0:
36            print ("Cost after iteration %i: %f" %(i, cost))
37
38    return parameters

```

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

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

```

1 def predict(parameters, X):
2     """
3         Using the learned parameters, predicts a class for each example in X
4
5         Arguments:
6             parameters -- python dictionary containing your parameters
7             X -- input data of size (n_x, m)
8             """
9
10        # Computes probabilities using forward propagation (threshold of 0.5)
11        A2, cache = forward_propagation(X, parameters)
12        predictions = (A2 > 0.5) # vector of predict 0/1 values
13
14    return predictions

```

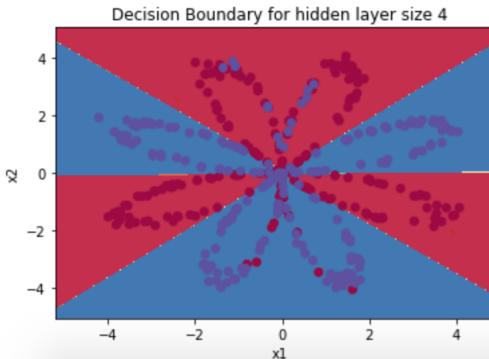
Now that we have a way to create a model and make predictions, we can use this on our planar dataset. We will use a hidden layer of size 4 for our Neural Network.

```

1 # Build a model with a n_h-dimensional hidden layer
2 parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000, print_cost=True)
3
4 # Plot the decision boundary
5 plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
6 plt.title("Decision Boundary for hidden layer size " + str(4))
7
8 # Print accuracy
9 predictions = predict(parameters, X)
10 print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T)) / float(Y.size)*100) + '%') # 90%
11
```

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 at 0x7f31307ead68>

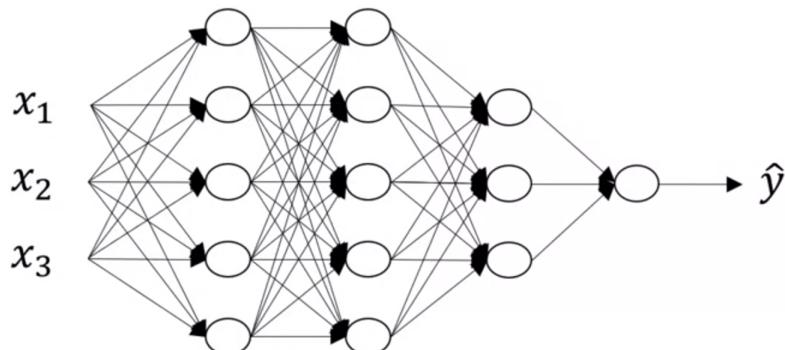


We can see that per 1000 iterations, the cost continued to decrease. The decision boundaries were quite accurate as well. Accuracy is really high (90%) compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision 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 example 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 $l = 1, \dots, L$. 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:

$$z^{[l]} = (n^{[l]}, 1)$$

$$W^{[l]} = (n^{[l]}, n^{[l-1]})$$

$$a^{[l]} = (n^{[l-1]}, 1)$$

$$b^{[l]} = (n^{[l]}, 1)$$

For m training examples:

$$Z^{[l]} = (n^{[l]}, m)$$

$$W^{[l]} = (n^{[l]}, n^{[l-1]})$$

$$A^{[l]} = (n^{[l-1]}, m)$$

$$b^{[l]} = (n^{[l]}, 1)$$

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

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

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

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

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

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

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

A quick note on **parameters 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 (Building)

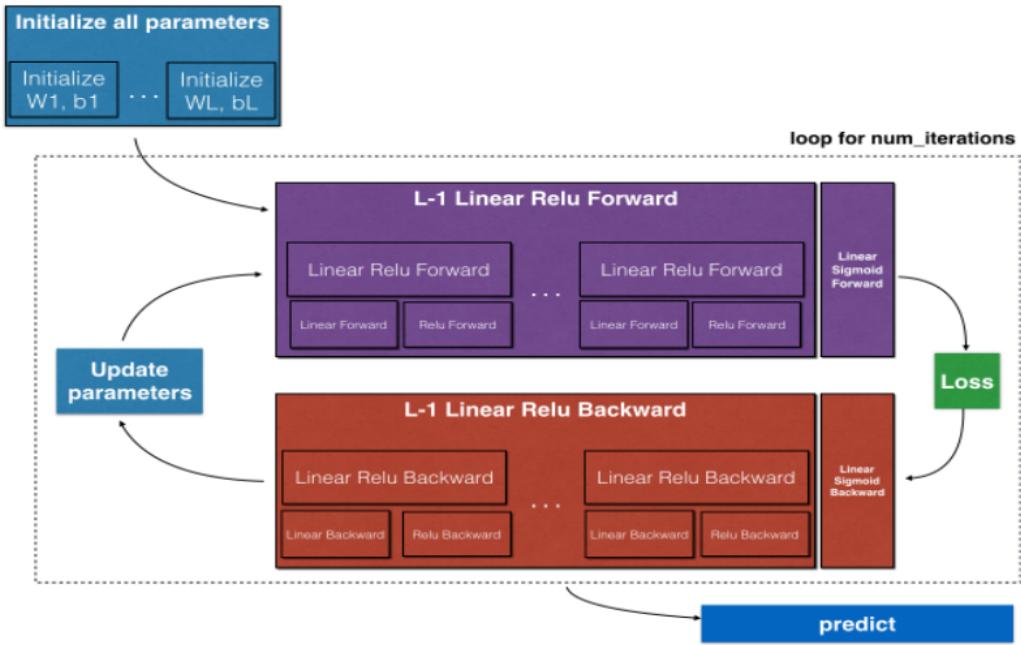
For this week, we will build an L-Layer Neural Network. In the next assignment, we will use this model for image classification. Lets begin by importing the necessary packages.

```
1 import numpy as np
2 import h5py
3 import matplotlib.pyplot as plt
4 from testCases_v4a import *
5 from dnn_utils_v2 import sigmoid, sigmoid_backward, relu, relu_backward
6
7 %matplotlib inline
8 plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
9 plt.rcParams['image.interpolation'] = 'nearest'
10 plt.rcParams['image.cmap'] = 'gray'
11
12 %load_ext autoreload
13 %autoreload 2
14
15 np.random.seed(1)
```

To build your neural network, you will be implementing several “helper functions”. These helper functions will be used in the next assignment to build a two-layer and an L-layer neural network. Here is an **outline of the assignment**:

- 1) Initialize the parameters for a two-layer network and for an L -layer neural network.
- 2) Implement the forward propagation module (shown in purple in the figure below).
 - Complete the LINEAR part of a layer’s forward propagation step (resulting in $Z^{[l]}$).
 - We give you the ACTIVATION function (relu/sigmoid).
 - Combine the previous two steps into a new [LINEAR→ACTIVATION] forward function.
 - Stack the [LINEAR→RELU] forward function $L-1$ time (for layers 1 through $L-1$) and add a [LINEAR→SIGMOID] at the end (for the final layer L). This gives you a new `L_model_forward` function.
- 3) Compute the loss.
- 4) Implement the backward propagation module (denoted in red in the figure below).
 - Complete the LINEAR part of a layer’s backward propagation step.
 - We give you the gradient of the ACTIVATE function (relu_backward/sigmoid_backward)
 - Combine the previous two steps into a new [LINEAR→ACTIVATION] backward function.
 - Stack [LINEAR→RELU] backward $L-1$ times and add [LINEAR→SIGMOID] backward in a new `L_model_backward` function
- 5) Finally update the parameters.

Note that for every forward function, there is a corresponding backward function. That is why at every step of your forward module you will be storing some values in a cache. The cached values are useful for computing gradients. In the backpropagation module you will then use the cache to calculate the gradients.



Lets begin by **initializing our parameters**. The first function will be for a *two layer model*.

```

1 def initialize_parameters(n_x, n_h, n_y): # Two Layer Neural Network
2 """
3     Argument:
4     n_x -- size of the input layer
5     n_h -- size of the hidden layer
6     n_y -- size of the output layer
7 """
8 np.random.seed(1)
9
10 W1 = np.random.randn(n_h, n_x) * 0.01 # Weight matrix
11 b1 = np.zeros((n_h, 1)) # Bias vector
12 W2 = np.random.randn(n_y, n_h) * 0.01 # Weight matrix
13 b2 = np.zeros((n_y, 1)) # Bias vector
14
15 parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
16 return parameters

```

The initialization for a *L-layer neural network* is more complicated because there are many more weight matrices and bias vectors. Remember that when we compute $WX + b$ in python, it carries out broadcasting. For example, if:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb +qe + rh) + u & (pc + qf + ri) + u \end{bmatrix}$$

We will store $n^{[l]}$, the number of units in different layers, in a variable ‘layer_dims’. For example, the ‘layer_dims’ for the “Planar Data classification model” from last week would have been [2,4,1]: There were two inputs, one hidden layer with 4 hidden units, and an output layer with 1 output unit. This means ‘W1’s shape was (4,2), ‘b1’ was (4,1), ‘W2’ was (1,4) and ‘b2’ was (1,1). Now you will generalize this to L layers.

```

1 def initialize_parameters_deep(layer_dims):
2 """
3 Arguments:
4 layer_dims -- python array containing the dimensions of each layer in our network
5
6 Returns:
7 parameters -- python dict containing your parameters "W1", "b1", ..., "WL", "bL"
8 """
9 np.random.seed(3)
10 parameters = {}
11 L = len(layer_dims) # number of layers in the network
12
13 for l in range(1, L):
14     parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * 0.01
15     parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))
16
17 return parameters

```

Now that you have initialized your parameters, you will do the **forward propagation** module. You will start by implementing some basic functions that you will use later when implementing the model:

- 1) LINEAR
- 2) LINEAR → ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- 3) [LINEAR → RELU] × (L-1) → LINEAR → SIGMOID (whole model)

The linear forward module (vectorized over all the examples) computes the following equations:

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

```

1 def linear_forward(A, W, b): # Step 1 from above
2 """
3 Implement the linear part of a layer's forward propagation.
4 """
5 Z = np.dot(W, A) + b # pre-activation parameter (Z = WA + b)
6 cache = (A, W, b)
7
8 return Z, cache
9
10 def linear_activation_forward(A_prev, W, b, activation): # Step 2 from above
11 """
12 Implement the forward propagation for the LINEAR->ACTIVATION layer
13
14 Arguments:
15 A_prev -- activations from previous layer: (size of previous layer, # of examples)
16 W -- weights matrix: shape (size of current layer, size of previous layer)
17 b -- bias vector, numpy array of shape (size of the current layer, 1)
18 activation -- activation to be used in this layer, stored as: "sigmoid" or "relu"
19 """
20 if activation == "sigmoid":
21     Z, linear_cache = linear_forward(A_prev, W, b)
22     A, activation_cache = sigmoid(Z)
23
24 elif activation == "relu":
25     Z, linear_cache = linear_forward(A_prev, W, b)
26     A, activation_cache = relu(Z)
27
28 cache = (linear_cache, activation_cache)
29 return A, cache # post-activation value, cache

```

Mathematical relation is: $A^{[l]} = g(Z^{[l]}) = g(W^{[l]} A^{[l-1]} + b^{[l]})$ where the activation “g” can be sigmoid() or relu(). Note that these are pre-built functions given to us ([click here for function details](#)).

For even more convenience when implementing the L -layer Neural Net, you will need a function that replicates the previous one ('linear_activation_forward' with RELU) $L - 1$ times, then follows that with one 'linear_activation_forward' with SIGMOID.

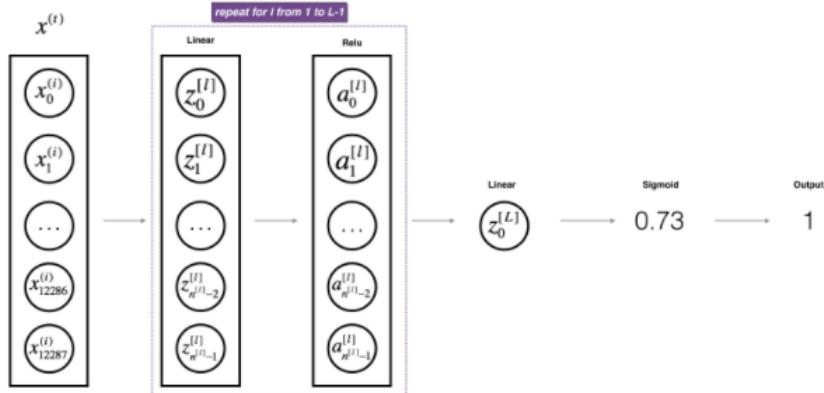


Figure 2 : [LINEAR -> RELU] \times $(L-1)$ -> LINEAR -> SIGMOID model

```

1 def L_model_forward(X, parameters):
2     """
3         Implement forward propagation for the [LINEAR -> RELU]* $(L-1)$  -> LINEAR -> SIGMOID
4
5     Arguments:
6     X -- data, numpy array of shape (input size, number of examples)
7     parameters -- output of initialize_parameters_deep()
8
9     Returns:
10    AL -- last post-activation value
11    caches -- list of caches containing every cache of linear_activation_forward()
12        (indexed from 0 to L-1)
13
14    caches = []
15    A = X
16    L = len(parameters) // 2 # number of layers in the neural network
17
18    # Implement [LINEAR -> RELU]* $(L-1)$ . Add "cache" to the "caches" list.
19    for l in range(1, L):
20        A_prev = A
21        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)],
22                                              parameters['b' + str(l)], 'relu')
23        caches.append(cache)
24
25    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
26    AL, cache = linear_activation_forward(A, parameters['W' + str(L)],
27                                          parameters['b' + str(L)], 'sigmoid')
28    caches.append(cache)
29
30    return AL, caches # y-hat (predictions), cache

```

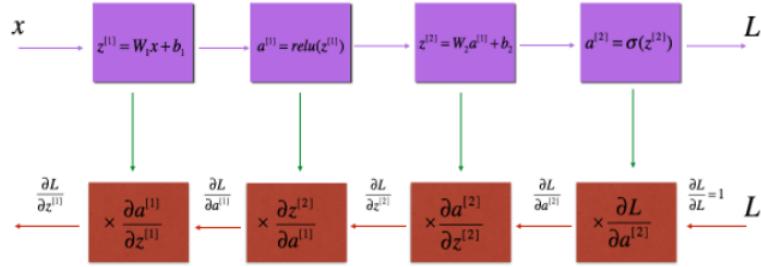
Now we have a full forward propagation that takes the input X and outputs a row vector $A^{[L]}$ containing your predictions. It also records all intermediate values in "caches". Using $A^{[L]}$, we can now **compute the cost** of your predictions to check if the model is actually learning. We will compute *Cross-Entropy* cost, J , using:

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

```

1  def compute_cost(AL, Y):
2      """
3          Implement the cost function defined by equation (7).
4
5          Arguments:
6          AL -- probability vector corresponding to your label predictions, shape (1, m)
7          Y -- true "label" vector, shape (1, m)
8
9          Returns:
10         cost -- cross-entropy cost
11         """
12
13         m = Y.shape[1]
14
15         cost = -(1/m)*np.sum(np.multiply(Y, np.log(AL)) + np.multiply((1-Y), np.log(1-AL)))
16
17         cost = np.squeeze(cost) # Reduce dimensions to just an integer
18         return cost

```

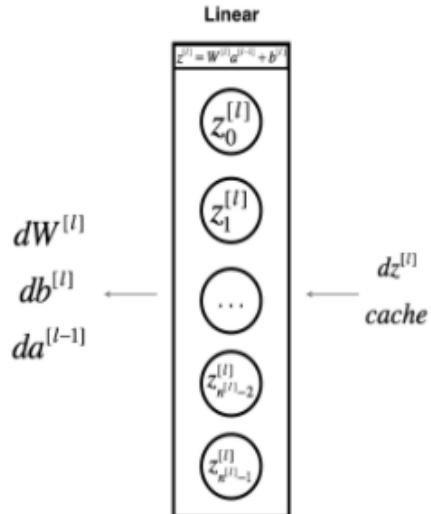


Just like with forward propagation, you will implement helper functions for **back propagation**. Remember that back propagation is used to calculate the gradient of the loss function with respect to the parameters. We are going to build the backward propagation in three steps:

- 1) LINEAR backward
- 2) LINEAR → ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation
- 3) [LINEAR → RELU] × (L-1) → LINEAR → SIGMOID backward (whole model).

Lets begin with step 1. For layer l , the linear part is $Z^{[l]}$ followed by an activation. Suppose you have already calculated the derivative $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$. You want to get $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$. The process and formulas for this step are:

$$\begin{aligned} dW^{[l]} &= \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \\ db^{[l]} &= \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^m dZ^{[l](i)} \\ dA^{[l-1]} &= \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \end{aligned}$$



```

1 def linear_backward(dZ, cache):
2     """
3         Implement the linear portion of backward propagation for a single layer (layer l)
4
5     Arguments:
6         dZ -- Gradient of the cost with respect to the linear output (of current layer l)
7         cache -- (A_prev, W, b) coming from the forward propagation in the current layer
8
9     Returns:
10        dA_prev -- Gradient of the cost with respect to the activation (of the previous
11                  layer l-1), same shape as A_prev
12        dW -- Gradient of the cost with respect to W (current layer l), same shape as W
13        db -- Gradient of the cost with respect to b (current layer l), same shape as b
14    """
15    A_prev, W, b = cache
16    m = A_prev.shape[1]
17
18    dW = (1/m) * np.dot(dZ, A_prev.T)
19    db = (1/m) * np.sum(dZ, axis=1, keepdims=True)
20    dA_prev = np.dot(W.T, dZ)
21
22    return dA_prev, dW, db

```

For step 2, we want to write a function that uses linear_backward and also computes the backward step for the activation function. The sigmoid_backward and relu_backward are both pre-built functions ([click here for details](#)). If g is the activation function, the two pre-built functions compute:

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

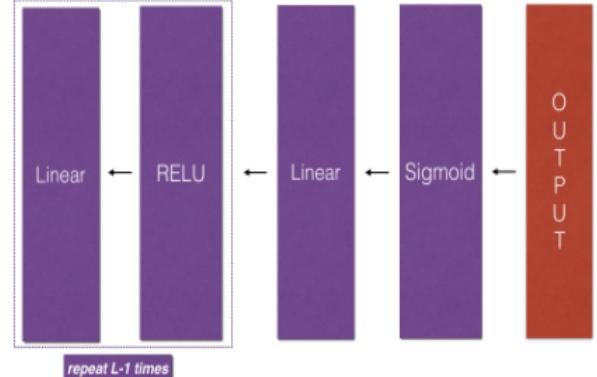
```

1 def linear_activation_backward(dA, cache, activation):
2     """
3         Implement the backward propagation for the LINEAR->ACTIVATION layer.
4
5     Arguments:
6         dA -- post-activation gradient for current layer l
7         cache -- (linear_cache, activation_cache) we store for computing backward
8                 propagation efficiently
9         activation -- stored as a text string: "sigmoid" or "relu"
10
11    Returns:
12        dA_prev -- Gradient of the cost with respect to the activation of the previous
13                  layer
14        dW -- Gradient of the cost with respect to W (current layer l), same shape as W
15        db -- Gradient of the cost with respect to b (current layer l), same shape as b
16    """
17    linear_cache, activation_cache = cache
18
19    if activation == "relu":
20        dZ = relu_backward(dA, activation_cache)
21        dA_prev, dW, db = linear_backward(dZ, linear_cache)
22
23    elif activation == "sigmoid":
24        dZ = sigmoid_backward(dA, activation_cache)
25        dA_prev, dW, db = linear_backward(dZ, linear_cache)
26
27    return dA_prev, dW, db

```

For step 3, we will implement the backward function for the whole network. Recall that when you implemented the ‘L_model_forward’ function, at each iteration, you stored a cache which contains (X , W , b , and z). In the back propagation module, you will use those variables to compute the gradients. Therefore, in the ‘L_model_backward’ function, you will iterate through all the hidden layers backward, starting from layer L . On each step, you will use the cached values for layer l to back propagate through layer l .

To initialize back propagation, we need the derivative of $A^{[L]}$. Through calculus, we find that this value is $-(\frac{Y}{AL} - \frac{1-Y}{1-AL})$. We can then feed this into linear_activation_backward using ‘sigmoid’. After that, we loop through and compute linear_activation_backward using ‘relu’ for all other layers.



```

1 def L_model_backward(AL, Y, caches):
2     """
3         Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR ->
4         SIGMOID group
5
6     Arguments:
7         AL -- probability vector, output of the forward propagation (L_model_forward())
8         Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
9         caches -- list of caches containing:
10            every linear_activation_forward() with "relu" (caches[l] for l in range 0 to (L-2))
11            the cache of linear_activation_forward() with "sigmoid" (it's caches[L-1])
12
13     Returns:
14         grads -- A dictionary with the gradients
15         grads["dA" + str(l)] = ...
16         grads["dW" + str(l)] = ...
17         grads["db" + str(l)] = ...
18
19         grads = {}
20         L = len(caches) # the number of layers
21         m = AL.shape[1]
22         Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
23
24         # Initializing the backpropagation
25         dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
26
27         # Lth layer (SIGMOID -> LINEAR) gradients.
28         current_cache = caches[L-1]
29         grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] =
30             linear_activation_backward(dAL, current_cache, 'sigmoid')
31
32         for l in reversed(range(L-1)): # Loop from l=L-2 to l=0
33             # lth layer: (RELU -> LINEAR) gradients.
34             current_cache = caches[l]
35             dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads['dA'+str(l+1)],
36                             current_cache, 'relu')
37             grads["dA" + str(l)] = dA_prev_temp
38             grads["dW" + str(l+1)] = dW_temp
39             grads["db" + str(l+1)] = db_temp
40
41     return grads

```

The final part of our model is to **update the parameters** using gradient descent. Recall that:

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

where α is the learning rate. After computing the updated parameters, store them in the parameters dictionary. We will use a loop to perform this on every l layer.

```

1 def update_parameters(parameters, grads, learning_rate):
2     """
3         Update parameters using gradient descent
4
5     Arguments:
6         parameters -- python dictionary containing your parameters
7         grads -- python dictionary containing your gradients, output of L_model_backward
8
9     Returns:
10    parameters -- python dictionary containing your updated parameters
11    parameters["W" + str(l)] = ...
12    parameters["b" + str(l)] = ...
13    """
14    L = len(parameters) // 2 # number of layers in the neural network
15
16    # Update rule for each parameter.
17    for l in range(L):
18        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate *
19                                grads["dW" + str(l+1)]
20        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate *
21                                grads["db" + str(l+1)]
22
23    return parameters

```

1.3.3 Programming Assignment 2 (Classifier)

You will use the functions you'd implemented in the previous assignment to build a deep network, and apply it to cat vs non-cat classification. Lets start by importing all the packages we will need to use.

```

1 import time
2 import numpy as np
3 import h5py
4 import matplotlib.pyplot as plt
5 import scipy
6 from PIL import Image
7 from scipy import ndimage
8 from dnn_app_utils_v3 import *
9
10 %matplotlib inline
11 plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
12 plt.rcParams['image.interpolation'] = 'nearest'
13 plt.rcParams['image.cmap'] = 'gray'
14
15 %load_ext autoreload
16 %autoreload 2
17
18 np.random.seed(1)

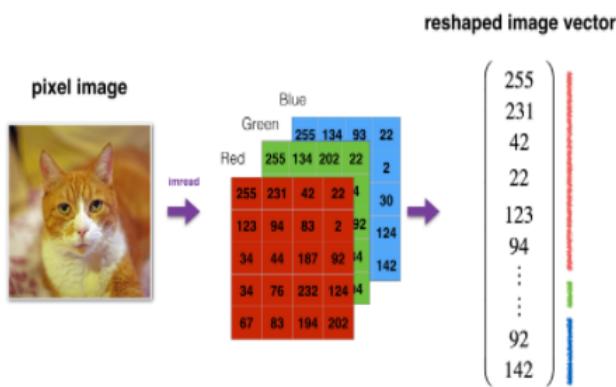
```

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

- a training set of m_train images labeled as cat (1) or non-cat (0)
- a test set of m_test images labeled as cat and non-cat
- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).

```
Number of training examples: 209
Number of testing examples: 50
Each image is of size: (64, 64, 3)
train_x_orig shape: (209, 64, 64, 3)
train_y shape: (1, 209)
test_x_orig shape: (50, 64, 64, 3)
test_y shape: (1, 50)
```

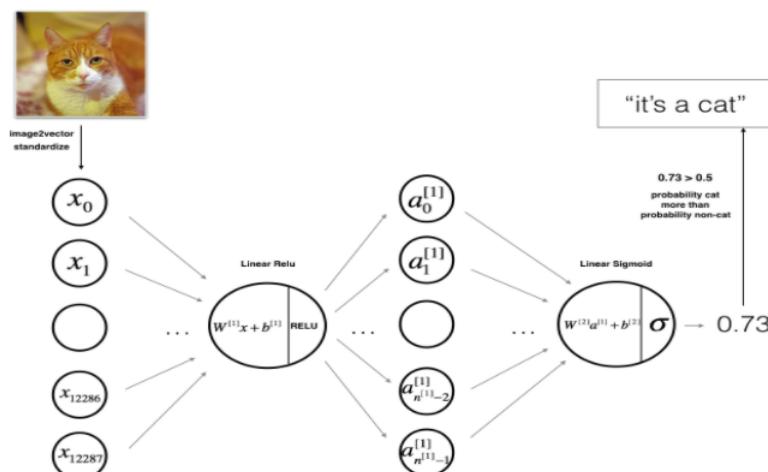
```
1 train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
2
3 m_train = train_x_orig.shape[0] # 209
4 num_px = train_x_orig.shape[1] # 64, image size = (64, 64, 3)
5 m_test = test_x_orig.shape[0] # 50
```



As usual, you **reshape** and **standardize** the images before feeding them to the network. For the .reshape() method, the “-1” makes reshape flatten the remaining dimensions. For the shape, $12288 = 64 \times 64 \times 3$, which is the size of one reshaped image vector.

```
1 # Reshape the training and test examples
2 train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T
3 test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T
4
5 # Standardize data to have feature values between 0 and 1.
6 train_x = train_x_flatten/255. # shape = (12288, 209)
7 test_x = test_x_flatten/255. # shape = (12288, 50)
```

We will start with the **2-Layer Neural Network**. We can see the architecture below:



Detailed architecture of above Two-Layer NN figure:

- The input is a (64,64,3) image which is flattened to a vector of size (12288, 1).
- The corresponding vector: $[x_0, x_1, \dots, x_{12287}]^T$ is then multiplied by $W^{[1]}$ of size ($n^{[1]}, 12288$).
- You then add a bias term and take its relu to get the following vector: $[a_0^{[1]}, a_1^{[1]}, \dots, a_{n^{[1]}-1}^{[1]}]^T$.
- You then repeat the same process.
- You multiply the resulting vector by $W^{[2]}$ and add your intercept (bias).
- Finally, you take the sigmoid of the result. If it is greater than 0.5, you classify it to be a cat.

We will follow the general **methodology** to build the model:

1. Initialize parameters / define hyperparameters
2. Loop for num_iterations:
 - a. Forward propagation
 - b. Compute cost function
 - c. Backward propagation
 - d. Update parameters (using parameters, and grads from backprop)
3. Use trained parameters to predict labels

```

1  ##### CONSTANTS DEFINING THE MODEL #####
2  n_x = 12288 # num_px * num_px * 3
3  n_h = 7
4  n_y = 1
5  layers_dims = (n_x, n_h, n_y) # (12288, 7, 1)
6
7  def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000,
8                      print_cost=False):
9      """
10      Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID .
11
12      Arguments:
13      X -- input data, of shape (n_x, number of examples)
14      Y -- true "label" vector (containing 1 if cat, 0 if non-cat), of shape (1, m)
15      layers_dims -- dimensions of the layers (n_x, n_h, n_y)
16      num_iterations -- number of iterations of the optimization loop
17      learning_rate -- learning rate of the gradient descent update rule
18      print_cost -- If set to True, this will print the cost every 100 iterations
19
20      Returns:
21      parameters -- a dictionary containing W1, W2, b1, and b2
22      """
23  np.random.seed(1)
24  grads = {}
25  costs = [] # to keep track of the cost
26  m = X.shape[1] # number of examples
27  (n_x, n_h, n_y) = layers_dims
28
29  parameters = initialize_parameters(n_x, n_h, n_y) # Initialize parameters dict
30
31  # Get W1, b1, W2 and b2 from the dictionary parameters.
32  W1 = parameters["W1"]
33  b1 = parameters["b1"]
34  W2 = parameters["W2"]
35  b2 = parameters["b2"]
36
37  # Loop (gradient descent)
38  for i in range(0, num_iterations):
39      # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID
40      A1, cache1 = linear_activation_forward(X, W1, b1, 'relu')
41      A2, cache2 = linear_activation_forward(A1, W2, b2, 'sigmoid')

```

```

1   # Compute cost
2   cost = compute_cost(A2, Y)
3
4   # Initializing backward propagation
5   dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))
6
7   # Backward propagation
8   dA1, dW2, db2 = linear_activation_backward(dA2, cache2, 'sigmoid')
9   dA0, dW1, db1 = linear_activation_backward(dA1, cache1, 'relu')
10
11  # Set grads to corresponding values
12  grads['dW1'] = dW1
13  grads['db1'] = db1
14  grads['dW2'] = dW2
15  grads['db2'] = db2
16
17  # Update parameters.
18  parameters = update_parameters(parameters, grads, learning_rate)
19
20  # Retrieve W1, b1, W2, b2 from parameters
21  W1 = parameters["W1"]
22  b1 = parameters["b1"]
23  W2 = parameters["W2"]
24  b2 = parameters["b2"]
25
26  # Print the cost every 100 training example
27  if print_cost and i % 100 == 0:
28      print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
29  if print_cost and i % 100 == 0:
30      costs.append(cost)
31
32  # plot the cost
33  plt.plot(np.squeeze(costs))
34  plt.ylabel('cost')
35  plt.xlabel('iterations (per hundreds)')
36  plt.title("Learning rate =" + str(learning_rate))
37  plt.show()
38
39  return parameters

```

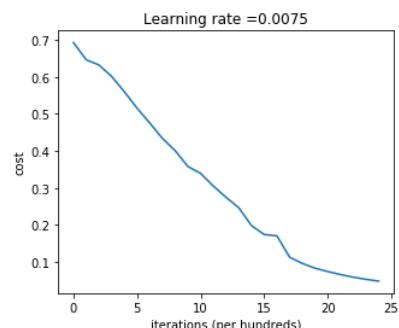
Now that we have a working model we can train the parameters, use them to classify images, and determine our accuracy. *Note*: You may notice that running the model on fewer iterations (say 1500) gives better accuracy on the test set. This is called *early stopping* and we will talk about it in the next course. Early stopping is a way to prevent overfitting.

```

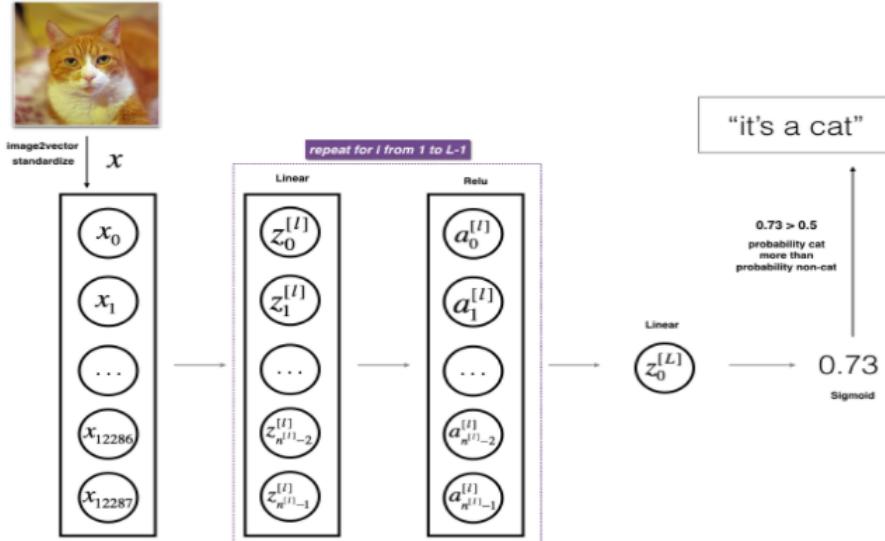
1  parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y),
2                                num_iterations = 2500, print_cost=True)
3
4  predictions_train = predict(train_x, train_y, parameters) # Accuracy = 1.0
5  predictions_test = predict(test_x, test_y, parameters) # 0.72

```

Cost after iteration 0	0.6930497356599888
Cost after iteration 100	0.6464320953428849
...	...
Cost after iteration 2400	0.048554785628770226



Now that we have created a two-layer Neural Network, lets take a look at how to implement a **L-Layer Neural Network**. Below is the general architecture we will use:



Detailed architecture of L-Layer NN:

- The input is a (64,64,3) image which is flattened to a vector of size (12288,1).
- The corresponding vector: $[x_0, x_1, \dots, x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ and then you add the intercept $b^{[1]}$. The result is called the linear unit.
- Next, you take the relu of the linear unit. This process could be repeated several times for each $(W^{[l]}, b^{[l]})$ depending on the model architecture.
- Finally, you take the sigmoid of the final linear unit. If it is greater than 0.5, you classify it as a cat.

```

1  ##### CONSTANTS #####
2  layers_dims = [12288, 20, 7, 5, 1] # 4-layer model
3
4  def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000,
5      print_cost=False):#lr was 0.009
6      """
7          Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.
8
9          Arguments:
10         X -- data, numpy array of shape (num_px * num_px * 3, number of examples)
11         Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, m)
12         layers_dims -- list containing the input size and each layer size, of length (L+1).
13         learning_rate -- learning rate of the gradient descent update rule
14         num_iterations -- number of iterations of the optimization loop
15         print_cost -- if True, it prints the cost every 100 steps
16
17     Returns:
18         parameters -- parameters learnt by the model. They can then be used to predict.
19         """
20         np.random.seed(1)
21         costs = [] # keep track of cost
22
23         parameters = initialize_parameters_deep(layers_dims) # Parameters initialization
24
25         for i in range(0, num_iterations): # Loop (gradient descent)
26             AL, caches = L_model_forward(X, parameters) # Forward propagation
27
28             cost = compute_cost(AL, Y) # Compute cost
29
30             grads = L_model_backward(AL, Y, caches) # Backward propagation.
31
32             parameters = update_parameters(parameters, grads, learning_rate) # Update params.

```

```

1   # Print the cost every 100 training example
2   if print_cost and i % 100 == 0:
3       print ("Cost after iteration %i: %f" %(i, cost))
4   if print_cost and i % 100 == 0:
5       costs.append(cost)
6
7   # plot the cost
8   plt.plot(np.squeeze(costs))
9   plt.ylabel('cost')
10  plt.xlabel('iterations (per hundreds)')
11  plt.title("Learning rate =" + str(learning_rate))
12  plt.show()
13
14 return parameters

```

With our working model we can now **train** the parameters and make **predictions** to determine our accuracy.

```

1 parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations = 2500,
2                             print_cost = True)
3
4 pred_train = predict(train_x, train_y, parameters) # Accuracy: 0.985645933014
5 pred_test = predict(test_x, test_y, parameters) # Accuracy: 0.8

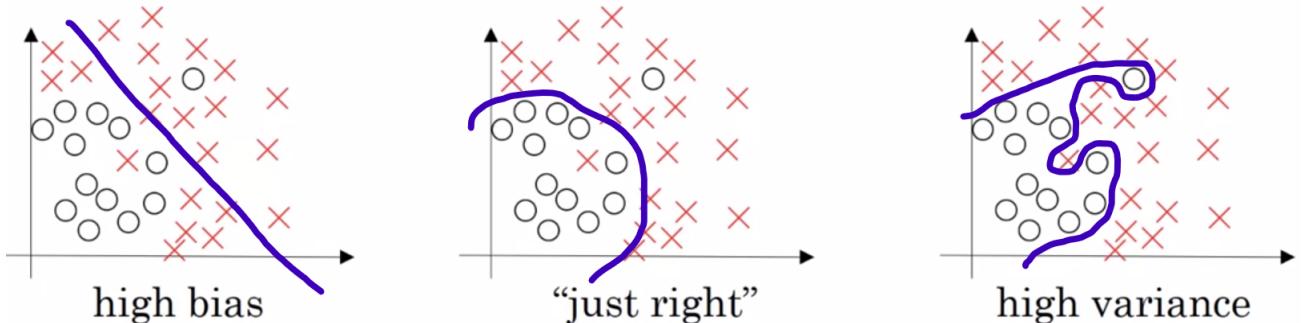
```

It seems that your 4-layer neural network has better performance (80%) than your 2-layer neural network (72%) on the same test set. This is good performance for this task. Though in the next course on “Improving deep neural networks” you will learn how to obtain even higher accuracy by systematically searching for better hyperparameters (learning_rate, layers_dims, num_iterations, and others you’ll also learn in the next course).

2 Improving Deep Neural Networks

2.1 Regularizing & Optimizing Your Neural Network

2.1.1 Bias/Variance



- *High Bias* leads to **underfitting**, where it cannot find distinct trends in the data.
- "Just right" is where we aim to be, where our model is fitted but not too close.
- *High Variance* leads to **overfitting**, where it is fit too close to the data.

Lets take a look at an example for a cat classification model. We know that humans will have approximately 0% error, so we say that the **Optimal (Bayes) error** is approximately 0% as well. For the following errors, we say:

Train Set Error	1%	15%	15%	0.5%
Dev Set Error	11%	16%	30%	1%
	High Var	High Bias	High Var & Bias	Low Var & Bias

A general rule is that you look at the *training set error* to get an understanding of how bad your **bias** is. Then, looking at how much higher your *dev set error* will give you an idea of how high your **variance** is. All of this is made under the assumption that your Bayes error is quite small and your training/dev sets are drawn from the same distribution.

When training a Neural Network, we want to follow these **general guidelines**:

- 1) Does the algorithm have high bias? (look at training data performance).
 - Possible Solutions: Bigger network (more n_h), train longer, different NN architecture.
- 2) Once bias is reduced, check for a variance problem. (look at dev set data performance).
 - Possible Solutions: More data, Regularization, different NN architecture.

There used to be a "*bias variance trade-off*", where an increase/decrease in one aspect would result in the opposite effect for the other aspect. However, nowadays this is no longer the case. As long as we can get more data to *decrease* our variance, as well as train a bigger network to *decrease* bias, both aspects will decrease instead of having a trade-off.

2.1.2 Regularization

If you suspect that your Neural Network is *overfitting* your data, meaning you have a high variance problem, one of the first things you should try is **Regularization**.

Lets first look at this idea in terms of Logistic Regression. Recall that we want to find the min of $J(w,b)$. We can do this by adding a *regularization parameter* (λ) multiplied by the *norm* of w squared ($\|w\|^2$).

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2$$

Above, the norm of w squared is called *L₂ regularization*: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

The **regularization parameter** (λ) is usually set with the dev set by using *cross-validation*, where you try a variety of values and see which performs the best.

To regularize over a Neural Network, we use the **Forbenius norm** (weight decay) of a matrix:

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|W^{[l]}\|_F^2$$

$$\|W^{[l]}\|_F^2 = \sum_{i=1}^{n^l} \sum_{j=1}^{n^{[l-1]}} (W_{i,j}^{[l]})^2$$

The rows “i” of the matrix should be the number of neurons in the current layer $n^{[l]}$, whereas the columns “j” of the weight matrix should equal the number of neurons in the previous layer $n^{[l-1]}$.

To implement **gradient descent**, we know use these updated formulas:

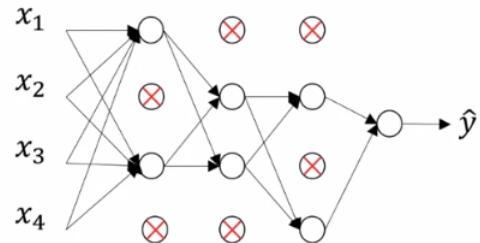
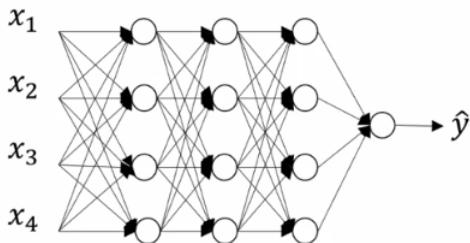
$$dW^{[l]} = (\text{backprop}) + \frac{\lambda}{m} W^{[l]}$$

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$

The way the regularization prevents overfitting is that the larger the λ value is, the closer the weight matrix will be to 0. Doing this will move you from high variance (right picture in above diagram) to high bias (left picture above). The concept is that the λ value will result to the “just right” case (middle picture). The λ value causes hidden units to have a much smaller impact, so it leads to a simpler/smaller network that is less prone to overfitting.

One example is by using tanh as our activation function, having a large λ value will cause $W^{[l]}$ to be smaller, meaning that $z^{[l]}$ will also be smaller (range of values will be reduced and more focused in the middle around 0). This causes every layer to be approximately linear, meaning that it will cause less overfitting by preventing complex decision boundaries and using a straighter line for the boundary.

2.1.3 Dropout Regularization



Dropout regularization is used to reduce Neural Networks by randomly choosing hidden units to remove from a Network, along with removing all inputs/outputs coming from the removed nodes. You do this for each training example, so each example i is trained on a different reduced Neural Network.

One way to implement dropout regularization is called **inverted dropout**. Lets use $l = 3$.

```

1  keep_prob = 0.8 # can be any value in range (0,1)
2  d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob # boolean array
3  a3 = np.multiply(a3, d3) # reduce activation for 0 valued d3 nodes (20% reduced)
4  a3 /= keep_prob # bumps values up by 20% so we dont change expected value for z4

```

Note that when make predictions on the test data, we use the dropout activation matrices. We do not want to calculate these values at test time because it will add noise to our predictions.

A few other forms of regularization that we can use are:

- Data Augmentation : manipulating the data (ex: flipping/cropping an image) to create more data.
- Early stopping : during gradient descent, we stop training when we hit a certain value.

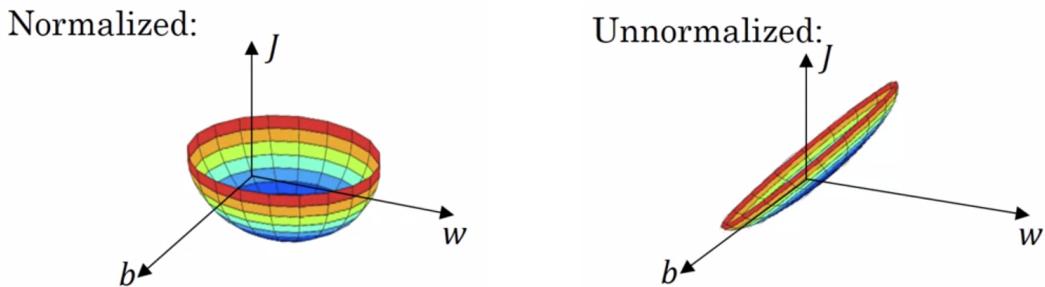
Orthogonalization tells us that we only want to focus on one task at a time. The first and most important is optimizing the cost function J so that we minimize $J(w, b)$. The second is then to not overfit our data, i.e. using regularization techniques in order to do this. L_2 regularization, even though you have to try a lot of values for λ and can be more expensive with larger models, is still better than early stopping.

2.1.4 Normalizing Inputs

One technique used to speed up our training is if we **normalize** our inputs. We can do this in two steps:

- 1) Subtract the mean, $\mu = \sum_{i=1}^m x^{(i)}$, from each training example so that $x = x - \mu$
- 2) Normalize the variance, $\sigma = \frac{1}{m} \sum_{i=1}^m x^{(i)} * * 2$, element-wise squared, so that x / σ

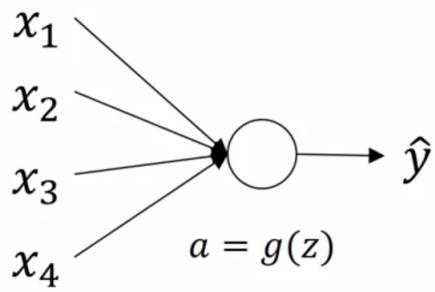
$$\frac{x - \mu}{\sigma}$$



We want to normalize our data because if we don't, then the cost function can be a very elongated. But if we normalize, then our cost function will be more symmetrical and need less steps for gradient descent (it can go straight to a minimum rather than oscillating).

2.1.5 Weight Initialization

When you begin to use much deeper Neural Networks with many hidden layers, you want to be careful in your choice of random initialization for the weights matrix. Lets first take a look at an example for a single neuron:

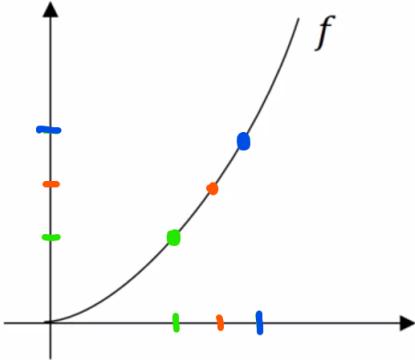


We know that $z = w_1x_1 + w_2x_2 + \dots + w_nx_n$ and we will set $b=0$. In order to make z not explode/shrink, we want to conclude that the larger n is, the smaller we want w_i to be. We can do this by setting the $Var(w_i) = \frac{1}{n}$. Note that if you are using a ReLU function, then we want $Var(w_i) = \frac{2}{n}$. In Python, the general formula for ReLU we use would be:

$$W^{[l]} = np.random.randn(shape) * np.sqrt(\frac{2}{n^{[l-1]}})$$

2.1.6 Gradient Checking

When you implement back propagation you'll find that there's a test called **gradient checking** that can help you make sure that your implementation of back prop is correct. In order to build up to this, lets first talk about how to *numerically approximate computations of gradients*.



We know that:
 $f(\theta) = \theta^3$
 $\epsilon = 0.01$
Green x-axis = $\theta - \epsilon = 0.99$
Blue x-axis = $\theta + \epsilon = 1.01$
Red x-axis = $\theta = 1$
Green y-axis = $f(\theta - \epsilon)$
Blue y-axis = $f(\theta + \epsilon)$
Red y-axis = $f(\theta)$

We want to find the gradient for the triangle created by the green and blue dot. This means that the *height* = $f(\theta + \epsilon) - f(\theta - \epsilon)$, and the *width* = 2ϵ . This means that:

$$g(\theta) = \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

If we plug in our values from above, we would see our **two-sided** gradient = 3.0001 (error = 0.0001). If we had done it how we had previously using a **one-sided** approach (only using θ and $\theta + \epsilon$ for our triangle), our gradient = 3.0301 (error = 0.0301). Note that in code, the two sided approach is twice as slow, but works much better since it is more accurate. Our big-O notation for error is: $O(\epsilon^2)$.

Now that we've seen that, lest take a look at how to implement **gradient checking**. We first must:

- 1) Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape (concatenate) into a big vector θ .
 - Note that now, $J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = J(\theta)$.
- 2) Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape (concatenate) into a big vector $d\theta$.

Now that we have $d\theta$, we want to see if it is the gradient of $J(\theta)$. In order to find this our, we use the following formula for every i (component of θ):

$$\begin{aligned} d\theta_{approx} &= \frac{J(\theta_1, \theta_2, \dots, \theta_{i+\epsilon}, \dots) - J(\theta_1, \theta_2, \dots, \theta_{i-\epsilon}, \dots)}{2\epsilon} \\ &\approx d\theta[i] = \frac{\partial J}{\partial d\theta_i} \end{aligned}$$

The final step is to see if our $d\theta_{approx} \approx d\theta$. We can do this by checking the difference between the two vectors by computing the Euclidean distance and normalizing it:

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 - \|d\theta\|_2}$$

If the value outputted by the formula above is similar to the value of ϵ , then we know our back propagation is working. However, the farther away we get from the value, the more we need to go back and check that our implementation is correct.

Notes on implementation of Gradient checking:

- Don't use in training, only to debug.
- If algorithm fails grad check, look at components to try to identify bugs.
- Remember regularization (add on regularization term to $J(\theta)$ in grad check).
- This does not work with dropout regularization.

2.1.7 Programming Assignment (Initialization)

In this assignment we will see how different initializations lead to different results. You will use a 3-layer neural network (already implemented for you). The initialization methods you will experiment with are:

- *Zeros initialization* - setting initialization = “zeros” in the input argument.
- *Random initialization* - setting initialization = “random” in the input argument. This initializes the weights to large random values.
- *He initialization* - setting initialization = “he” in the input argument. This initializes the weights to random values scaled according to a paper by He et al., 2015.

```
1 def model(X, Y, learning_rate = 0.01, num_iterations = 15000, print_cost = True,
2           initialization = "he"):
3     """
4         Implements a 3-layer neural network: LINEAR->RELU->LINEAR->RELU->LINEAR->SIGMOID.
5
6     Arguments:
7     X - input data, of shape (2, number of examples)
8     Y - true "label" vector (0 for red dots; 1 for blue dots), of shape (1, m)
9     print_cost - if True, print the cost every 1000 iterations
10    initialization - flag to choose initialization to use ("zeros","random" or "he")
11
12    grads = {}
13    costs = [] # to keep track of the loss
14    m = X.shape[1] # number of examples
15    layers_dims = [X.shape[0], 10, 5, 1]
16
17    # Initialize parameters dictionary.
18    if initialization == "zeros":
19        parameters = initialize_parameters_zeros(layers_dims)
20    elif initialization == "random":
21        parameters = initialize_parameters_random(layers_dims)
22    elif initialization == "he":
23        parameters = initialize_parameters_he(layers_dims)
24
25    # Loop (gradient descent)
26    for i in range(0, num_iterations):
27        # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID.
28        a3, cache = forward_propagation(X, parameters)
29
30        cost = compute_loss(a3, Y) # Loss
31
32        grads = backward_propagation(X, Y, cache) # Backward propagation.
33
34        # Update parameters.
35        parameters = update_parameters(parameters, grads, learning_rate)
36
37        # Print the loss every 1000 iterations
38        if print_cost and i % 1000 == 0:
39            print("Cost after iteration {}: {}".format(i, cost))
40            costs.append(cost)
41
42    # plot the loss
43    plt.plot(costs)
44    plt.ylabel('cost')
45    plt.xlabel('iterations (per hundreds)')
46    plt.title("Learning rate = " + str(learning_rate))
47    plt.show()
48
49    return parameters
```

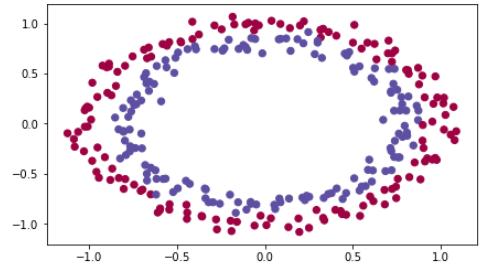
Lets import the packages and planar dataset that we want to classify.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import sklearn
4 import sklearn.datasets

5
6 # load image dataset: blue/red dots in circles
7 train_X, train_Y, test_X, test_Y = load_dataset()

```



First, lets use **zero initialization**. There are two types of parameters to initialize in a neural network:

- The weight matrices ($W^{[1]}, W^{[2]}, W^{[3]}, \dots, W^{[L-1]}, W^{[L]}$)
- The bias vectors ($b^{[1]}, b^{[2]}, b^{[3]}, \dots, b^{[L-1]}, b^{[L]}$)

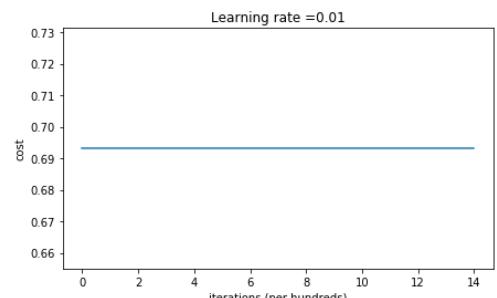
You'll see later that this does not work well since it fails to "break symmetry".

```

1 def initialize_parameters_zeros(layers_dims):
2     """
3         layer_dims -- python array (list) containing the size of each layer.
4     """
5     parameters = {}
6     L = len(layers_dims) # number of layers in the network
7
8     for l in range(1, L):
9         parameters['W' + str(l)] = np.zeros((layers_dims[l], layers_dims[l-1]))
10        parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
11    return parameters # dict containing: "W1", "b1", ... , "WL", "bL"
12
13 parameters = model(train_X, train_Y, initialization = "zeros")
14 predictions_train = predict(train_X, train_Y, parameters) # 0.5
15 predictions_test = predict(test_X, test_Y, parameters) # 0.5

```

If you were to run the above code, we would see that for each iterations the cost is not changing. The performance is really bad, and if you were to print the predictions for the training/test set, you would see that the algorithm is predicting 0's every time. Initializing the weights to zero fails to break symmetry, and is similar to training on $n^{[l]} = 1$ for every layer. It is only okay to initialize b to zeros, but never W to zeros.



To break symmetry, lets initialize the weights randomly. Following **random initialization**, each neuron can then proceed to learn a different function of its inputs. Note that for W , we are initializing to large random values scaled by 10.

```

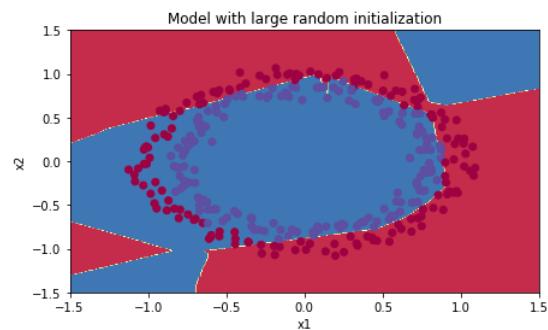
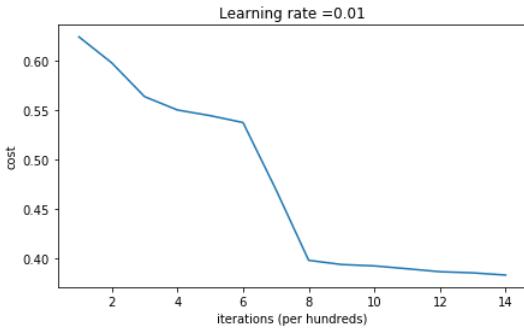
1 def initialize_parameters_random(layers_dims):
2     """
3         layer_dims -- python array (list) containing the size of each layer.
4     """
5     np.random.seed(3)
6     parameters = {}
7     L = len(layers_dims) # integer representing the number of layers
8
9     for l in range(1, L):
10        parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1]) * 10
11        parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
12
13 return parameters

```

```

1 parameters = model(train_X, train_Y, initialization = "random")
2 predictions_train = predict(train_X, train_Y, parameters) # 0.83
3 predictions_test = predict(test_X, test_Y, parameters) # 0.86
4
5 plt.title("Model with large random initialization")
6 axes = plt.gca()
7 axes.set_xlim([-1.5,1.5])
8 axes.set_ylim([-1.5,1.5])
9 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)

```



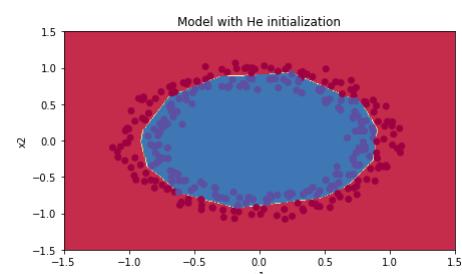
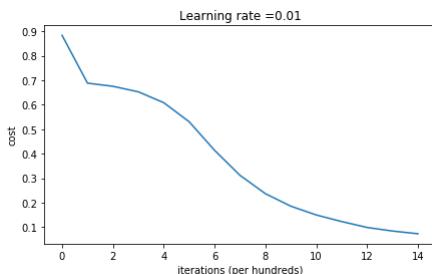
If we were to run the above, although the first iteration (0) is infinity, the cost goes down each 1000 iterations, thus we have broken symmetry. If you train this network longer you will see better results, but initializing with overly large random numbers slows down the optimization. Overall, using very large random value to initialize weights does not work very well.

Finally, we will try **He Initialization**. This is very similar to *Xavier Initialization* except instead of using a scaling factor of 1, we use a scaling factor of 2. Note that this is recommended with layers that use a ReLU activation function. We use : $\sqrt{\frac{2}{\text{previous layer dimensions}}}$

```

1 def initialize_parameters_he(layers_dims):
2     # layer_dims -- python array (list) containing the size of each layer.
3     np.random.seed(3)
4     parameters = {}
5     L = len(layers_dims) - 1 # integer representing the number of layers
6
7     for l in range(1, L + 1):
8         parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1]) *
9             np.sqrt(2/layers_dims[l-1])
10        parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
11
12    return parameters
13
14 parameters = model(train_X, train_Y, initialization = "he")
15 predictions_train = predict(train_X, train_Y, parameters) # 0.993333
16 predictions_test = predict(test_X, test_Y, parameters) # 0.96
17
18 plt.title("Model with He initialization")
19 axes = plt.gca()
20 axes.set_xlim([-1.5,1.5])
21 axes.set_ylim([-1.5,1.5])
22 plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)

```

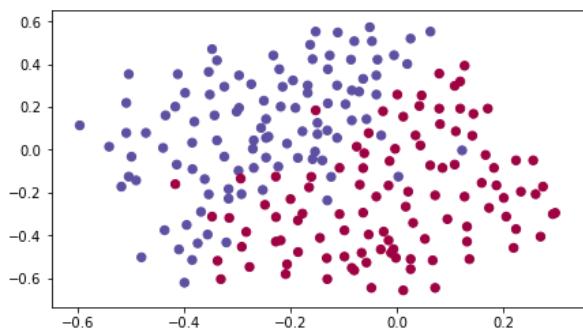
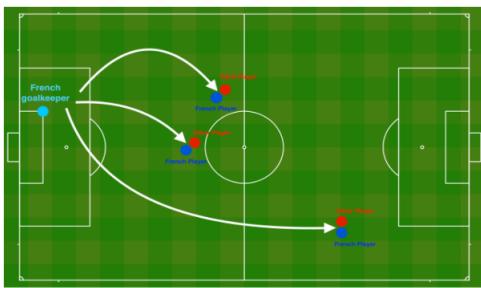


2.1.8 Programming Assignment (Regularization)

We will recommend positions where France's goal keeper should kick the ball so that the French team's players can then hit it with their head. We have data from the past 10 games played by the French team. Each dot corresponds to a position on the football field where a football player has hit the ball with his/her head after the French goal keeper has shot the ball from the left side of the football field:

- If the dot is blue, it means the French player managed to hit the ball with his/her head.
- If the dot is red, it means the other team's player hit the ball with their head.

Our goal is to create a deep learning model to find the positions on the field where the goalkeeper should kick the ball.



The dataset is a little noisy, but from a quick glance we can see the a diagonal line would work decent in separating the two data points.

We will load in the packages that we will need for the project, along with a Neural Network that has already been implemented for us. Note that to use the model in *regularization* mode, we just change the λ value to something besides 0. And to use the model in *dropout* mode, we change the `keep_prob` to a value below 1.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from reg_utils import sigmoid, relu, plot_decision_boundary, initialize_parameters,
4                     load_2D_dataset, predict_dec
5 from reg_utils import compute_cost, predict, forward_propagation,
6                     backward_propagation, update_parameters
7
8 import sklearn
9 import sklearn.datasets
10 import scipy.io
11 from testCases import *
12
13 train_X, train_Y, test_X, test_Y = load_2D_dataset()
14
15 def model(X, Y, learning_rate = 0.3, num_iterations = 30000, print_cost = True,
16           lambd = 0, keep_prob = 1):
17     """
18     Implements a 3-layer neural network: LINEAR->RELU->LINEAR->RELU->LINEAR->SIGMOID.
19
20     Arguments:
21     X -- input data, of shape (input size, number of examples)
22     Y -- true "label" vector (1 for blue dot / 0 for red dot), of shape (output size,m)
23     learning_rate -- learning rate of the optimization
24     num_iterations -- number of iterations of the optimization loop
25     print_cost -- If True, print the cost every 10000 iterations
26     lambd -- regularization hyperparameter, scalar
27     keep_prob - probability of keeping a neuron active during drop-out, scalar.
28
29     Returns:
30     parameters -- parameters learned by the model. They can then be used to predict.
31     """

```

```

1  grads = {}
2  costs = [] # to keep track of the cost
3  m = X.shape[1] # number of examples
4  layers_dims = [X.shape[0], 20, 3, 1]
5
6  parameters = initialize_parameters(layers_dims) # Initialize parameters dictionary.
7
8  # Loop (gradient descent)
9  for i in range(0, num_iterations):
10     # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID.
11     if keep_prob == 1:
12         a3, cache = forward_propagation(X, parameters)
13     elif keep_prob < 1:
14         a3, cache = forward_propagation_with_dropout(X, parameters, keep_prob)
15
16     # Cost function
17     if lambd == 0:
18         cost = compute_cost(a3, Y)
19     else:
20         cost = compute_cost_with_regularization(a3, Y, parameters, lambd)
21
22     # Backward propagation.
23     assert(lambd==0 or keep_prob==1) # only use one or the other
24     if lambd == 0 and keep_prob == 1:
25         grads = backward_propagation(X, Y, cache)
26     elif lambd != 0:
27         grads = backward_propagation_with_regularization(X, Y, cache, lambd)
28     elif keep_prob < 1:
29         grads = backward_propagation_with_dropout(X, Y, cache, keep_prob)
30
31     parameters = update_parameters(parameters, grads, learning_rate) # Update params
32
33     if print_cost and i % 10000 == 0: # Print the loss every 10000 iterations
34         print("Cost after iteration {}: {}".format(i, cost))
35     if print_cost and i % 1000 == 0:
36         costs.append(cost)
37
38 # plot the cost
39 plt.plot(costs)
40 plt.ylabel('cost')
41 plt.xlabel('iterations (x1,000)')
42 plt.title("Learning rate = " + str(learning_rate))
43 plt.show()
44
45 return parameters

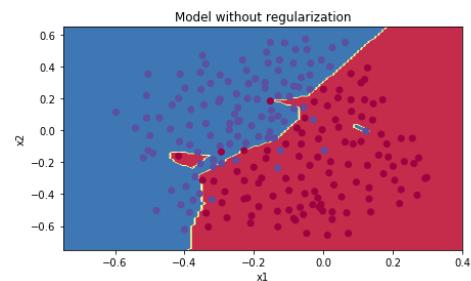
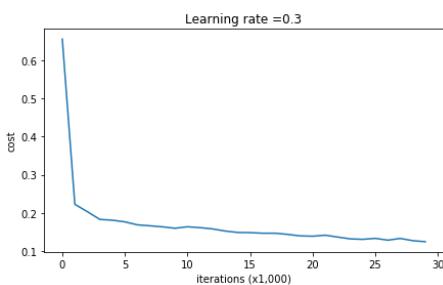
```

We first want to train a model with **no regularization** (baseline model). If you take a look at the decision boundary, you can see the model is overfitting the training set on the noisy points.

```

1  parameters = model(train_X, train_Y)
2  predictions_train = predict(train_X, train_Y, parameters) # 0.94786
3  predictions_test = predict(test_X, test_Y, parameters) # 0.915

```



The standard way to avoid overfitting is called **L2 REGULARIZATION**. It consists of appropriately modifying your cost function, from:

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

To:

$$J_{regularized} = \underbrace{-\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log(a^{[L](i)}) + (1 - y^{(i)}) \log(1 - a^{[L](i)}))}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m} \frac{\lambda}{2} \sum_l \sum_k \sum_j W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

```

1 def compute_cost_with_regularization(A3, Y, parameters, lambd):
2     """
3         Implement the cost function with L2 regularization. See formula above.
4
5     Arguments:
6     A3 -- post-activation, output of forward propagation, of shape (output size, m)
7     Y -- "true" labels vector, of shape (output size, number of examples)
8     parameters -- python dictionary containing parameters of the model
9
10    Returns:
11    cost - value of the regularized loss function (formula (2))
12    """
13
14    m = Y.shape[1]
15    W1 = parameters["W1"]
16    W2 = parameters["W2"]
17    W3 = parameters["W3"]
18
19    cross_entropy_cost = compute_cost(A3, Y) # the cross-entropy part of the cost
20
21    L2_regularization_cost = (lambd/(2*m)) * np.sum(np.sum(np.square(W1)) +
22                                np.sum(np.square(W2)) + np.sum(np.square(W3)))
23
24    cost = cross_entropy_cost + L2_regularization_cost
25
26    return cost

```

Now, since we have changed the value cost, we must also change backward propagation. All the gradients have to be computed with respect to this new cost. For each, you have to add the regularization term's gradient

$$\frac{d}{dW} \left(\frac{1}{2} \frac{\lambda}{m} W^2 \right) = \frac{\lambda}{m} W$$

```

1 def backward_propagation_with_regularization(X, Y, cache, lambd):
2     """
3         Implements the backward propagation to which we added an L2 regularization.
4
5     Arguments:
6     X -- input dataset, of shape (input size, number of examples)
7     Y -- "true" labels vector, of shape (output size, number of examples)
8     cache -- cache output from forward_propagation()
9     lambd -- regularization hyperparameter, scalar
10
11    Returns:
12    gradients -- Dictionary with gradients and parameters
13    """
14
15    m = X.shape[1]
16    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

```

```

1  dZ3 = A3 - Y
2  dW3 = 1./m * np.dot(dZ3, A2.T) + (lambd/m)*W3
3  db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
4
5  dA2 = np.dot(W3.T, dZ3)
6  dZ2 = np.multiply(dA2, np.int64(A2 > 0))
7  dW2 = 1./m * np.dot(dZ2, A1.T) + (lambd/m)*W2
8  db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
9
10 dA1 = np.dot(W2.T, dZ2)
11 dZ1 = np.multiply(dA1, np.int64(A1 > 0))
12 dW1 = 1./m * np.dot(dZ1, X.T) + (lambd/m)*W1
13 db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
14
15 gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3, "dA2": dA2,
16             "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
17             "dZ1": dZ1, "dW1": dW1, "db1": db1}
18
19 return gradients

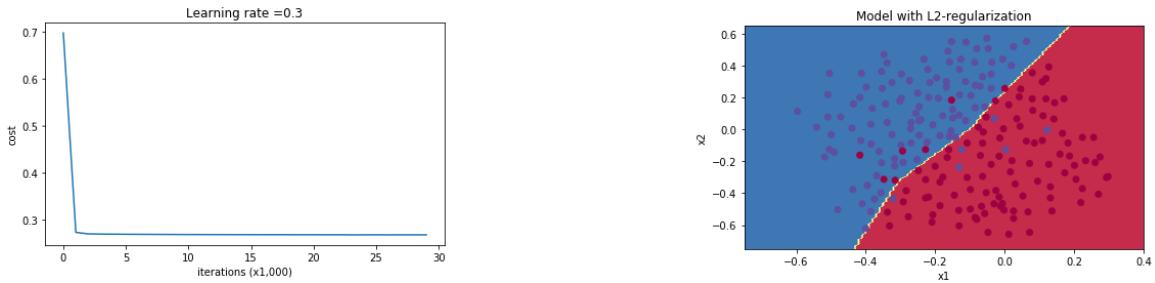
```

Now lets run the model with $\lambda = 0.7$

```

1 parameters = model(train_X, train_Y, lambd = 0.7)
2 predictions_train = predict(train_X, train_Y, parameters) # 0.93838
3 predictions_test = predict(test_X, test_Y, parameters) # 0.93

```



We have not only increased our accuracy on the test set by 1.5%, but we are also not longer overfitting the data. Note that L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to “oversmooth”, resulting in a model with high bias. Three key takeaways from this are:

- The cost computation: A regularization term is added to the cost.
- The backpropagation function: There are extra terms in the gradients with respect to W.
- Weights end up smaller (“weight decay”) by being pushed to smaller values.

Finally, **DROPOUT** is a widely used regularization technique that is specific to deep learning. It randomly shuts down some neurons in each iteration. At each iteration, you shut down (= set to zero) each neuron of a layer with probability $1 - \text{keep_prob}$ or keep it with probability keep_prob . The dropped neurons don’t contribute to the training in both the forward and backward propagations of the iteration.

When you shut some neurons down, you actually modify your model. The idea behind drop-out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.

The first step is to implement *forward propagation* with dropout. You are using a 3 layer neural network, and will add dropout to the first and second hidden layers. We will not apply dropout to the input layer or output layer.

```

1 def forward_propagation_with_dropout(X, parameters, keep_prob = 0.5):
2     """
3         Implements the forward propagation: LINEAR -> RELU + DROPOUT -> LINEAR ->
4             RELU + DROPOUT -> LINEAR -> SIGMOID.
5
6     Arguments:
7         X -- input dataset, of shape (2, number of examples)
8         parameters -- python dictionary containing your parameters
9             W1 -- weight matrix of shape (20, 2)
10            b1 -- bias vector of shape (20, 1)
11            W2 -- weight matrix of shape (3, 20)
12            b2 -- bias vector of shape (3, 1)
13            W3 -- weight matrix of shape (1, 3)
14            b3 -- bias vector of shape (1, 1)
15            keep_prob - probability of keeping a neuron active during drop-out, scalar
16
17     Returns:
18         A3 -- last activation value, output of the forward propagation, of shape (1,1)
19         cache -- tuple, information stored for computing the backward propagation
20     """
21     np.random.seed(1)
22
23     # retrieve parameters
24     W1 = parameters["W1"]
25     b1 = parameters["b1"]
26     W2 = parameters["W2"]
27     b2 = parameters["b2"]
28     W3 = parameters["W3"]
29     b3 = parameters["b3"]
30
31     # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
32     Z1 = np.dot(W1, X) + b1 # linear
33     A1 = relu(Z1) # relu
34     D1 = np.random.rand(A1.shape[0], A1.shape[1]) # Step 1: initialize matrix D1
35     D1 = (D1 < keep_prob).astype(int) # Step 2: convert entries of D1 to 0 or 1
36     A1 = A1 * D1 # Step 3: shut down some neurons of A1
37     A1 = A1/keep_prob # Step 4: scale the value of neurons that haven't been shut down
38
39     Z2 = np.dot(W2, A1) + b2 # linear
40     A2 = relu(Z2) # relu
41     D2 = np.random.rand(A2.shape[0], A2.shape[1]) # Step 1: initialize matrix D2
42     D2 = (D2 < keep_prob).astype(int) # Step 2: convert entries of D2 to 0 or 1
43     A2 = A2 * D2 # Step 3: shut down some neurons of A2
44     A2 = A2/keep_prob # Step 4: scale the value of neurons that haven't been shut down
45
46     Z3 = np.dot(W3, A2) + b3 # linear
47     A3 = sigmoid(Z3) # sigmoid
48
49     cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)
50     return A3, cache

```

The next step is to implement the backward propagation with dropout. As before, we are training a 3 layer network. Add dropout to the first and second hidden layers, using the masks $D^{[1]}$ and $D^{[2]}$ stored in the cache. There are two steps to carry out when doing back propagation:

- 1) We want to shut down the same neurons when performing back propagation that we did during forward propagation.
- 2) Also, we must divide $dA^{[1]}$ by $keep_prob$ in order to scale by the same amount.

```

1 def backward_propagation_with_dropout(X, Y, cache, keep_prob):
2     """
3         Implements the backward propagation of our baseline model to which we added dropout
4
5         Arguments:
6             X -- input dataset, of shape (2, number of examples)
7             Y -- "true" labels vector, of shape (output size, number of examples)
8             cache -- cache output from forward_propagation_with_dropout()
9             keep_prob - probability of keeping a neuron active during drop-out, scalar
10
11        Returns:
12            gradients -- A dictionary with gradients and parameters
13        """
14    m = X.shape[1]
15    (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3) = cache
16
17    dZ3 = A3 - Y
18    dW3 = 1./m * np.dot(dZ3, A2.T)
19    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
20
21    dA2 = np.dot(W3.T, dZ3)
22    dA2 = dA2 * D2 # Step 1: shut down the same neurons as during the forward prop
23    dA2 = dA2 / keep_prob # Step 2: Scale the value of neurons
24
25    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
26    dW2 = 1./m * np.dot(dZ2, A1.T)
27    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
28
29    dA1 = np.dot(W2.T, dZ2)
30    dA1 = dA1 * D1 # Step 1: shut down the same neurons as during the forward prop
31    dA1 = dA1 / keep_prob # Step 2: Scale the value of neurons
32
33    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
34    dW1 = 1./m * np.dot(dZ1, X.T)
35    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
36
37    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3, "dA2": dA2,
38                 "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
39                 "dZ1": dZ1, "dW1": dW1, "db1": db1}
40
41    return gradients

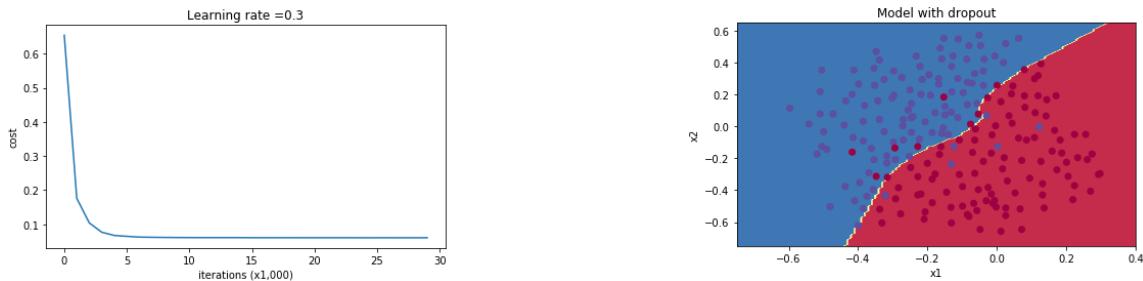
```

Now we can run the model using dropout (with a `keep_prob = 0.86`). It means at every iteration you shut down each neurons of layer 1 and 2 with 14% probability.

```

1 parameters = model(train_X, train_Y, keep_prob = 0.86, learning_rate = 0.3)
2 predictions_train = predict(train_X, train_Y, parameters) # 0.9289
3 predictions_test = predict(test_X, test_Y, parameters) # 0.95

```



As we can see, our model has increased its test accuracy by 5% from the baseline by using dropout. Remember that dropout is only used during training and never during testing. Our deep learning frameworks (such as TensorFlow) come with a dropout layer implementation that we will learn about in the future.

2.1.9 Programming Assignment (Gradient Checking)

```

1 # Packages
2 import numpy as np
3 from testCases import *
4 from gc_utils import sigmoid, relu, dictionary_to_vector, vector_to_dictionary,
5                               gradients_to_vector

```

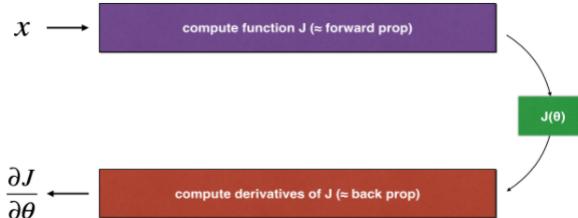
Backpropagation computes the gradients $\frac{\partial J}{\partial \theta}$, where θ denotes the parameters of the model. J is computed using forward propagation and your loss function. If you are confident that your forward propagation is correct and that your computing J correctly, you can use your code for computing J to verify the code for computing $\frac{\partial J}{\partial \theta}$.

Let's look back at the definition of a derivative (or gradient):

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \rightarrow 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

We can use the above equation and a small value for ε to check that your code for computing $\frac{\partial J}{\partial \theta}$ is correct. Remember that we want our error to be equal to or smaller than our ϵ value.

Consider a **1D linear function** $J(\theta) = \theta x$. The model contains only a single real-valued parameter θ , and takes x as input. We will implement code to compute J and its derivative. We will then use gradient checking to make sure our backward propagation is correct.



First start with x , then evaluate the function $J(x)$ (“forward propagation”). Then compute the derivative $\frac{\partial J}{\partial \theta}$ (“backward propagation”).

```

1 def forward_propagation(x, theta):
2     """
3         Implement the linear forward propagation (J = theta * x)
4     """
5     J = np.multiply(theta, x)
6     return J
7
8 def backward_propagation(x, theta):
9     """
10        Computes the derivative of J with respect to theta (dtheta = x).
11    """
12    dtheta = x
13    return dtheta

```

Now we implement gradient checking to see if our back propagation function is working. We first compute $gradapprox$ by using the formula above and a small value for ϵ . We follow these steps:

1. $\theta^+ = \theta + \varepsilon$
2. $\theta^- = \theta - \varepsilon$
3. $J^+ = J(\theta^+)$
4. $J^- = J(\theta^-)$
5. $gradapprox = \frac{J^+ - J^-}{2\varepsilon}$

We then compute the gradient using backwards propagation and store it in variable *grad*. Finally, we compute the difference with the following formula:

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2}$$

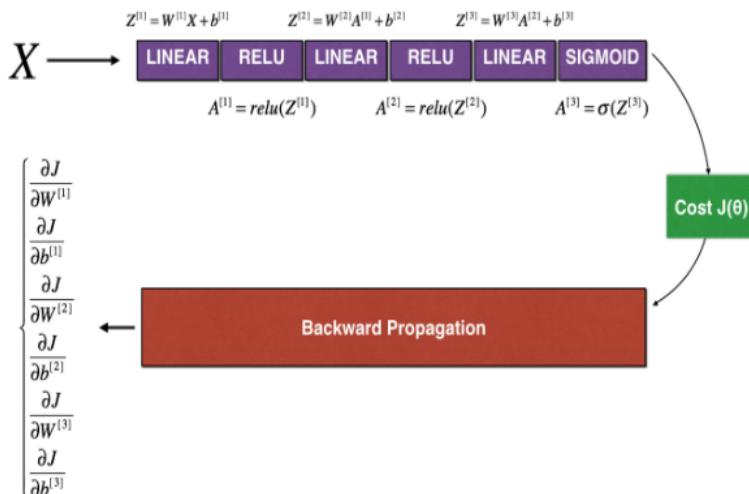
Note that If this difference is small (less than ϵ), you can be quite confident that you have computed your gradient correctly. Otherwise, there may be a mistake in the gradient computation.

```

1 def gradient_check(x, theta, epsilon = 1e-7):
2     """
3         Implement gradient checking for backward propagation.
4
5     Arguments:
6         x -- a real-valued input
7         theta -- our parameter, a real number as well
8         epsilon -- tiny shift to the input to compute approximated gradient with
9
10    Returns:
11        difference -- difference between the approx gradient and the back prop gradient
12    """
13
14    # Compute gradapprox
15    thetaplus = theta + epsilon # Step 1
16    thetaminus = theta - epsilon # Step 2
17    J_plus = forward_propagation(x, thetaplus) # Step 3
18    J_minus = forward_propagation(x, thetaminus) # Step 4
19    gradapprox = (J_plus - J_minus) / (2.*epsilon) # Step 5
20
21    grad = backward_propagation(x, theta) # Compute gradient
22
23    numerator = np.linalg.norm(grad - gradapprox)
24    denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
25    difference = numerator/denominator
26
27    if difference < 1e-7: # epsilon value
28        print ("The gradient is correct!")
29    else:
30        print ("The gradient is wrong!")
31
32    return difference

```

Now, we want to see how to implement **N-Dimensional** gradient checking. The following picture shows the forward and back propagation steps our model takes.



Our forward and back propagation are already built for us, so lets take a look:

```
1 def forward_propagation_n(X, Y, parameters):
2     """
3         Arguments:
4             X -- training set for m examples
5             Y -- labels for m examples
6             parameters -- python dictionary containing your parameters
7             """
8
9     # retrieve parameters
10    m = X.shape[1]
11    W1 = parameters["W1"]
12    b1 = parameters["b1"]
13    W2 = parameters["W2"]
14    b2 = parameters["b2"]
15    W3 = parameters["W3"]
16    b3 = parameters["b3"]
17
18    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
19    Z1 = np.dot(W1, X) + b1
20    A1 = relu(Z1)
21    Z2 = np.dot(W2, A1) + b2
22    A2 = relu(Z2)
23    Z3 = np.dot(W3, A2) + b3
24    A3 = sigmoid(Z3)
25
26    # Cost
27    logprobs = np.multiply(-np.log(A3), Y) + np.multiply(-np.log(1 - A3), 1 - Y)
28    cost = 1./m * np.sum(logprobs)
29
30    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3)
31    return cost, cache
32
33 def backward_propagation_n(X, Y, cache):
34     """
35         Arguments:
36             X -- input datapoint, of shape (input size, 1)
37             Y -- true "label"
38             cache -- cache output from forward_propagation_n()
39             """
40
41    m = X.shape[1]
42    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache
43
44    dZ3 = A3 - Y
45    dW3 = 1./m * np.dot(dZ3, A2.T)
46    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
47
48    dA2 = np.dot(W3.T, dZ3)
49    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
50    dW2 = 1./m * np.dot(dZ2, A1.T)
51    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)
52
53    dA1 = np.dot(W2.T, dZ2)
54    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
55    dW1 = 1./m * np.dot(dZ1, X.T)
56    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
57
58    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
59                 "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
60                 "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}
61
62    return gradients
```

We will use the same formula from above in order to find $\frac{\partial J}{\partial \theta}$. However, θ is not a scalar anymore. It is a dictionary called “parameters”. We implemented a function “`dictionary_to_vector()`” for you. It converts the “parameters” dictionary into a vector called “values”, obtained by reshaping all parameters ($W_1, b_1, W_2, b_2, W_3, b_3$) into vectors and concatenating them. The inverse function is “`vector_to_dictionary`” which outputs back the “parameters” dictionary.

```

1  def gradient_check_n(parameters, gradients, X, Y, epsilon = 1e-7):
2      """
3          Arguments:
4              parameters -- python dictionary containing your parameters
5              grad -- output of backward_propagation_n, contains gradients
6              x -- input datapoint, of shape (input size, 1)
7              y -- true "label"
8              epsilon -- tiny shift to the input to compute approximated gradient with
9      """
10     # Set-up variables
11     parameters_values, _ = dictionary_to_vector(parameters)
12     grad = gradients_to_vector(gradients)
13     num_parameters = parameters_values.shape[0]
14     J_plus = np.zeros((num_parameters, 1))
15     J_minus = np.zeros((num_parameters, 1))
16     gradapprox = np.zeros((num_parameters, 1))
17
18     # Compute gradapprox
19     for i in range(num_parameters):
20
21         # Compute J_plus[i]
22         thetaplus = np.copy(parameters_values)
23         thetaplus[i][0] = thetaplus[i][0] + epsilon
24         J_plus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaplus))
25
26
27         # Compute J_minus[i]
28         thetaminus = np.copy(parameters_values)
29         thetaminus[i][0] = thetaminus[i][0] - epsilon
30         J_minus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaminus))
31
32         # Compute gradapprox[i]
33         gradapprox[i] = (J_plus[i] - J_minus[i]) / (2.*epsilon)
34
35     # Compare gradapprox to backward propagation gradients by computing difference.
36     numerator = np.linalg.norm(grad-gradapprox)
37     denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
38     difference = numerator/denominator
39
40     if difference > 2e-7:
41         print ("There is a mistake in the backward propagation! difference = " +
42               str(difference))
43     else:
44         print ("Your backward propagation works perfectly fine! difference = " +
45               str(difference))
46
47     return difference

```

A few takeaways from this assignment:

- Gradient Checking is slow! Approximating the gradient with $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ is computationally costly. For this reason, we don’t run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.
- The gradient checking we implemented doesn’t work with dropout. You would usually run the gradient check algorithm without dropout to make sure your backprop is correct, then add dropout.

2.2 Optimization Algorithms

2.2.1 Mini-Batch Gradient Descent

We will eventually start to train with datasets that have millions of training examples in them. In order to quickly train models, we want to split both the x and y datasets into **mini-batches**, in which we split our training data and their corresponding labels into equally small sizes.

Note that the notation to denote a mini batch will be $x^{\{t\}}$ and $y^{\{t\}}$, with shape $(n_x, \text{batch size})$.

To perform gradient descent using a mini batch, where our batch size (m) will be 1,000. Mini-batch gradient descent performs much faster than a full batch and is the preferred method. The following represents 1 *epoch* (one pass through the training set). Note that this is only one pass through the training set, we may want to put a loop around the for loop below in order to take multiple steps of gradient descent.

for t=1, ..., num_of_batches

```

    Forward propagation on  $x^{\{t\}}$  (vectorized implementation)
    Compute cost  $J^{\{t\}} = \frac{1}{m} \sum_{i=1}^l L(\hat{y}^{(i)}, y^{(i)}) + \text{regularization term}$ 
    Compute backpropagation for gradients using  $(x^{\{t\}}, y^{\{t\}})$ .
    Update parameters  $W^{[l]}$  and  $B^{[l]}$ 
```

When training on mini-batches, if you plot $J^{\{t\}}$ it will be more noisy (not a smooth line) but still trends downwards. You also must choose a batch size that isn't too big or too small. If it is too big then it will take too long per iteration, but if it is too small then you lose speed from vectorization. Some guidelines to **choosing a batch size**:

- If training set is under 2000, use batch gradient descent.
- Typical mini-batch sizes are: 64, 128, 256, and 512.
- Make sure that $(x^{\{t\}}, y^{\{t\}})$ fits in your CPU or GPU memory.

2.2.2 Exponentially Weighted Averages

There are algorithms that are faster than gradient descent, but in order to understand them we need to be able to use **exponentially weighted averages**. This will be a key component for our optimization algorithms, and we can use the following general formula:

$$V_t = \beta V_{t-1} + (1 - \beta)\theta_t$$

Note that the larger the β value is, the more weight you are giving to the previous observation and the smoother your line will be (but it may also be slightly shifted). The smaller the β value becomes, the less weight the previous observation has and the more noisy your line will be.

We can use **bias correction** in order to compute these averages more accurately. To do this, we divide our values by $(1-\beta^t)$. This gives us the general formula:

$$\frac{V_t}{1 - \beta^t} = \frac{\beta V_{t-1} + (1 - \beta)\theta_t}{1 - \beta^t}$$

We use this mainly in our “initial” phase, where this new formula helps mostly in the beginning of our weighted average computations where t is small. But as t grows larger, it will almost zero out the β value and be similar to just dividing both sides by 1.

2.2.3 Gradient Descent with Momentum

In general, the **gradient descent with momentum** algorithm almost always works faster than standard gradient descent. The basic idea is to compute an exponentially weighted average of your gradients, and then use that gradient to update your weights instead.

In general, we want our learning on the vertical axis to be slower in order to avoid oscillation (which leads to divergence in some cases). We also want the horizontal axis to have faster learning so it moves towards to minimum.

To implement momentum on a given iteration t :

- Compute dW , db on current mini-batch.
- Compute $V_{dw} = \beta V_{dw} + (1 - \beta)dW$
- Compute $V_{db} = \beta V_{db} + (1 - \beta)db$
- Update parameters with: $W = W - \alpha V_{dw}$
 $b = b - \alpha V_{db}$

We now have the hyperparameters α and β . Note that the most common value for $\beta = 0.9$ (where we average over the last 10 gradients). We also initialize $V_{dw} = 0$ and $V_{db} = 0$.

2.2.4 RMSprop

RMSprop, known as Root Mean Square prop, is another algorithm that can speed up gradient descent. The general idea is that RMSprop reduces the noise on the vertical axis when performing gradient descent, leading to a straighter line to the minimum value we want.

To implement RMSprop on a given iteration t :

- Compute dW , db on current mini-batch.
- Compute $S_{dw} = \beta_2 S_{dw} + (1 - \beta_2)dW^2$ (note dW^2 is element wise square).
- Compute $S_{db} = \beta_2 S_{db} + (1 - \beta_2)db^2$ (note db^2 is element wise square).
- Update parameters with: $W = W - \alpha \frac{dW}{\sqrt{S_{dw} + \epsilon}}$
 $b = b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}}$

Note that the ϵ that is added to the bottom of the fraction for updating the parameters is just a very small value to ensure that we do not divide by 0 when implementing the algorithm in code.

2.2.5 Adam

The **Adam** (adaptive moment estimation) optimization algorithm takes the momentum and RMSprop algorithm and combines them. When implementing this, we initialize $V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, and $S_{db} = 0$. Another important note is that the Adam optimizer uses bias correction. So on a given iteration t :

- Compute dW , db on current mini-batch.
- Compute V_{dw} and V_{db} from *momentum* algorithm.
- Compute S_{dw} and S_{db} from *RMSprop* algorithm.
- Bias Correction: $V_{dw}^{corrected} = \frac{V_{dw}}{(1 - \beta_1^t)}$, $V_{db}^{corrected} = \frac{V_{db}}{(1 - \beta_1^t)}$
 $S_{dw}^{corrected} = \frac{S_{dw}}{(1 - \beta_2^t)}$, $S_{db}^{corrected} = \frac{S_{db}}{(1 - \beta_2^t)}$
- Update parameters with: $W = W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}}$
 $b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$

Common hyperparameter choices (recommended by creators of Adam):

- α : needs to be tuned
- β_1 : 0.9
- β_2 : 0.999
- ϵ : 10^{-8}

2.2.6 Learning Rate Decay

One thing that might help to speed up your algorithm is to slowly reduce your learning rate over time, known as **learning rate decay**. This is useful because as learning converges, reducing the rate can help us take smaller steps and get closer to the minimum value we want.

Remember that 1 epoch = 1 pass through the dataset, so we can implement learning rate decay by:

$$\alpha = \frac{1}{1 + \text{decay_rate} * \text{num_epoch}} \alpha_0$$

2.2.7 Programming Assignment (Optimization)

Until now, we've always used Gradient Descent to update the parameters and minimize the cost. In this assignment, you will learn more advanced optimization methods that can speed up learning and perhaps even get you to a better final value for the cost function. Having a good optimization algorithm can be the difference between waiting days vs. just a few hours to get a good result.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.io
4 import math
5 import sklearn
6 import sklearn.datasets
7
8 from opt_utils_v1a import load_params_and_grads, initialize_parameters,
9         forward_propagation, backward_propagation
10 from opt_utils_v1a import compute_cost, predict, predict_dec, plot_decision_boundary,
11      load_dataset
12 from testCases import *
```

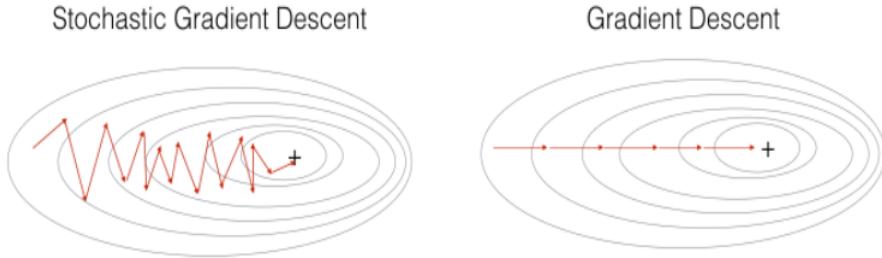
We will first implement **batch gradient descent parameter updating**, which is what we have been using previously. Recall that rule is for layers 1 to L:

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$

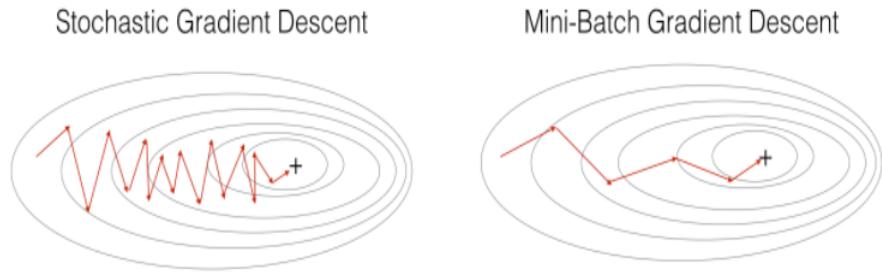
$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

```
1 def update_parameters_with_gd(parameters, grads, learning_rate):
2     """
3     Update parameters using one step of gradient descent
4
5     Arguments:
6     parameters -- python dictionary containing your parameters
7     grads -- python dictionary containing your gradients
8     learning_rate -- the learning rate, scalar.
9
10    Returns:
11    parameters -- python dictionary containing your updated parameters
12    """
13    L = len(parameters) // 2 # number of layers in the neural networks
14
15    # Update rule for each parameter
16    for l in range(L):
17        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate *
18                           grads["dW" + str(l+1)]
19        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate *
20                           grads["db" + str(l+1)]
21
22    return parameters
```

One variant of this is **Stochastic Gradient Descent** (SGD), which is equivalent to mini-batch gradient descent where each mini-batch has just 1 example. The update rule above does not change, but what does change is that we compute the gradient on just one example at a time and then updating the parameters. When the training set is large, SGD can be faster. But the parameters will “oscillate” toward the minimum rather than converge smoothly.



Although SGD can be more efficient than batch gradient descent, you’ll get better results if you use a batch-size in between the two. We call this **mini-batch gradient descent**, for which we loop over small batches of data and then update the parameters. An important note is that with all three of these, you will have to tune α .



There are two steps in **building mini-batches** from the training set (X, Y) :

- 1) **Shuffle** - Create a shuffled version of the training set (X, Y) where each column of X and Y represents a training example. Note that this is done synchronously between X and Y such that after the shuffling, the i^{th} column of X is the example corresponding to the i^{th} label in Y . The shuffling step ensures that examples will be split randomly into different mini-batches.
- 2) **Partition** - Partition the shuffled (X, Y) into mini-batches of size $mini_batch_size$. Note that the number of training examples is not always divisible by $mini_batch_size$. The last mini batch might be smaller, but you don’t need to worry about this.

Note that if the last mini-batch is smaller than the given batch size (in our case we will use 64), then we will round down the number of full mini batches to the nearest integer using `math.floor()`. This means for the total full mini-batches, we use:

$$math.floor\left(\frac{m}{mini_batch_size}\right)$$

This means that the final mini-batch, if not a full size, will be found by using:

$$m - mini_batch_size * math.floor\left(\frac{m}{mini_batch_size}\right)$$

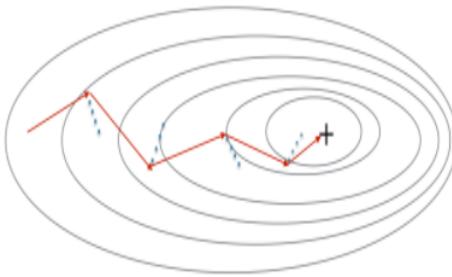
We will implement this idea in Python code, where we handle an end case when the last batch is not a full size.

```

1 def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
2     """
3         Creates a list of random minibatches from (X, Y)
4
5         Arguments:
6             X -- input data, of shape (input size, m)
7             Y -- true "label" vector (1 for blue dot / 0 for red dot), of shape (1, m)
8             mini_batch_size -- size of the mini-batches, integer
9
10        Returns:
11            mini_batches -- list of synchronous (mini_batch_X, mini_batch_Y)
12        """
13        np.random.seed(seed) # To make your "random" minibatches the same as ours
14        m = X.shape[1] # number of training examples
15        mini_batches = []
16
17        # Step 1: Shuffle (X, Y)
18        permutation = list(np.random.permutation(m))
19        shuffled_X = X[:, permutation]
20        shuffled_Y = Y[:, permutation].reshape((1,m))
21
22        # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
23        num_complete_minibatches = math.floor(m/mini_batch_size) # round down
24        for k in range(0, num_complete_minibatches):
25            mini_batch_X = shuffled_X[:, k*mini_batch_size: (k+1)*mini_batch_size]
26            mini_batch_Y = shuffled_Y[:, k*mini_batch_size: (k+1)*mini_batch_size]
27            mini_batch = (mini_batch_X, mini_batch_Y)
28            mini_batches.append(mini_batch)
29
30        # Handling the end case (last mini-batch < mini_batch_size)
31        if m % mini_batch_size != 0:
32            mini_batch_X = shuffled_X[:, -(m-mini_batch_size*num_complete_minibatches):m]
33            mini_batch_Y = shuffled_Y[:, -(m-mini_batch_size*num_complete_minibatches):m]
34            mini_batch = (mini_batch_X, mini_batch_Y)
35            mini_batches.append(mini_batch)
36
37    return mini_batches

```

Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will “oscillate” toward convergence. Using momentum can reduce these oscillations. **Momentum** takes into account the past gradients to smooth out the update. We will store the ‘direction’ of the previous gradients in the variable v . Formally, this will be the exponentially weighted average of the gradient on previous steps.



We will first **initialize the velocity**. It needs to be an array of zeros that are the same size as the parameters they will be applied on. Because they are zeros, the algorithm will take a few iterations to “build up” velocity and start to take bigger steps.

```

1 def initialize_velocity(parameters):
2     """
3         Initializes the velocity as a python dictionary
4
5     Arguments:
6         parameters -- python dictionary containing your parameters.
7
8     Returns:
9         v -- python dictionary containing the velocity.
10
11    """
12
13    L = len(parameters) // 2 # number of layers in the neural networks
14    v = {}
15
16    # Initialize velocity
17    for l in range(L):
18        v["dW" + str(l+1)] = np.zeros((parameters["W" + str(l+1)].shape))
19        v["db" + str(l+1)] = np.zeros((parameters["b" + str(l+1)].shape))
20
21    return v

```

Now that we have initialized our velocity, we want to implement the **parameters update with momentum**. The update rule is that for layers t to L:

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1 - \beta)dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1 - \beta)db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

where L is the number of layers, β is the momentum and α is the learning rate. A common value for β is between 0.8 to 0.999 (default is 0.9 if you don't feel like tuning it).

```

1 def update_parameters_with_momentum(parameters, grads, v, beta, learning_rate):
2     """
3         Update parameters using Momentum
4
5     Arguments:
6         parameters -- python dictionary containing your parameters:
7             grads -- python dictionary containing your gradients for each parameters:
8             v -- python dictionary containing the current velocity:
9             beta -- the momentum hyperparameter, scalar
10            learning_rate -- the learning rate, scalar
11
12     Returns:
13         parameters -- python dictionary containing your updated parameters
14         v -- python dictionary containing your updated velocities
15     """
16
17     L = len(parameters) // 2 # number of layers in the neural networks
18
19     # Momentum update for each parameter
20     for l in range(L):
21         v["dW" + str(l+1)] = beta*v["dW" + str(l+1)] + (1-beta)*grads["dW" + str(l+1)]
22         v["db" + str(l+1)] = beta*v["db" + str(l+1)] + (1-beta)*grads["db" + str(l+1)]
23         parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
24                                     learning_rate*v["dW" + str(l+1)]
25         parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
26                                     learning_rate*v["db" + str(l+1)]
27
28     return parameters, v

```

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSprop and Momentum. The update rule for this is that for layers 1 to L:

$$\begin{cases} v_{dW[l]} = \beta_1 v_{dW[l]} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W[l]} \\ v_{dW[l]}^{corrected} = \frac{v_{dW[l]}}{1 - (\beta_1)^t} \\ s_{dW[l]} = \beta_2 s_{dW[l]} + (1 - \beta_2) \left(\frac{\partial \mathcal{J}}{\partial W[l]} \right)^2 \\ s_{dW[l]}^{corrected} = \frac{s_{dW[l]}}{1 - (\beta_2)^t} \\ W[l] = W[l] - \alpha \frac{v_{dW[l]}^{corrected}}{\sqrt{s_{dW[l]}^{corrected}} + \epsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ϵ is a very small number to avoid dividing by zero

We will first **initialize the Adam parameters**, which need to be arrays of zeros.

```

1 def initialize_adam(parameters) :
2     """
3         Initializes v and s as two python dictionaries
4
5         Arguments:
6             parameters -- python dictionary containing your parameters.
7
8         Returns:
9             v -- dict that contains the exponentially weighted average of the gradient.
10            s -- dict that contains the exponentially weighted average of the squared gradient.
11            """
12
13     L = len(parameters) // 2 # number of layers in the neural networks
14     v = {}
15     s = {}
16
17     # Initialize v, s.
18     for l in range(L):
19         v["dW" + str(l+1)] = np.zeros((parameters["W" + str(l+1)].shape))
20         v["db" + str(l+1)] = np.zeros((parameters["b" + str(l+1)].shape))
21         s["dW" + str(l+1)] = np.zeros((parameters["W" + str(l+1)].shape))
22         s["db" + str(l+1)] = np.zeros((parameters["b" + str(l+1)].shape))
23
24     return v, s

```

Now we want to implement **parameter updating with Adam**.

```

1 def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate = 0.01,
2                                     beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8):
3
4     """
5         Update parameters using Adam
6
7         Arguments:
8             parameters -- python dictionary containing your parameters:
9             grads -- python dictionary containing your gradients for each parameters:
10            v -- Adam variable, moving average of the first gradient, python dictionary
11            s -- Adam variable, moving average of the squared gradient, python dictionary
12            learning_rate -- the learning rate, scalar.
13            beta1 -- Exponential decay hyperparameter for the first moment estimates
14            beta2 -- Exponential decay hyperparameter for the second moment estimates
15            epsilon -- hyperparameter preventing division by zero in Adam updates

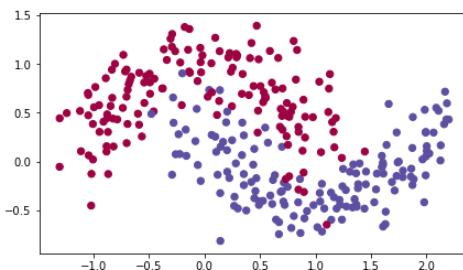
```

```

1 """
2 Returns:
3     parameters -- python dictionary containing your updated parameters
4     v -- Adam variable, moving average of the first gradient, python dictionary
5     s -- Adam variable, moving average of the squared gradient, python dictionary
6 """
7 L = len(parameters) // 2 # number of layers in the neural networks
8 v_corrected = {} # Initializing first moment estimate
9 s_corrected = {} # Initializing second moment estimate
10
11 # Perform Adam update on all parameters
12 for l in range(L):
13     # Moving average of the gradients
14     v["dW" + str(l+1)] = beta1*v["dW" + str(l+1)] + (1-beta1)*grads["dW" + str(l+1)]
15     v["db" + str(l+1)] = beta1*v["db" + str(l+1)] + (1-beta1)*grads["db" + str(l+1)]
16
17     # Compute bias-corrected first moment estimate
18     v_corrected["dW" + str(l+1)] = v["dW" + str(l+1)] / (1-np.power(beta1,t))
19     v_corrected["db" + str(l+1)] = v["db" + str(l+1)] / (1-np.power(beta1,t))
20
21     # Moving average of the squared gradients
22     s["dW" + str(l+1)] = beta2*s["dW" + str(l+1)] + (1-beta2)*np.power(grads["dW" +
23                                         str(l+1)], 2)
24     s["db" + str(l+1)] = beta2*s["db" + str(l+1)] + (1-beta2)*np.power(grads["db" +
25                                         str(l+1)], 2)
26
27     # Compute bias-corrected second moment estimate
28     s_corrected["dW" + str(l+1)] = s["dW" + str(l+1)] / (1-np.power(beta2,t))
29     s_corrected["db" + str(l+1)] = s["db" + str(l+1)] / (1-np.power(beta2,t))
30
31     # Update parameters
32     parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate *
33                                     v_corrected["dW" + str(l+1)] /
34                                     np.sqrt(s_corrected["dW" + str(l+1)] + epsilon)
35     parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate *
36                                     v_corrected["db" + str(l+1)] /
37                                     np.sqrt(s_corrected["db" + str(l+1)] + epsilon)
38
39 return parameters, v, s

```

We now have three working optimization algorithms (mini-batch gradient descent, Momentum, Adam). Let's **implement a model** with each of these optimizers and observe the difference. Lets use the following "moons" dataset to test the different optimization methods.



We have already implemented a 3-layer neural network. You will train it with:

- *Mini-batch Gradient Descent*: it will call your function:
 - `update_parameters_with_gd()`
- *Mini-batch Momentum*: it will call your functions:
 - `initialize_velocity()` and `update_parameters_with_momentum()`
- *Mini-batch Adam*: it will call your functions:
 - `initialize_adam()` and `update_parameters_with_adam()`

```

1 def model(X, Y, layers_dims, optimizer, learning_rate = 0.0007, mini_batch_size = 64,
2         beta = 0.9, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8,
3         num_epochs = 10000, print_cost = True):
4     """
5         3-layer neural network model which can be run in different optimizer modes.
6
7     Arguments:
8         X -- input data, of shape (2, m)
9         Y -- true "label" vector (1 for blue dot / 0 for red dot), of shape (1, m)
10        layers_dims -- python list, containing the size of each layer
11        learning_rate -- the learning rate, scalar.
12        mini_batch_size -- the size of a mini batch
13        beta -- Momentum hyperparameter
14        beta1 -- Exponential decay hyperparameter for the past gradients estimates
15        beta2 -- Exponential decay hyperparameter for the past squared gradients estimates
16        epsilon -- hyperparameter preventing division by zero in Adam updates
17        num_epochs -- number of epochs
18        print_cost -- True to print the cost every 1000 epochs
19
20    Returns:
21        parameters -- python dictionary containing your updated parameters
22    """
23    L = len(layers_dims) # number of layers in the neural networks
24    costs = [] # to keep track of the cost
25    t = 0 # initializing the counter required for Adam update
26    seed = 10
27    m = X.shape[1] # number of training examples
28
29    parameters = initialize_parameters(layers_dims) # Initialize parameters
30
31    # Initialize the optimizer
32    if optimizer == "gd":
33        pass # no initialization required for gradient descent
34    elif optimizer == "momentum":
35        v = initialize_velocity(parameters)
36    elif optimizer == "adam":
37        v, s = initialize_adam(parameters)
38
39    # Optimization loop
40    for i in range(num_epochs):
41        # Define the random minibatches.
42        seed = seed + 1 # Increment seed to reshuffle differently after each epoch
43        minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
44        cost_total = 0
45
46        for minibatch in minibatches:
47            (minibatch_X, minibatch_Y) = minibatch # Select a minibatch
48
49            # Forward propagation
50            a3, caches = forward_propagation(minibatch_X, parameters)
51
52            # Compute cost and add to the cost total
53            cost_total += compute_cost(a3, minibatch_Y)
54
55            # Backward propagation
56            grads = backward_propagation(minibatch_X, minibatch_Y, caches)
57
58            # Update parameters
59            if optimizer == "gd":
60                parameters = update_parameters_with_gd(parameters, grads, learning_rate)

```

```

1    elif optimizer == "momentum":
2        parameters, v = update_parameters_with_momentum(parameters, grads, v, beta,
3                                                        learning_rate)
4
5    elif optimizer == "adam":
6        t = t + 1 # Adam counter
7        parameters, v, s = update_parameters_with_adam(parameters, grads, v, s, t,
8                                                        learning_rate, beta1, beta2,
9                                                        epsilon)
10
11    cost_avg = cost_total / m
12
13    # Print the cost every 1000 epoch
14    if print_cost and i % 1000 == 0:
15        print ("Cost after epoch %i: %f" %(i, cost_avg))
16    if print_cost and i % 100 == 0:
17        costs.append(cost_avg)
18
19    # plot the cost
20    plt.plot(costs)
21    plt.ylabel('cost')
22    plt.xlabel('epochs (per 100)')
23    plt.title("Learning rate = " + str(learning_rate))
24    plt.show()
25
26
27    return parameters

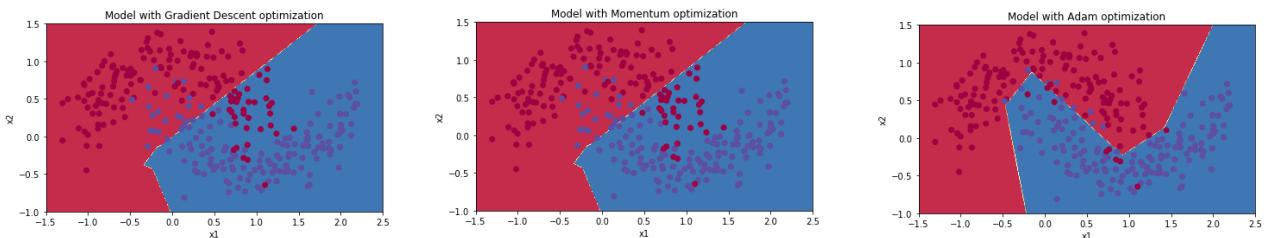
```

We can now build a model with each of these optimizers.

```

1 # Mini-batch Gradient Descent
2 layers_dims = [train_X.shape[0], 5, 2, 1]
3 parameters = model(train_X, train_Y, layers_dims, optimizer = "gd")
4 predictions1 = predict(train_X, train_Y, parameters)
5
6 # Mini-batch Gradient Descent with Momentum
7 layers_dims = [train_X.shape[0], 5, 2, 1]
8 parameters = model(train_X, train_Y, layers_dims, beta = 0.9, optimizer = "momentum")
9 predictions2 = predict(train_X, train_Y, parameters)
10
11 # Mini-batch with Adam
12 layers_dims = [train_X.shape[0], 5, 2, 1]
13 parameters = model(train_X, train_Y, layers_dims, optimizer = "adam")
14 predictions3 = predict(train_X, train_Y, parameters)

```



The accuracies for the optimizers are as follows: Gradient Descent (79.7%), Momentum (79.7%), and Adam (94%).

Momentum usually helps, but given the small learning rate and the simplistic dataset, its impact is almost negligible. Also, the huge oscillations you see in the cost come from the fact that some mini-batches are more difficult than others for the optimization algorithm.

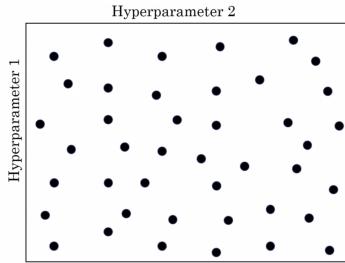
Adam on the other hand, clearly outperforms mini-batch gradient descent and Momentum. If you run the model for more epochs on this simple dataset, all three methods will lead to very good results. However, you've seen that Adam converges a lot faster.

2.3 Hyperparameter Tuning, Batch Normalization, and Frameworks

2.3.1 Tuning process

We have many options for **hyperparameter tuning** that can help to improve our model. Some of these could include α , β , n_h , mini-batch size, and others. Note that the most important hyperparameter to tune is usually α .

When we are trying values, we want to make sure we are randomly selecting these hyperparameters value and not using a grid so that we can use more distinct value for each of the hyperparameters. Below is an example of how we would want to chose our values for two hyperparameters:



Another method is called **coarse to fine**, in which we find a group of values that work best with our model. From there, we “zoom in” and create a box around the values that work best, and try more values in the neighboring range.

It is important to pick the appropriate **scale** on which to explore the hyperparameters. One example is choosing a value for α where we know that it should be between 0.0001 and 1. We want to sample uniformly, but we also don’t want a majority of the samples to come from one side of the value range. To avoid this, we can use **log scaling**, for which we do the following:

- Compute $a = \log_{10}(\min \alpha \text{ value})$
- Compute $b = \log_{10}(\max \alpha \text{ value})$
- We now want to sample from 10^a to 10^b
- Set $r = \text{random value in range } [a, b]$
- Set $\alpha = 10^r$

Another tricky problem is tuning the value for β used in exponentially weighted averages. Let say we expect β to be between 0.9 and 0.999, then we can say that $1-\beta$ is in the range 0.1 to 0.001. We then do the same steps as above, but set $\beta = 1 - 10^r$.

There are two main ways that people perform hyperparameter tuning:

- **Babysitting One Model** : This is for when you don’t have many computational resources (CPUs or GPUs) available to you. For training over multiple days, at the end of each day you change certain hyperparameters in our to decrease the cost function.
- **Training Parallel Models** : This is when your train multiple models on the same problem, but with different hyperparameters. You then compare the results from each of them and pick the hyperparameters that performed best. This will be computationally expensive.

2.3.2 Batch Normalization

We have seen previously that when we normalize our input features (X), our gradient descent works much smoother and our model is able to train faster. The question now is can we **normalize the activation** matrices in our deep Neural Network for each hidden layer (meaning normalize the input of $a^{[l-1]}$ for hidden layer l)? We will do this by normalizing $Z^{[l-1]}$.

In order to **implement Batch Norm** for a given layer l :

- Given some intermediate values in the NN such as $z^{(1)}, \dots, z^{(m)}$

$$\begin{aligned}\mu &= \frac{1}{m} \sum_i z^{(i)} \\ \sigma^2 &= \frac{1}{m} \sum_i (z^{(i)} - \mu)^2 \\ z_{norm}^{(i)} &= \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{z}^{(i)} &= \gamma z_{norm}^{(i)} + \beta\end{aligned}$$

We don't always want the hidden units to have a mean = 0 and standard variance = 1. To avoid this, we compute \tilde{z} , where γ and β are learnable parameters of the model than can be updated. Note these parameters just let us set what we want the mean to be whatever we want it to be. For the later computations in our neural network, we now use $\tilde{z}^{(i)}$ instead of $z^{(i)}$ for a given layer l .

When implementing batch norm with mini-batches, you use only the mini-batch data to calculate the batch norm parameters. Another important note is that when calculating $z^{[l]}$, we now drop the $b^{[l]}$ parameter and replace it with $\beta^{[l]}$. We do this because we want to normalize the data without increasing it by a value b , and after we normalize we then add in β . Also, the shape of both $\beta^{[l]}$ and $\gamma^{[l]}$ will be $(n^{[l]}, 1)$.

To implement **gradient descent with batch norm for mini-batches**, we create a for loop around the following steps for $t=1$ to `num_of_mini_batches`:

- Compute forward propagation on $X^{\{t\}}$
 - In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
- Use back propagation to compute $dW^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$ (remember we drop b).
- Update parameters $W^{[l]}$, $\beta^{[l]}$, and $\gamma^{[l]}$ as usual.

Note that in programming frameworks, you can usually implement batch norm with one line of code.

Batch norm processes our data one mini-batch at a time, but at **test time** we might need to process just one example at a time. Note that in the above equation, μ and σ^2 are computed for each mini-batch. However, during test time we need a different way to compute these values. We can do this by estimating using *exponentially weighted averages across all mini-batches*.

To explain further, we keep a running average of our μ and σ^2 values. When it is test time, we use these averages to compute the z_{norm} , which we then use to compute \tilde{z} using our learned values for γ and β :

$$\begin{aligned}z_{norm} &= \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{z} &= \gamma z_{norm} + \beta\end{aligned}$$

2.3.3 Multi-Class Classification (Softmax Regression)

Softmax Regression is a generalized version of Logistic Regression that allows use to classify C classes instead of just a binary classification which we have previously focused on. The output layer for our Neural Network will have $n^{[L]} = C$, where it matches the number of different classes we are trying to classify. This also means that \hat{y} will have shape $(n^{[L]}, 1)$.

This model uses a **Softmax layer** for the output layer in order to generate these outputs. That means for layer L , we compute $z^{[L]}$ as usual. But for the activation function, we do the following:

$$t = e^{(z^{[L]})}$$

$$a^{[L]} = \frac{t}{\sum_{i=1}^C t_i}$$

Above, t is a temporary variable. We then divide this variables by the sum of all its values to *normalize* the output. This will then result in a vector containing decimal outputs that represents probabilities, which has the shape (C,1).

2.3.4 Programming Frameworks (TensorFlow)

IMPORTANT NOTE: This is in TensorFlow 1.0, so the syntax may be different with TensorFlow 2.0

Lets first take a look at how to **implement gradient descent** to minimize a cost function:

$$w^2 - 10w + 25$$

```

1 import numpy as np
2 import tensorflow as tf
3
4 w = tf.Variable(0, dtype=tf.float32) # define w
5 cost = tf.add(tf.add(w**2, tf.multiply(-10.,w)), 25) # define cost function
6 train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
7
8 init = tf.global_variables_initializer() # standard code
9 session = tf.Session() # create new session
10 session.run(init) # initialize our variables
11
12 for i in range(1000):
13     session.run(train) # train the model
14     print(session.run(w)) # 4.9999

```

Above, we wanted to minimize the cost function so that it would equal 0. The ideal value for w is 5, and our code ended up at 4.9999.

Lets now look at how to **read data** into TensorFlow and minimize the training data.

```

1 coefficient = np.array([[1.], [-10.], [25.]]) # data to plug into x
2 x = tf.placeholder(tf.float32, [3,1]) # think of as empty variables
3 cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
4 train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
5
6 init = tf.global_variables_initializer() # standard code
7 session = tf.Session() # create new session
8 session.run(init) # initialize our variables
9
10 for i in range(1000):
11     session.run(train, feed_dict={x:coefficients}) # map x to our coefficients
12     print(session.run(w)) # 4.9999 (same as above)

```

Note that TensorFlow automatically implements back propagation when performing forward propagation, so we do not need to explicitly write code for this.

2.3.5 Programming Assignment (TensorFlow 1)

In this assignment, we will first go over an introduction to TensorFlow and writing code for the steps we have learned about Deep Networks. After this, we will then build a model to classify sign language numbers of a persons hand (numbers 0 to 5).

Lets import the packages we will need for this assignment.

```
1 import math
2 import numpy as np
3 import h5py
4 import matplotlib.pyplot as plt
5 import tensorflow as tf
6 from tensorflow.python.framework import ops
7 from tf_utils import load_dataset, random_mini_batches, convert_to_one_hot, predict
8
9 %matplotlib inline
10 np.random.seed(1)
```

Lets start by computing the following **Linear function**: $Y = WX + b$, where W and X are random matrices and b is a random vector.

```
1 def linear_function():
2     np.random.seed(1)
3
4     X = tf.constant(np.random.randn(3,1), name='X')
5     W = tf.constant(np.random.randn(4,3), name='W')
6     b = tf.constant(np.random.randn(4,1), name='b')
7     Y = tf.add(tf.matmul(W,X), b)
8
9     # Create the session and run it on the variable you want to calculate
10    sess = tf.Session()
11    result = sess.run(Y)
12
13    # close the session
14    sess.close()
15
16    return result
```

Now that we have our linear function, lets calculate the **Sigmoid**. Luckily, TensorFlow has built in Neural Network functions such as *tf.sigmoid* and *tf.softmax*.

```
1 def sigmoid(z):
2     """
3         Computes the sigmoid of z
4
5         Arguments:
6             z -- input value, scalar or vector
7     """
8
9     # Create a placeholder for x
10    x = tf.placeholder(tf.float32, name='x')
11
12    # compute sigmoid(x)
13    sigmoid = tf.sigmoid(x)
14
15    # Create a session
16    with tf.Session() as sess:
17        # Run session and map z to x
18        result = sess.run(sigmoid, feed_dict={x:z})
19
20    return result
```

You can also use a built-in function to **compute the cost** of your neural network. So instead of needing to write code to compute this as a function of $a^{[2](i)}$ and $y^{(i)}$ for $i=1\dots m$:

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

you can do it in one line of code in TensorFlow. Note that the built in function will automatically compute the Sigmoid of z to get a.

```

1 def cost(logits, labels):
2     """
3         Computes the cost using the sigmoid cross entropy
4
5     Arguments:
6         logits -- vector containing z, output of the last linear unit
7         labels -- vector of labels y (1 or 0)
8
9     Note: logits will feed into z, and labels into y.
10    """
11    # Create the placeholders for "logits" (z) and "labels" (y)
12    z = tf.placeholder(tf.float32, name='z')
13    y = tf.placeholder(tf.float32, name='y')
14
15    # Use the loss function
16    cost = tf.nn.sigmoid_cross_entropy_with_logits(logits=z, labels=y)
17
18    sess = tf.Session() # Create a session
19
20    cost = sess.run(cost, feed_dict={z:logits, y:labels}) # Run the session
21
22    sess.close() # Close the session
23
24    return cost

```

Many times in deep learning you will have a y vector with numbers ranging from 0 to C-1, where C is the number of classes. If C is for example 4, then you might have the following y vector which you will need to convert as follows:



This is called **one hot encoding**, because in the converted representation exactly one element of each column is “hot” (meaning set to 1). In TensorFlow, we can do this with one line.

```

1 def one_hot_matrix(labels, C):
2     """
3         Arguments:
4             labels -- vector containing the labels
5             C -- number of classes, the depth of the one hot dimension
6     """
7
8     C = tf.constant(C, name='C') # our 'depth'
9
10    one_hot_matrix = tf.one_hot(labels, C, axis=0)
11
12    sess = tf.Session()
13    one_hot = sess.run(one_hot_matrix)
14    sess.close()
15    return one_hot # matrix of 0s and 1s

```

Now we will **initialize a vector** of zeros and ones.

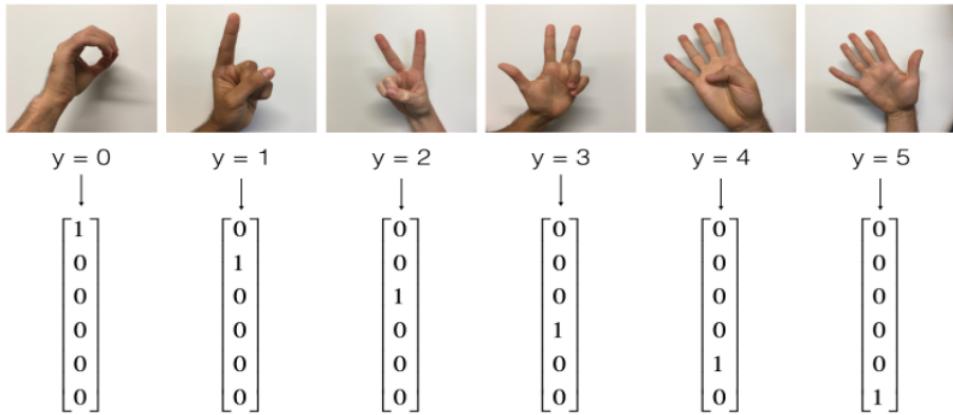
```

1 def ones(shape):
2     """
3     Arguments:
4     shape -- shape of the array you want to create
5     """
6     ones = tf.ones(shape)
7
8     sess = tf.Session()
9     ones = sess.run(ones)
10    sess.close()
11
12    return ones

```

Now we will **build a Neural Network** using TensorFlow. Our problem statement is that we want to create a model that can turn sign language numbers pictures (valued 0 to 5) into regular numbers.

- Training set: 1080 pictures (64x64 pixels) of signs representing numbers (180 pictures per number).
- Test set: 120 pictures (64x64 pixels) of signs representing numbers (20 pictures per number).



```

1 # Loading the dataset
2 X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset()

```

We want to **flatten** the image dataset, then **normalize** it by dividing by 255. On top of that, you will convert each label to a **one-hot** vector.

```

1 # Flatten the training and test images
2 X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1).T
3 X_test_flatten = X_test_orig.reshape(X_test_orig.shape[0], -1).T
4 # Normalize image vectors
5 X_train = X_train_flatten/255. # shape (12288, 1080)
6 X_test = X_test_flatten/255. # shape (6, 1080)
7 # Convert training and test labels to one hot matrices
8 Y_train = convert_to_one_hot(Y_train_orig, 6) # shape (6, 1080)
9 Y_test = convert_to_one_hot(Y_test_orig, 6) # shape (6, 120)

```

Note that 12288 comes from 64x64x3. Each image is square, 64 by 64 pixels, and 3 is for the RGB colors. Please make sure all these shapes make sense to you before continuing.

The goal is to build an algorithm capable of recognizing a sign with high accuracy. To do so, you are going to build a TensorFlow model that is almost the same as one you have previously built in NumPy for cat recognition (but now using a softmax output).

The model is LINEAR → RELU → LINEAR → RELU → LINEAR → SOFTMAX. The SIGMOID output layer has been converted to a SOFTMAX. A SOFTMAX layer generalizes SIGMOID to when there are more than two classes.

The first task is to create **placeholders** for ‘X’ and ‘Y’. This will allow you to later pass your training data in when you run your session. Note that n_x is the size of an image vector (12288) and n_y is the number of classes (6). We also use *none* for the shape because it allows us to be more flexible with m .

```

1 def create_placeholders(n_x, n_y):
2     X = tf.placeholder(tf.float32, [n_x, None], name='X')
3     Y = tf.placeholder(tf.float32, [n_y, None], name='Y')
4
5     return X, Y

```

The second task is to **initialize the parameters**. You are going to use Xavier Initialization for weights and Zero Initialization for biases. The shapes are given to us.

```

1 def initialize_parameters():
2     tf.set_random_seed(1) # so that your "random" numbers match ours
3
4     W1 = tf.get_variable("W1", [25,12288], initializer = tf.contrib.layers.
5                         xavier_initializer(seed = 1))
6     b1 = tf.get_variable("b1", [25,1], initializer = tf.zeros_initializer())
7     W2 = tf.get_variable("W2", [12,25], initializer = tf.contrib.layers.
8                         xavier_initializer(seed = 1))
9     b2 = tf.get_variable("b2", [12,1], initializer = tf.zeros_initializer())
10    W3 = tf.get_variable("W3", [6,12], initializer = tf.contrib.layers.
11                         xavier_initializer(seed = 1))
12    b3 = tf.get_variable("b3", [6,1], initializer = tf.zeros_initializer())
13
14    parameters = {"W1": W1, "b1": b1, "W2": W2,
15                  "b2": b2, "W3": W3, "b3": b3}
16
17    return parameters

```

We will now implement the **forward propagation** module. It is important to note that the forward propagation stops at ‘z3’. The reason is that in TensorFlow the last linear layer output is given as input to the function computing the loss. Note that we do not output any *cache*, we will see why during back propagation.

```

1 def forward_propagation(X, parameters):
2     """
3         Implements: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SOFTMAX
4
5         Arguments:
6             X -- input dataset placeholder, of shape (input size, m)
7             parameters -- python dictionary containing your parameters
8
9         Returns:
10            Z3 -- the output of the last LINEAR unit (used for softmax)
11
12    W1 = parameters['W1']
13    b1 = parameters['b1']
14    W2 = parameters['W2']
15    b2 = parameters['b2']
16    W3 = parameters['W3']
17    b3 = parameters['b3']
18
19    Z1 = tf.add(tf.matmul(W1, X), b1)
20    A1 = tf.nn.relu(Z1)
21    Z2 = tf.add(tf.matmul(W2, A1), b2)
22    A2 = tf.nn.relu(Z2)
23    Z3 = tf.add(tf.matmul(W3, A2), b3)
24
25    return Z3

```

Now it is time to **compute the cost**. Note that ‘logits’ and ‘labels’ are expected to have the shape (number of examples, number of classes), so that is why we transpose them below.

```

1 def compute_cost(Z3, Y):
2     """
3         Computes the cost
4
5         Arguments:
6             Z3 -- output of forward prop (the last LINEAR unit), of shape (6, m)
7             Y -- "true" labels vector placeholder, same shape as Z3
8
9         Returns:
10            cost - Tensor of the cost function
11        """
12
13    # fit the requirement for cost computation
14    logits = tf.transpose(Z3)
15    labels = tf.transpose(Y)
16
17    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
18                                            labels=labels))
19
20    return cost

```

Unlike before, frameworks handle all the **backpropagation** and the **parameters update** in 1 line of code. It is very easy to incorporate this line in the model. After you compute the cost function, you will create an *optimizer* object. You have to call this object along with the cost when running the `tf.Session`. When called, it will perform an optimization on the given cost with the chosen method and learning rate. One example of how to do this would be:

```

1 optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).
2         minimize(cost)
3 _, c = sess.run([optimizer, cost], feed_dict={X: minibatch_X, Y: minibatch_Y})

```

Now we will **build a model** using these functions and our training data.

```

1 def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.0001,
2           num_epochs = 1500, minibatch_size = 32, print_cost = True):
3     """
4         Implements a 3-layer tensorflow NN: LINEAR->RELU->LINEAR->RELU->LINEAR->SOFTMAX .
5
6         Arguments:
7             X_train -- training set (input size = 12288, number of training examples = 1080)
8             Y_train -- test set (output size = 6, number of training examples = 1080)
9             X_test -- training set (input size = 12288, number of training examples = 120)
10            Y_test -- test set (output size = 6, number of test examples = 120)
11            learning_rate -- learning rate of the optimization
12            num_epochs -- number of epochs of the optimization loop
13            minibatch_size -- size of a minibatch
14            print_cost -- True to print the cost every 100 epochs
15
16        Returns:
17            parameters -- parameters learnt by the model. They can then be used to predict.
18        """
19
20    ops.reset_default_graph() # able to rerun the model without overwriting tf
21    variables
22    tf.set_random_seed(1) # to keep consistent results
23    seed = 3 # to keep consistent results
24    (n_x, m) = X_train.shape # (n_x: input size, m : number of examples in the train
25    set)
26    n_y = Y_train.shape[0] # n_y : output size
27    costs = [] # To keep track of the cost

```

```

1   X, Y = create_placeholders(n_x, n_y) # Create Placeholders
2
3   parameters = initialize_parameters() # Initialize parameters
4
5   Z3 = forward_propagation(X, parameters) # Build the forward prop in the tf graph
6
7   cost = compute_cost(Z3, Y) # Add cost function to tensorflow graph
8
9   # Backpropagation: Define the tensorflow optimizer
10  optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
11
12  init = tf.global_variables_initializer() # Initialize all the variables
13
14  with tf.Session() as sess: # Start the session to compute the tensorflow graph
15      sess.run(init) # Run the initialization
16      for epoch in range(num_epochs): # Do the training loop
17          epoch_cost = 0. # Defines a cost related to an epoch
18          num_minibatches = int(m / minibatch_size) # number of minibatches
19          seed = seed + 1
20          minibatches = random_mini_batches(X_train, Y_train, minibatch_size, seed)
21
22          for minibatch in minibatches:
23              (minibatch_X, minibatch_Y) = minibatch # Select a minibatch
24
25              # Run the session to execute the "optimizer" and the "cost"
26              _, minibatch_cost = sess.run([optimizer, cost], feed_dict={X:minibatch_X,
27                                              Y:minibatch_Y})
28              epoch_cost += minibatch_cost / minibatch_size
29
30          # Print the cost every epoch
31          if print_cost == True and epoch % 100 == 0:
32              print ("Cost after epoch %i: %f" % (epoch, epoch_cost))
33          if print_cost == True and epoch % 5 == 0:
34              costs.append(epoch_cost)
35
36          # plot the cost
37          plt.plot(np.squeeze(costs))
38          plt.ylabel('cost')
39          plt.xlabel('iterations (per fives)')
40          plt.title("Learning rate =" + str(learning_rate))
41          plt.show()
42
43          # lets save the parameters in a variable
44          parameters = sess.run(parameters)
45          print("Parameters have been trained!")
46
47          # Calculate the correct predictions
48          correct_prediction = tf.equal(tf.argmax(Z3), tf.argmax(Y))
49
50          # Calculate accuracy on the test set
51          accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
52
53          print("Train Accuracy:", accuracy.eval({X: X_train, Y: Y_train}))
54          print("Test Accuracy:", accuracy.eval({X: X_test, Y: Y_test}))
55
56      return parameters

```

Our model resulted in a 99.9% train accuracy, and a 71.7% test accuracy. Overall, our model seems big enough to fit the data well, but given the difference between the train and test set, we could try adding in L2 or dropout regularization to reduce overfitting. Each time we run the session on a mini-batch, it trains the parameters and will eventually obtain the best parameters we could find.

3 Structuring Machine Learning Projects

3.1 Machine Learning Strategy 1

The chain of assumptions are:

- 1) Fit training set well on cost function → tune by bigger network / different optimizer.
- 2) Fit dev set well on cost function → tune by regularization / bigger training set.
- 3) Fit test set well on cost function → tune with a bigger dev set.
- 4) Performs well in real world → tune by changing dev set or cost function.

3.1.1 Evaluation Metrics

If we are hyperparameter tuning or trying new algorithms, our progress will be much faster if you have a single real number evaluation metric that lets us quickly tell if the new thing we just tried is working better or worse than your last idea. It's a good idea to have a **single number evaluation metric** to do this.

Precision - Of examples recognized as a certain class, the percentage that are correctly classified.

Recall - The percentage of actual class member that are correctly recognized.

F1 Score - The average of precision and recall (best metric to use when comparing classifiers).

Having a well defined dev set along with a single number evaluation metric allows us to tell which classifier is better. This can help us speed up the tuning our of model.

Another way we can choose which model we want to use after training is by using a **optimizing metric** and **satisficing metric**. For example, say we want to choose our model where we maximize accuracy subject to run time being less than a certain value. This means that accuracy is our *optimizing metric*, while run time is our *satisficing metric*.

In general, lets say we have N metrics we are observing for a given model. Then we want to choose 1 optimizing metric and N-1 satisficing metrics.

3.1.2 Setting up the Dev and Test sets

We want to make sure that our dev set and test set both come from the **same distribution**. Lets say that our data comes from many different regions (US, UK, China, etc.). Instead of choosing certain regions for each set, we want to *randomly shuffle* all of the data into dev/test sets to make sure that it is distributed as evenly as possible.

The general guideline is to choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on. You don't want to train your model to do one thing, only to then use it to try and predict something else.

We often work with very large datasets (millions of observations). When it comes to **splitting the data**, it could be enough to do a 98%, 1%, 1% split. For example, if we had 1,000,000 observations, then this split would give us 10,000 observations in both the dev and test sets. The size of the test set should be big enough to give high confidence in the overall performance of your system.

One note is that after deploying you model, if it allows users to use their own pictures, they might not be as high quality as the ones in the training/dev sets. If this is the case, you may want to add some of these more 'low quality' pictures into your sets in order to prepare your model.

3.1.3 Comparing to Human-Level Performance

Over time, our model can increase its accuracy to beyond human-level. However, this will eventually slow down and begin to level off. Ideally, we want our model to reach **Bayes optimal error**, or the best possible error, meaning that no function that computes $x \rightarrow y$ will perform better. This does not need to be 100% accuracy (it is normal to have some type of error) which can be caused by low quality pictures, audio, video, etc.

One reason that the model begins to slow down after passing human-level performance is because the difference between Bayes optimal error and human error is very close for many deep learning problems (image classification, audio, etc.). Another reason is because there are many tools that can help us surpass human-level performance such as:

- Get more labeled data from humans.
- Gain insight from manual error analysis.
- Better analysis of bias/variance.

We want our learning algorithm to do well on the data sets, but not too well. We can use **human-level error**, which we will consider very close to Bayes error and use as a proxy, to gauge where we want our model training/dev error to be near.

- **Avoidable bias** : the difference between human (Bayes) error and the training error.
- **Variance** : the difference between training and dev error.

If our human error is much lower than the training/dev error, we will focus on reducing the avoidable bias. If our human error is very close to the training/dev error, we will focus on variance reduction.

The two fundamental assumptions of supervised learning:

- 1) You can fit the training set pretty well (achieve low *avoidable bias*).
- 2) The training set performance generalizes pretty well to the dev/test set (achieve low *variance*).

We can reduce the following by:

- 1) Avoidable bias - bigger model, optimization algorithms, different NN architecture/hyperparameters.
- 2) Variance - more data, regularization, different NN architecture/hyperparameters.

3.2 Machine Learning Strategy 2

3.2.1 Error Analysis