Deep Neural Network

In this project, we will build a deep neural network from scratch.

The network consists of L layers (including the input layer). The last layer uses sigmoid activation, while the others use ReLU activation.

The training dataset is split into **mini batches**, which seeks to find a balance between stochastic gradient descent and batch gradient descent.

Three **initialization** methods are used: random initialization (standard normal distribution), He initialization, and Xavier initialization.

Two **regularization** techniques are implemented: inverted dropout and L2 regularization.

Three **optimization** algorithms are applied: gradient descent, Momentum, and Adam.

With **gradient checking**, we are able to numerically check the derivatives computed by our code to make sure that the implementation is correct.

1. Import Packages and Set Default Parameters

- numpy (http://www.numpy.org) is the main package for scientific computing with Python.
- matplotlib (http://matplotlib.org) is a library to plot graphs in Python.
- util_func provides some necessary functions for the calculations, e.g., Sigmoid, RELU.

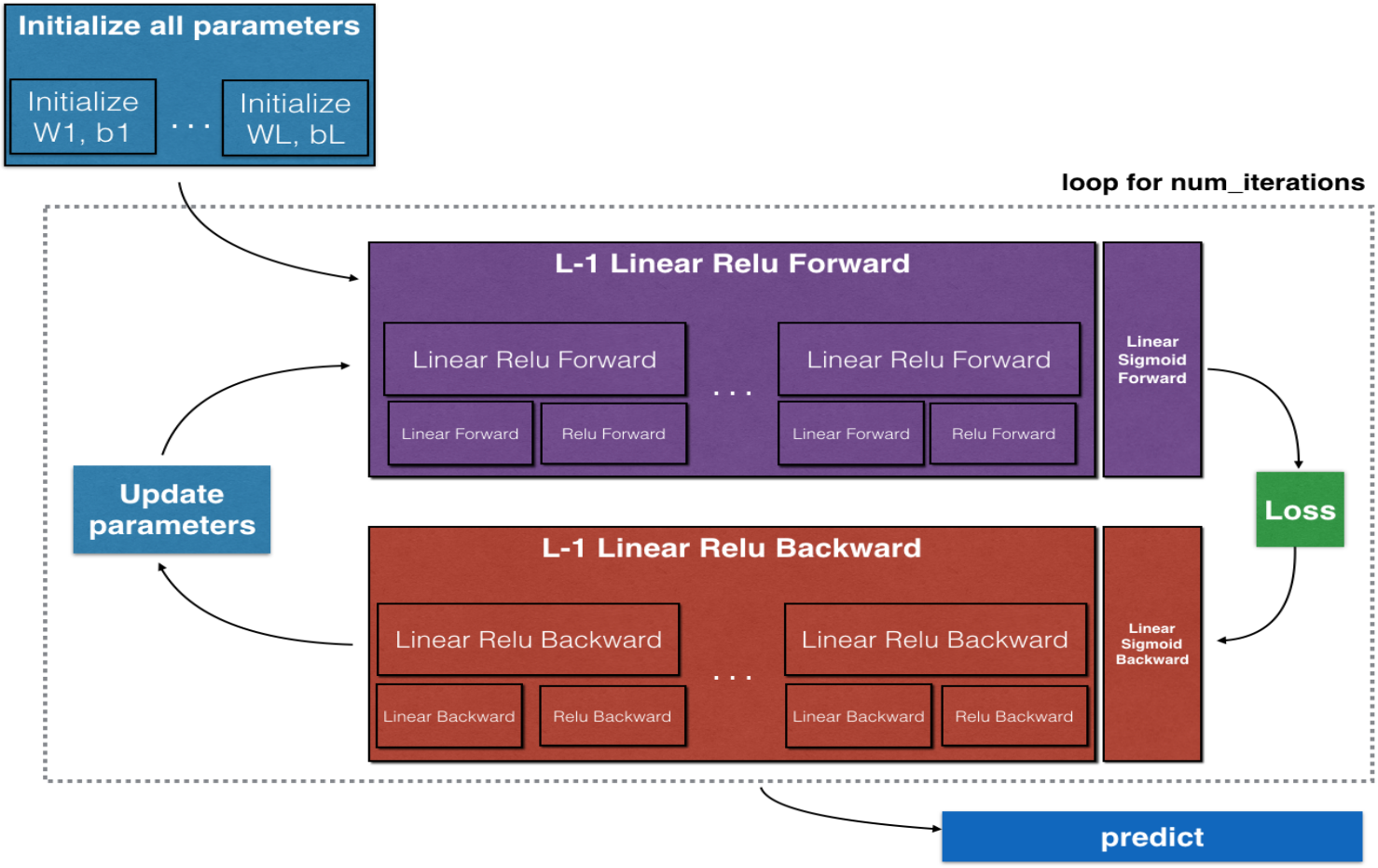
In [1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from util_func import *
from IPython.display import HTML
```

In [2]:
```
np.random.seed(20)
```

2. Outline of the Deep Neural Network

The procedures to implement the deep neural network is the following:

- Initialize the parameters (weight matrices W and bias vectors b) for an L-layer neural network.
- Implement forward propagation (shown in purple in the figure below):
  - For each layer l, complete the LINEAR part ($Z^{[l]}$).
  - Compute the ACTIVATION function (relu/sigmoid).
  - Stack the [LINEAR->RELU] forward function L-1 times (for layers 1 through L-1) and add a [LINEAR->SIGMOID] at the end (for the final layer L).
- Compute the cost.
- Implement the backward propagation (shown in red in the figure below):
  - For each layer l, compute $dZ^{[l]}$ using $dA^{[l]}$ from the last step.
  - compute $dW^{[l]}$, $db^{[l]}$ and $dA^{[l-1]}$ using $dZ^{[l]}$.
  - Repeat L times backward.
- Update the parameters.


Initialize all parameters
Initialize W1, b1
. . .
Initialize WL, bL
loop for num_iterations
L-1 Linear Relu Forward
Linear Relu Forward
Linear Forward
Relu Forward
. . .
Linear Relu Forward
Linear Forward
Relu Forward
Linear Sigmoid Forward
Loss
Update parameters
L-1 Linear Relu Backward
Linear Relu Backward
Linear Backward
Relu Backward
. . .
Linear Relu Backward
Linear Backward
Relu Backward
Linear Sigmoid Backward
predict

The functions to complete the steps above are included in this notebook.

Note that in the codes, the total number of layers, L, includes the input layer (l = 0).

Note that for every forward propagation, there is a corresponding backward propagation. That is why at forward propagation we will be storing some values in a cache. In the backpropagation propagation we will then use the cache to calculate the gradients.

Notation:

- Superscript $[l]$ denotes a quantity associated with the l^{th} layer.
  - Example: $a^{[L]}$ is the L^{th} layer activation. $W^{[L]}$ and $b^{[L]}$ are the L^{th} layer parameters.
- Superscript (i) denotes a quantity associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a_i^{[l]}$ denotes the i^{th} entry of the l^{th} layer's activations.

3. Initialization

3.1 Gradient Descent

We should make sure that our dimensions match between each layer. Recall that $n^{[l]}$ is the number of units in layer l. Thus for example if the size of our input X is $(12288, 209)$ (with $m = 209$ examples) then:

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
⋮	⋮	⋮	⋮	⋮
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Remember that when we compute $WX + b$ in python, it carries out broadcasting. For example, if:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix} \tag{1}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb + qe + rh) + u & (pc + qf + ri) + u \end{bmatrix} \tag{2}$$

- We will store $n^{[l]}$, the number of units in different layers, in a variable `layer_dims`. For example, if `layer_dims`is [2,4,1]: There are two inputs, one hidden layer with 4 hidden units, and an output layer with 1 output unit.
- Use zeros initialization for the biases.

A well chosen initialization can:

- Speed up the convergence of gradient descent
- Increase the odds of gradient descent converging to a lower training (and generalization) error

We can use three methods to initialize the weight matrices:

- Random initialization -- setting init_method = "random_normal" in the input argument. This initializes the weights to random values with standard normal distribution.
- He initialization -- setting init_method = "he" in the input argument. This initializes the weights to random values with a scaling factor of `sqrt(2/dimension of the previous layer)`, which is what He initialization recommends for layers with a ReLU activation.
- Xavier initialization -- setting init_method = "xavier" in the input argument. This initializes the weights to random values with a scaling factor of `sqrt(1/dimension of the previous layer)`, which is recommended for layers with a tanh activation.

Note that in general, initializing all the weights to zero results in the network failing to break symmetry. This means that every neuron in each layer will learn the same thing, and the network is no more powerful than a linear classifier such as logistic regression. Poor initialization can lead to vanishing/exploding gradients, which also slows down the optimization algorithm.

In [3]:
```python
# initialize parameters: weight matrices and bias vectors for each layer

def init_params(layer_dims, init_method, seed = 0):
    """
    Arguments:
    layer_dims: python array, layer_dims[l] is the number of units in the lth layer.
                l = 0 is the input layer, the last l is the output layer.
    init_method: choose which initialization method to use:
                "random_normal":  random values following standard normal distribution.
                "he": He initialization
                "xavier": Xavier initialization
    seed: random seed

    Returns:
    params: python dictionary containing weight matrices wl and bias vectors bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
            Wl has the dimension of (layer_dims[l], layer_dims[l - 1]).
            bl has the dimension of (layer_dims[l], 1)

    Use random initialization for the weight matrices, and use zeros initialization for the biases.
    """

    np.random.seed(seed)

    params = {} # parameters to be returned

    L = len(layer_dims) # total number of layers, including the input layer.

    for l in range(1, L):
        if init_method == "random_normal":
            params['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * 10
        elif init_method == "he":
            params['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * np.sqrt(2 / layer_dims[l-1])
        elif init_method == "xavier":
             params['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * np.sqrt(1 / layer_dims[l-1])
        params['b' + str(l)] = np.zeros((layer_dims[l], 1))

    return params
```

3.2 Momentum

Initialize the velocities for Momentum optimization. See Section 8.2 in this notebook for details.

$$v_{dW} = \beta * v_{dW} + (1 - \beta) * dW \tag{3}$$

$$v_{db} = \beta * v_{db} + (1 - \beta) * db \tag{4}$$

In [4]:
```
# initialize velocities for momentum optimization

def init_momentum(layer_dims):
    """
    Arguments:
    layer_dims: python array, layer_dims[l] is the number of units in the lth layer.
                l = 0 is the input layer, the last l is the output layer.

    Returns:
    v: python dictionary containing velocities for each layer,
            v['dW1'], v['dW2'], ..., v['dWl'], ..., v['db1'], v['db2'], ..., v['dbl'], ...
            v['dWl'] has the dimension of (layer_dims[l], layer_dims[l - 1])
            v['dbl'] has the dimension of (layer_dims[l], 1)

    Use zero initialization.
    """

    v = {}

    L = len(layer_dims) # total number of layers, including the input layer.

    for l in range(1, L):
        v['dW' + str(l)] = np.zeros((layer_dims[l], layer_dims[l-1]))
        v['db' + str(l)] = np.zeros((layer_dims[l], 1))

    return v
```

3.3 Adam Optimization

Initialize the velocities and squared gradients for Adam optimization. See Section 8.3 in this notebook for details.

$$v_{dW} = \beta_1 * v_{dW} + (1 - \beta_1) * dW \tag{5}$$

$$v_{db} = \beta_1 * v_{db} + (1 - \beta_1) * db \tag{6}$$

$$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * dW^2 \tag{7}$$

$$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * db^2 \tag{8}$$

$$v_{dW}^{corrected} = v_{dW} / (1 - \beta_1^t) \tag{9}$$

$$v_{db}^{corrected} = v_{db} / (1 - \beta_1^t) \tag{10}$$

$$S_{dW}^{corrected} = S_{dW} / (1 - \beta_2^t) \tag{11}$$

$$S_{db}^{corrected} = S_{db} / (1 - \beta_2^t) \tag{12}$$

In [5]:
```
# initialize v and s for Adam optimization

def init_adam(layer_dims):
    """
    Arguments:
    layer_dims: python array, layer_dims[l] is the number of units in the lth layer.
                l = 0 is the input layer, the last l is the output layer.

    Returns:
    v: python dictionary containing velocities for each layer,
            v['dW1'], v['dW2'], ..., v['dWl'], ..., v['db1'], v['db2'], ..., v['dbl'], ...
            v['dWl'] has the dimension of (layer_dims[l], layer_dims[l - 1]).
            v['dbl'] has the dimension of (layer_dims[l], 1).
    s: python dictionary containing squared gradients for each layer,
            s['dW1'], s['dW2'], ..., s['dWl'], ...,  s['db1'], s['db2'], ..., s['dbl'], ...
            s['dWl'] has the dimension of (layer_dims[l], layer_dims[l - 1]).
            s['dbl'] has the dimension of (layer_dims[l], 1)

    Use zero initialization.
    """

    v = {}
    s = {}

    L = len(layer_dims) # total number of layers, including the input layer.

    for l in range(1, L):
        v['dW' + str(l)] = np.zeros((layer_dims[l], layer_dims[l-1]))
        v['db' + str(l)] = np.zeros((layer_dims[l], 1))
        s['dW' + str(l)] = np.zeros((layer_dims[l], layer_dims[l-1]))
        s['db' + str(l)] = np.zeros((layer_dims[l], 1))

    return v, s
```

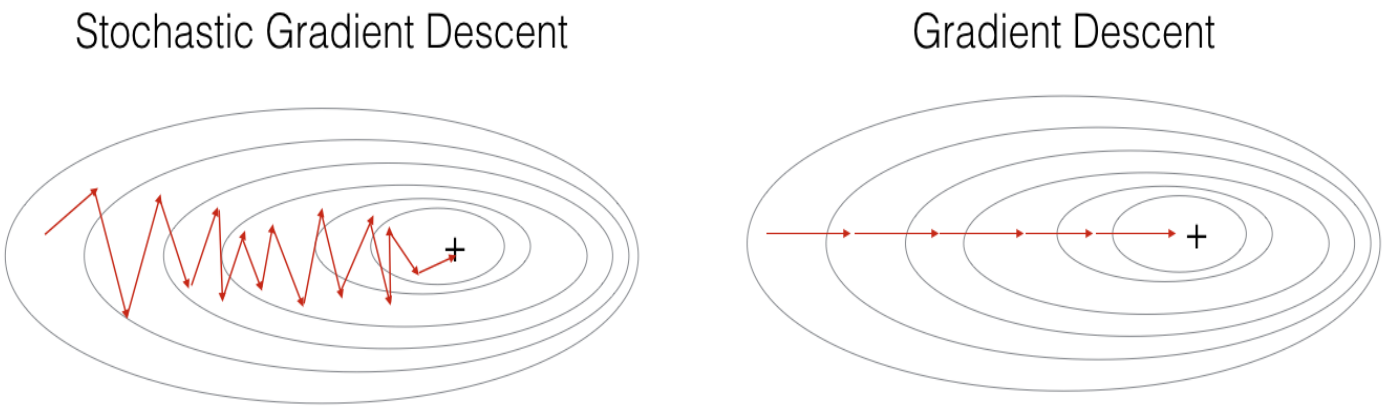
4. Mini-Batch Gradient Descent

4.1 Batch Gradient Descent

When we take gradient steps with respect to all m examples on each step, it is also called Batch Gradient Descent.

4.2 Stochastic Gradient Descent (SGD)

A variant of this is Stochastic Gradient Descent (SGD), which is equivalent to mini-batch gradient descent where each mini-batch has just 1 example. When the training set is large, SGD can be faster. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD)

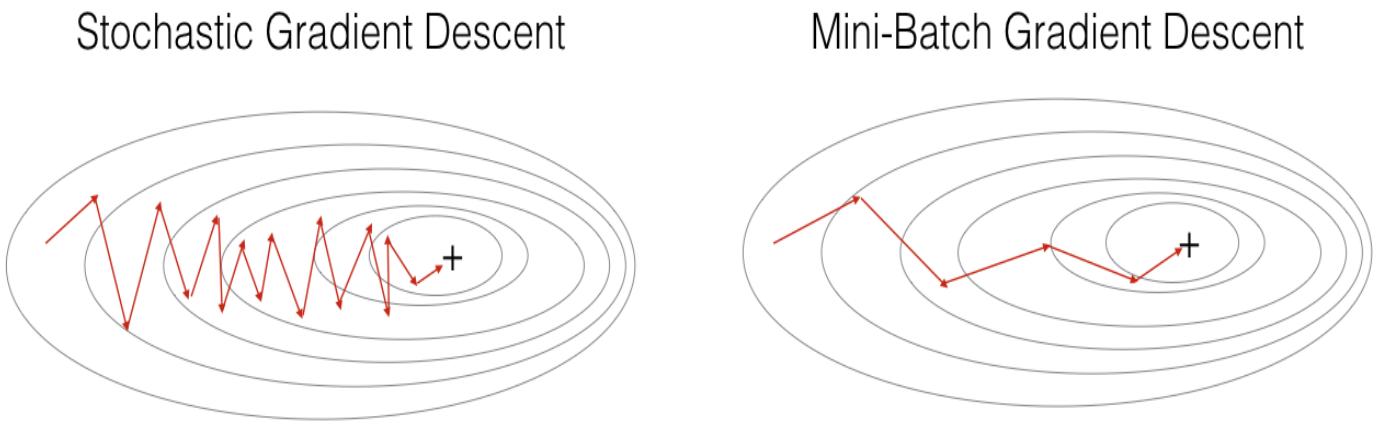


Implementing SGD requires 3 for-loops in total:

1. Over the number of iterations
2. Over the m training examples
3. Over the layers (to update all parameters, from $(W^{[1]}, b^{[1]})$ to $(W^{[L-1]}, b^{[L-1]})$)

4.3 Mini-Batch Gradient Descent

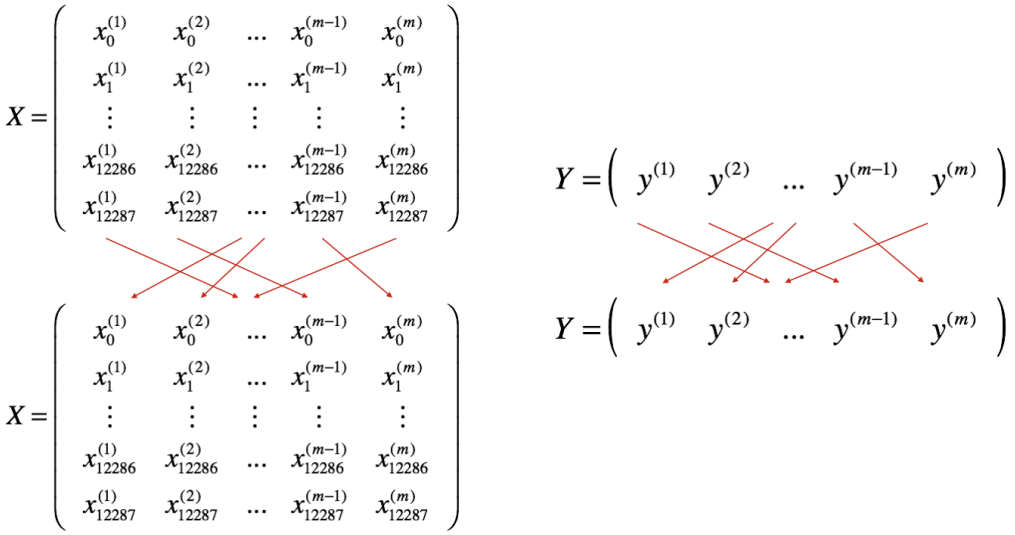
In practice, we'll often get faster results if we use neither the whole training set, nor only one training example, to perform each update. Mini-batch gradient descent uses an intermediate number of examples for each step. With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).



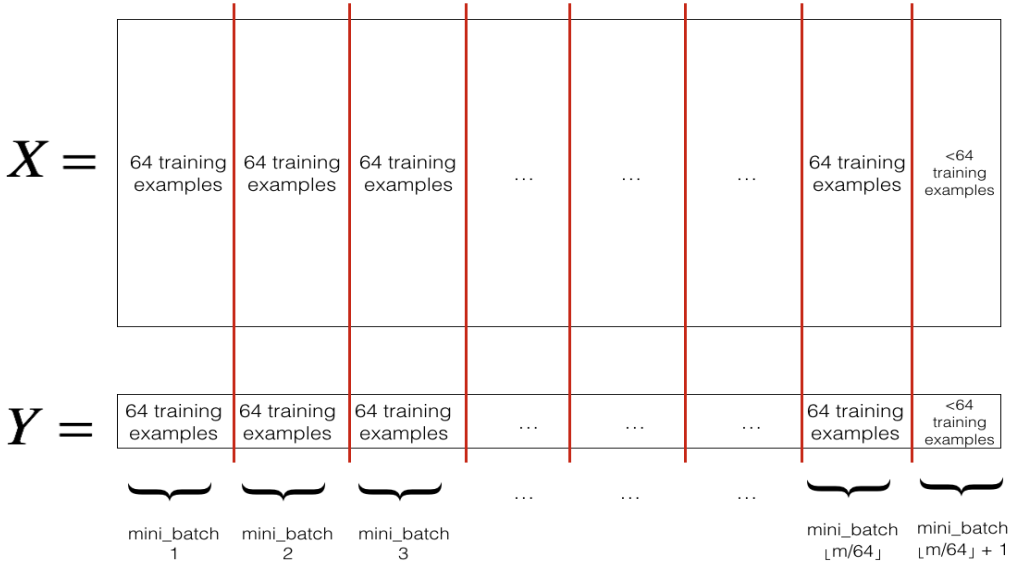
Note that powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

There are two steps:

- **Shuffle**: Create a shuffled version of the training set (X, Y). Note that the random shuffling 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.



- **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.


X =
64 training examples
64 training examples
64 training examples
…
…
…
64 training examples
<64 training examples
Y =
64 training examples
64 training examples
64 training examples
…
…
…
64 training examples
<64 training examples
mini_batch 1
mini_batch 2
mini_batch 3
mini_batch ⌊m/64⌋
mini_batch ⌊m/64⌋ + 1

In [6]:
```python
# create mini batches randomly, each with the size of mini_batch_size.

def random_mini_batches(X, Y, mini_batch_size, seed = 0):
    """
    Argument:
    X: input features, with dimension of (number of features, number of training examples)
    Y: true labels, the actual y values, with the dimension of (1, number of training examples)
    mini_batch_size: size of each mini batch.
    seed: random seed

    Returns:
    mini_batches: list of (mini_batch_X, mini_batch_Y)
    """

    np.random.seed(seed)

    m = X.shape[1] # total number of examples
    mini_batches = []

    # for batch gradient descent, no need to shuffle
    if mini_batch_size == X.shape[1]:
        mini_batch = (X, Y)
        mini_batches.append(mini_batch)
        return mini_batches

    # Step 1: shuffle
    permutation_indices = list(np.random.permutation(m))
    shuffled_X = X[:, permutation_indices]
    shuffled_Y = Y[:, permutation_indices]

    # Step 2: partition
    num_batch_except_last = math.floor(m / mini_batch_size)
    for i in range(num_batch_except_last):
        mini_batch_X = shuffled_X[:, i * mini_batch_size:(i + 1) * mini_batch_size - 1]
        mini_batch_Y = shuffled_Y[:, i * mini_batch_size:(i + 1) * mini_batch_size - 1]
        mini_batches.append((mini_batch_X, mini_batch_Y))
    # last mini batch
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[:, num_batch_except_last * mini_batch_size:]
        mini_batch_Y = shuffled_Y[:, num_batch_except_last * mini_batch_size:]
        mini_batches.append((mini_batch_X, mini_batch_Y))

    return mini_batches
```

5. Forward Propagation Module

5.1 Linear Forward

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

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{13}$$

where $A^{[0]} = X$.

In [7]:
```
# Calculate the Z value for forward propagation

def linear_forward_Z(A_prev, W, b):
    """
    Arguments:
    A_prev: the activations of the previous layer, with the dimension of (number of units in the previous layer, number of tra
ining examples)
    W: weight matrix for current layer
    b: bias vector for current layer

    Returns:
    Z: the input of current layer's activation function, also called pre-activation parameter, Z[l] = W[l]A[l - 1] + b[l],
        with the dimension of (number of units in current layer, number of training examples)
    linear_cache: (A_prev, W, b), stored for calculating backward propagation
    """


    Z = np.dot(W, A_prev) + b
    linear_cache = (A_prev, W, b)

    # verify the dimension correctness
    assert(Z.shape == (W.shape[0], A_prev.shape[1]))

    return Z, linear_cache
```

5.2 Linear-Activation Forward

In this notebook, we use two activation functions:

- **Sigmoid**: $\sigma(Z) = \frac{1}{1+e^{-(Z)}}$. The `sigmoid` function is provided in util_func.py.
- **ReLU**: $RELU(Z) = max(0, Z)$. The `relu` function is provided in util_func.py.

We group two functions (Linear and Activation) into one function (LINEAR->ACTIVATION),

$A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$ where the activation "g" can be sigmoid() or relu().

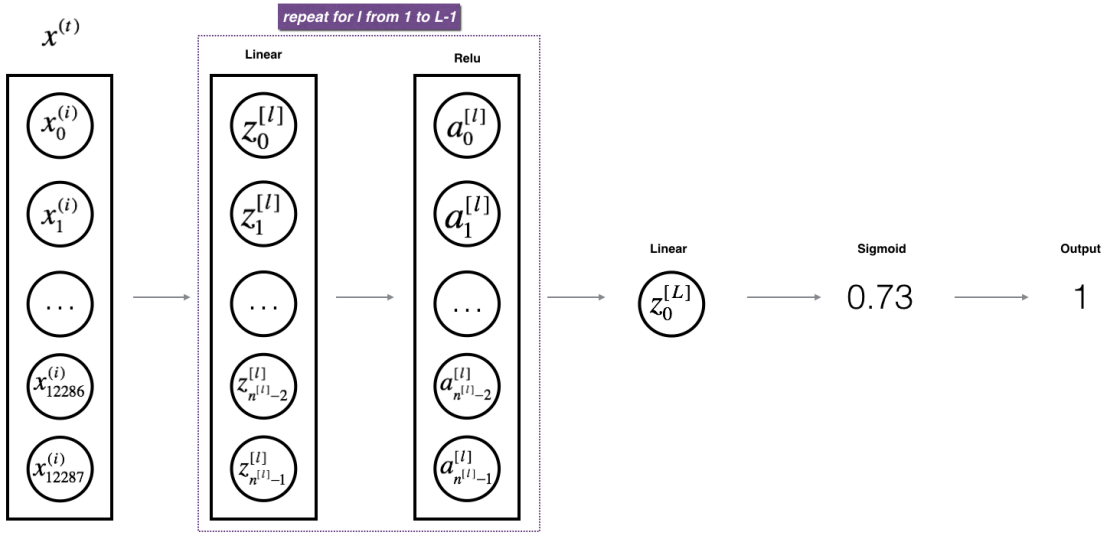
Note: In deep learning, the "[LINEAR->ACTIVATION]" computation is counted as a single layer in the neural network, not two layers.

In [8]:
```
# Calculate the activation function for current layer

def linear_forward_activation(A_prev, W, b, activation):
    """
    Arguments:
    A_prev: the activations of the previous layer, with the dimension of (number of units in the previous layer, number of tra
ining examples)
    W: weight matrix for current layer
    b: bias vector for current layer
    activation: "sigmoid" or "relu"

    Returns:
    A: the output of current layer's activation function, also called post-activation value,
        with the dimension of (number of units in current layer, number of training examples)
    cache: ((A_prev, W, b), Z) stored for calculating backward propagation
    """

    Z, linear_cache = linear_forward_Z(A_prev, W, b)
    if activation == "sigmoid":
        A = sigmoid(Z)
    elif activation == "relu":
        A = relu(Z)

    cache = (linear_cache, Z)

    return A, cache
```

5.3 L-Layer Forward Propagation

5.3.1 Without Dropout

When implementing the L-layer Neural Net, we need a function that replicates the previous one (`linear_forward_activation` with RELU) $L-1$ times, then follows that with one `linear_forward_activation` with SIGMOID.



In the code below, the variable `Aout` will denote $A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]}A^{[L-1]} + b^{[L]})$. (This is sometimes also called `Y hat`, i.e., $\hat{Y}$.)

The code below provides a full forward propagation that takes the input X and outputs a row vector $A^{[L]}$ containing our predictions. It also records all intermediate values in "caches".

In [9]:
```
# whole forward propagation of L-layer deep neural network, without dropout.
# The output layer uses sigmoid activation, other layers use relu activation.
# L includes the input layer

def L_layer_forward(X, params):
    """
    Arguments:
    X: input features, with dimension of (number of features, number of training examples)
    params: python dictionary containing weight matrices wl and bias vectors bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...

    Returns:
    Aout: output activation, i.e., the predicted y value, with the dimension of (1, number of examples)
    caches: the list of cache from linear_forward_activation(A_prev, W, b, activation) (totally L - 1 of them since L counts t
he input layer,
    indexed from 0 to L - 2)
    """

    caches = []

    # Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result),
    # you can use the // operator.
    L = len(params) // 2 + 1 # total number of layers including the input layer

    A_prev = X

    # relu activation for the layers except the last one
    for l in range(1, L - 1):
        W = params['W' + str(l)]
        b = params['b' + str(l)]
        A, cache = linear_forward_activation(A_prev, W, b, "relu")
        caches.append(cache)
        A_prev = A

    # sigmoid activation for the last layer
    Aout, cache = linear_forward_activation(A_prev, params['W' + str(L - 1)], params['b' + str(L - 1)], "sigmoid")
    caches.append(cache)

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

    return Aout, caches
```

5.3.2 Inverted Dropout

Dropout is a widely used regularization technique that is specific to deep learning.**It randomly shuts down some neurons in each iteration.**

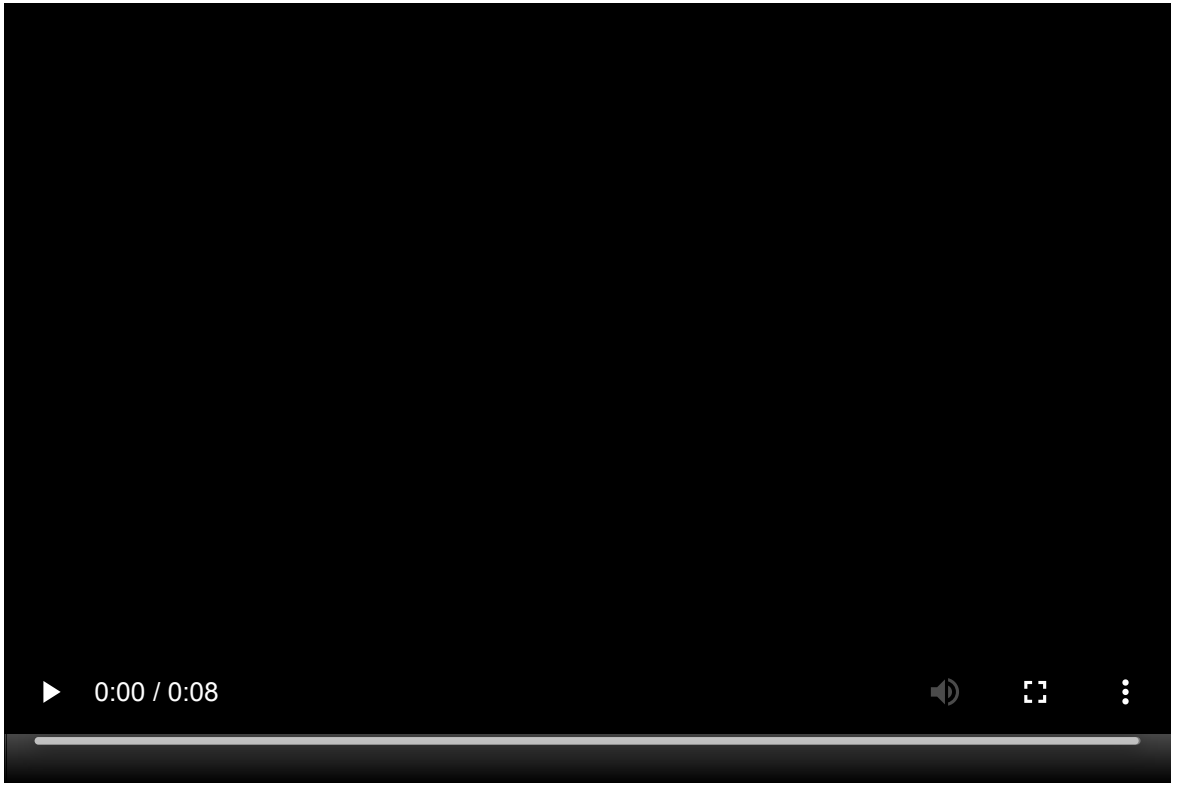
At each iteration, we shut down (= set to zero) each neuron of a layer with probability $1 - keep\_prob$ or keep it with probability $keep\_prob$. The dropped neurons don't contribute to the training in both the forward and backward propagations of the iteration.

To understand drop-out, consider this conversation with a friend:

- Friend: "Why do you need all these neurons to train your network and classify images?".
- You: "Because each neuron contains a weight and can learn specific features/details/shape of an image. The more neurons I have, the more featurse my model learns!"
- Friend: "I see, but are you sure that your neurons are learning different features and not all the same features?"
- You: "Good point... Neurons in the same layer actually don't talk to each other. It should be definitly possible that they learn the same image features/shapes/forms/details... which would be redundant. There should be a solution."

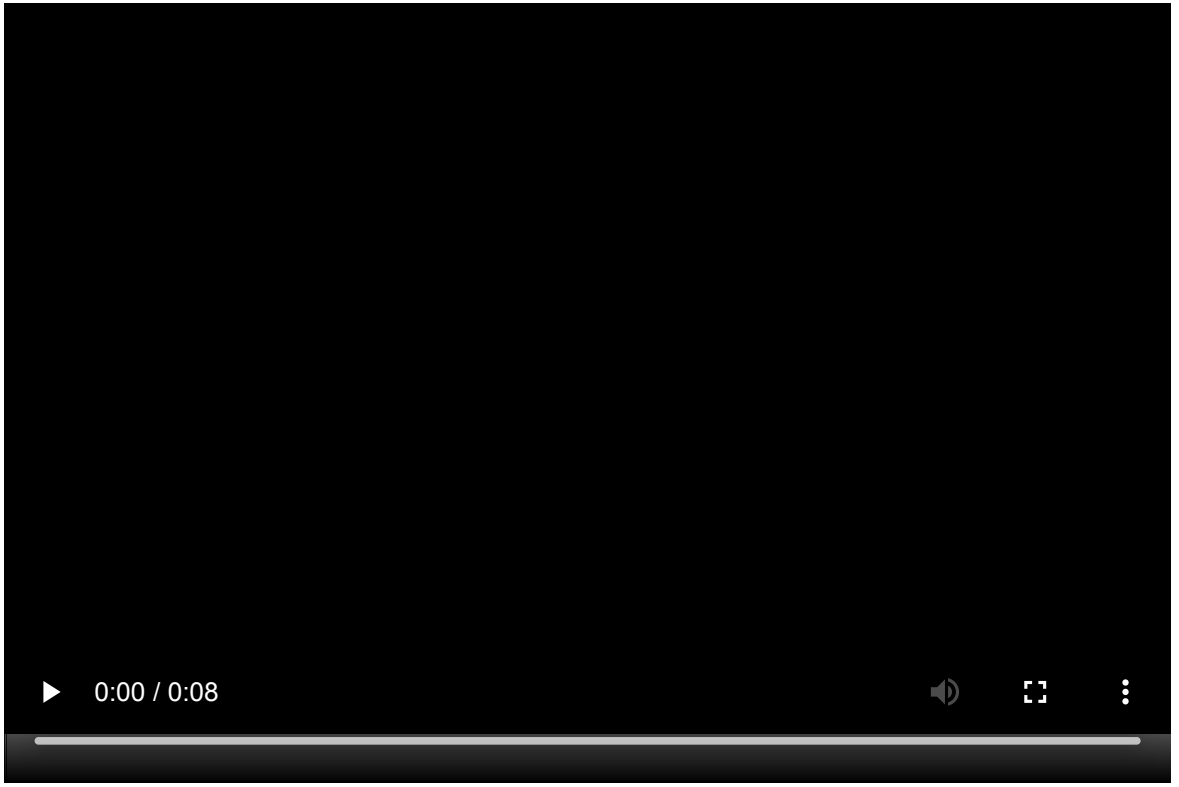
In [10]:
```
HTML('''
<center>
    <video width="620" height="440" controls>
        <source src="videos/dropout1.mp4" type="video/mp4">
    </video>
</center>
''')
```

Out[10]:



In [11]:
```
HTML('''
<center>
    <video width="620" height="440" controls>
        <source src="videos/dropout2.mp4" type="video/mp4">
    </video>
</center>
''')
```

Out[11]:



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

We carry out 4 Steps:

1. Create a random matrix $D^{[1]} = [d^{[1](1)}, d^{[1](2)} \dots d^{[1](m)}]$ of the same dimension as $A^{[1]}$.
2. Set each entry of $D^{[1]}$ to be 0 with probability (`1-keep_prob`) or 1 with probability (`keep_prob`), by thresholding values in $D^{[1]}$ appropriately.
3. Set $A^{[1]}$ to $A^{[1]} * D^{[1]}$. (shutting down some neurons). We can think of $D^{[1]}$ as a mask.
4. Divide $A^{[1]}$ by `keep_prob`. By doing this we are assuring that the result of the cost will still have the same expected value as without drop-out. (This technique is also called inverted dropout.)

In [12]:
```
# whole forward propagation of L-layer deep neural network, with inverted dropout.
# The output layer uses sigmoid activation, other layers use relu activation.
# L includes the input layer

def L_layer_forward_inverted_dropout(X, params, keep_prob, seed = 0):
    """
    Arguments:
    X: input features, with dimension of (number of features, number of training examples)
    params: python dictionary containing weight matrices wl and bias vectors bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
    keep_prob: parameter for inverted dropout,the probability of keeping a neuron
    seed: random seed

    Returns:
    Aout: output activation, i.e., the predicted y value, with the dimension of (1, number of examples)
    caches: the list of cache from linear_forward_activation(A_prev, W, b, activation) (totally L - 1 of them since L counts t
he input layer,
    indexed from 0 to L - 2)
    dropout_masks: the list of mask matrices for shutting down neurons in each layer (except the input and output layers)
    """

    np.random.seed(seed)

    caches = []
    dropout_masks = []

    L = len(params) // 2 + 1 # total number of layers including the input layer

    A_prev = X

    # relu activation for the layers except the last one
    for l in range(1, L - 1):
        W = params['W' + str(l)]
        b = params['b' + str(l)]
        A, cache = linear_forward_activation(A_prev, W, b, "relu")
        caches.append(cache)

        #mask = np.random.rand(A.shape[0], A.shape[1]) # random samples from a uniform distribution over [0, 1)
        #If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample
        mask = np.random.random_sample(A.shape)

        # to set all the entries of a matrix X to 0 (if entry is less than 0.4) or 1 (if entry is more than 0.4) you would do:
X = (X < 0.4).
        # Note that 0 and 1 are respectively equivalent to False and True.
```

```
            mask = (mask < keep_prob)
            A = A * mask
            A = A / keep_prob
            dropout_masks.append(mask)

            A_prev = A

        # sigmoid activation for the last layer
        Aout, cache = linear_forward_activation(A_prev, params['W' + str(L - 1)], params['b' + str(L - 1)], "sigmoid")
        caches.append(cache)

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

        return Aout, caches, dropout_masks
```

6. Cost Function

6.1 Non-Regularized

Compute the cross-entropy cost J, using the following formula:

$$J = -\frac{1}{m} \sum_{i=1}^{m} (y^{(i)} \log\left(a^{[L](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[L](i)}\right)) \tag{14}$$

In [13]:
```python
# calculate the cross-entropy cost function

def cost_func(Aout, Y):
    """
    Arguments:
    Aout: output activation, i.e., the predicted y value, with the dimension of (1, number of examples)
    Y: true labels, the actual y values, with the dimension of (1, number of examples)

    Returns:
    cost: the cross-entropy cost
    """

    m = Y.shape[1]

    # numpy.multiply(): multiply arguments element-wise.Equivalent to x1 * x2 in terms of array broadcasting.
    # numpy.nansum(): treating Not a Numbers (NaNs) as zero.
    cost = -1 / m * np.nansum(np.multiply(Y, np.log(Aout)) + np.multiply(1 - Y, np.log(1 - Aout)), axis = 1, keepdims = True)
    cost = np.squeeze(cost) # e.g. this turns [[5]] into 5
    assert(cost.shape == ())

    return cost
```

6.2 L2 Regularization

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}} \tag{15}$$

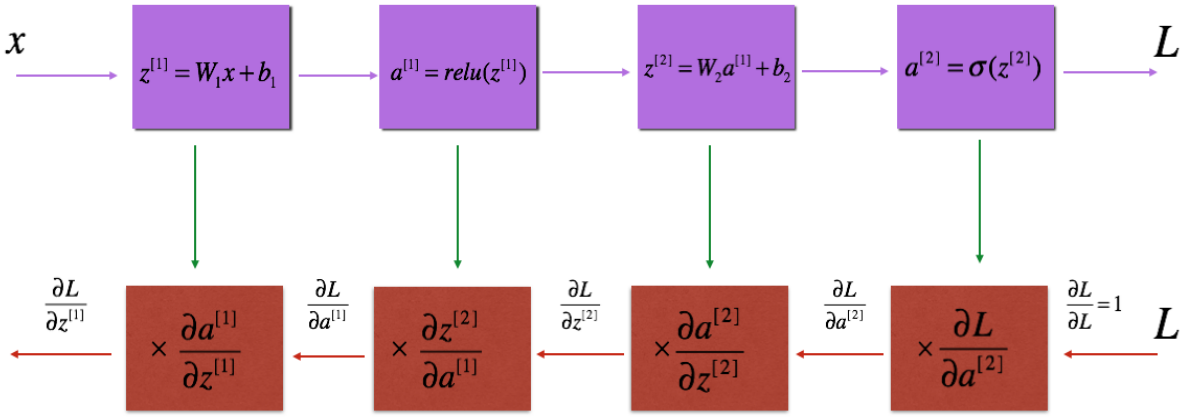
What is L2-regularization actually doing?

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function we drive all the weights to smaller values. It becomes too costly for the cost to have large weights. This leads to a smoother model in which the output changes more slowly as the input changes.

In [14]:
```
# calculate the cost function with L2 regularization.

def cost_func_L2_Regul(Aout, Y, params, lambd):
    """
    Arguments:
    Aout: output activation, i.e., the predicted y value, with the dimension of (1, number of examples)
    Y: true labels, the actual y values, with the dimension of (1, number of training examples)
    params: python dictionary containing weight matrices wl and bias vectors bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
    lambd: the lambda parameter for L2 regularization

    Returns:
    cost: the cost value
    """

    m = Y.shape[1]
    cross_entropy_cost = -1 / m * np.nansum(np.multiply(Y, np.log(Aout)) + np.multiply(1 - Y, np.log(1 - Aout)), axis = 1, kee
pdims = True)

    L = len(params) // 2 + 1
    L2_regularization_cost = 0
    for l in range(1, L):
        L2_regularization_cost += np.nansum(np.square(params['W' + str(l)]))
    L2_regularization_cost *= lambd / (2 * m)

    cost = cross_entropy_cost + L2_regularization_cost
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    return cost
```

7. Backward Propagation Module

Back propagation is used to calculate the gradient of the loss function with respect to the parameters.



The chain rule of calculus can be used to derive the derivative of the loss $\mathcal{L}$ with respect to $z^{[1]}$ in a 2-layer network as follows:

$$\frac{d\mathcal{L}(a^{[2]}, y)}{dz^{[1]}} = \frac{d\mathcal{L}(a^{[2]}, y)}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}} \frac{da^{[1]}}{dz^{[1]}} \tag{16}$$

In order to calculate the gradient $dW^{[1]} = \frac{\partial L}{\partial W^{[1]}}$, we use the previous chain rule and we do $dW^{[1]} = dz^{[1]} \times \frac{\partial z^{[1]}}{\partial W^{[1]}}$. During the backpropagation, at each step we multiply our current gradient by the gradient corresponding to the specific layer to get the gradient we want.

Equivalently, in order to calculate the gradient $db^{[1]} = \frac{\partial L}{\partial b^{[1]}}$, we use the previous chain rule and we do $db^{[1]} = dz^{[1]} \times \frac{\partial z^{[1]}}{\partial b^{[1]}}$.

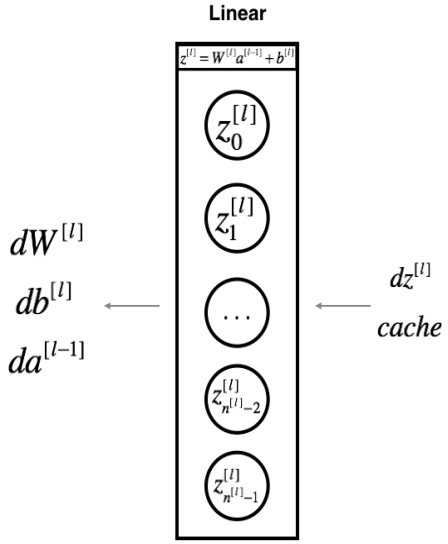
This is why we talk about **backpropagation**.

7.1 Linear Backward

7.1.1 Non-Regularized

For layer l, the linear part is: $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$ (followed by an activation).

Suppose we have already calculated the derivative $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$. We want to get $\left(dW^{[l]}, db^{[l]}, dA^{[l-1]}\right)$.



The three outputs $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$ are computed using the input $dZ^{[l]}$.Here are the formulas:

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{17}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{18}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{19}$$

Note that in this calculation, dA and dZ are alwarys the derivatives with respect to the loss function NOT the cost function. We treat dA and dZ differently from dW and dB. Namely, dA and dZ are always computing the derivatives d$\mathcal{L}$/dA and d$\mathcal{L}$/dZ respectively, but dW and db are computing the derivatives dJ/dW and dJ/db. This is why we don't have the 1/m factor in the dA formula.

In [15]:
```python
# backward propagation for current layer, given dZ
# In our calculation, dA and dZ are alwarys the derivatives with respect to the loss function NOT the cost function.
# We treat dA and dZ differently from dW and dB. Namely, dA and dZ are always computing the derivatives dL/dA and dL/dZ respec
tively,
# but dW and db are computing the derivatives dJ/dW and dJ/db.
# This is why we don't have the 1/m factor in the dA formula

def linear_backward_from_dZ(dZ, linear_cache):
    """
    Arguments:
    dZ: gradient of the loss with respect to Z for current layer
    linear_cache: (A_prev, W, b) stored from linear forward propagation for current layer

    Returns:
    dA_prev: gradient of the loss with respect to A_prev, for current layer, with the same dimension of A_prev
    dW: gradient of the cost with respect to W, for current layer, with the same dimension of W
    db: gradient of the cost with respect to b, for current layer, with the same dimension of b
    """

    A_prev, W, b = linear_cache
    m = dZ.shape[1]

    dW = 1 / m * np.dot(dZ, A_prev.T)
    db = 1 / m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T, dZ)

    assert(dW.shape == W.shape)
    assert(db.shape == b.shape)
    assert(dA_prev.shape == A_prev.shape)

    return dA_prev, dW, db
```

7.1.2 L2 Regularization

If L2 regularization is used, because we change the cost, we have to change backward propagation as well. All the gradients have to be computed with respect to this new cost. The changes only concern dW. For each, we have to add the regularization term's gradient ($\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W$).

In [16]:

```python
# backward propagation for current layer, given dZ, with L2 regularization

def linear_backward_from_dZ_L2_Regul(dZ, linear_cache, lambd):
    """
    Arguments:
    dZ: gradient of the loss with respect to Z for current layer
    linear_cache: (A_prev, W, b) stored from forward propagation for current layer
    lambd: the lambda parameter for L2 regularization

    Returns:
    dA_prev: gradient of the loss with respect to A_prev, for current layer, with the same dimension of A_prev
    dW: gradient of the cost with respect to W, for current layer, with the same dimension of W
    db: gradient of the cost with respect to b, for current layer, with the same dimension of b
    """

    A_prev, W, b = linear_cache
    m = dZ.shape[1]

    dW = 1 / m * np.dot(dZ, A_prev.T) + (lambd / m) * W # this is the only difference between L2 and non-regularized
    db = 1 / m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T, dZ)

    assert(dW.shape == W.shape)
    assert(db.shape == b.shape)
    assert(dA_prev.shape == A_prev.shape)

    return dA_prev, dW, db
```

7.2 Linear-Activation Backward

7.2.1 Non-Regularized

Next, we create a function that merges **`linear_backward_from_dZ`** and the backward step for the activation (dA).

Two backward functions are provided in util_func.py:

- **`sigmoid_backward(dA, Z)`**: Implements the backward propagation for SIGMOID unit.
- **`relu_backward(dA, Z)`**: Implements the backward propagation for RELU unit.

If $g(.)$ is the activation function, `sigmoid_backward` and `relu_backward` compute

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}) \tag{20}$$

.

Then we just use $dZ^{[l]}$ as the input of above function, linear_backward_from_dZ, to get $dW^{[l]}$, $db^{[l]}$, $dA^{[l-1]}$.

In [17]:
```
# backward propagation for current layer, given dA

def activation_backward_from_dA(dA, cache, activation):
    """
    Arguments:
    dA: gradient of the loss with respect to A[l] for current layer l
    cache: ((A_prev, W, b), Z) stored from forward propagation for current layer
    activation: "sigmoid" or "relu"

    Returns:
    dA_prev: gradient of the loss with respect to A_prev (A[l - 1]), for current layer, with the same dimension of A_prev
    dW: gradient of the cost with respect to W, for current layer, with the same dimension of W
    db: gradient of the cost with respect to b, for current layer, with the same dimension of b
    """
    linear_cache, Z = cache

    if activation == "sigmoid":
        dZ = sigmoid_backward(dA, Z)
    elif activation == "relu":
        dZ = relu_backward(dA, Z)

    dA_prev, dW, db = linear_backward_from_dZ(dZ, linear_cache)

    return dA_prev, dW, db
```

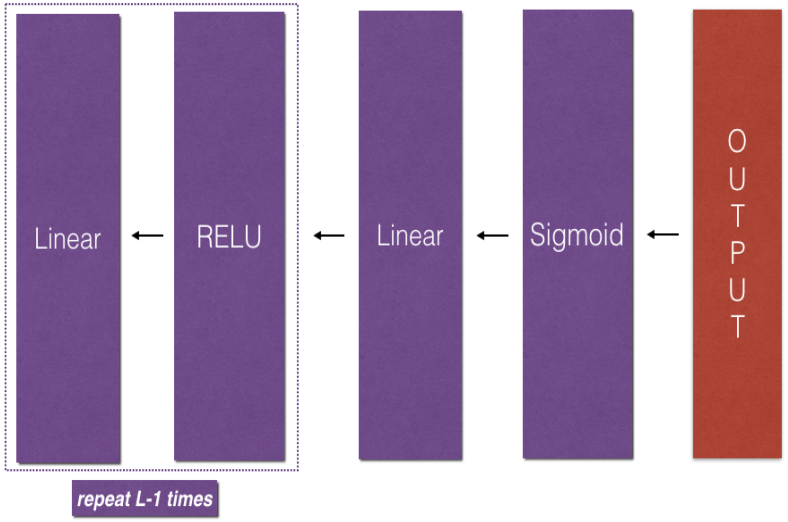
7.2.2 L2 Regularization

In [18]:
```
# backward propagation for current layer, given dA, with L2 regularization

def activation_backward_from_dA_L2_Regul(dA, cache, activation, lambd):
    """
    Arguments:
    dA: gradient of the loss with respect to A[l] for current layer l
    cache: ((A_prev, W, b), Z) stored from forward propagation for current layer
    activation: "sigmoid" or "relu"
    lambd: the lambda parameter for L2 regularization

    Returns:
    dA_prev: gradient of the loss with respect to A_prev (A[l - 1]), for current layer, with the same dimension of A_prev
    dW: gradient of the cost with respect to W, for current layer, with the same dimension of W
    db: gradient of the cost with respect to b, for current layer, with the same dimension of b
    """
    linear_cache, Z = cache

    if activation == "sigmoid":
        dZ = sigmoid_backward(dA, Z)
    elif activation == "relu":
        dZ = relu_backward(dA, Z)

    dA_prev, dW, db = linear_backward_from_dZ_L2_Regul(dZ, linear_cache, lambd)

    return dA_prev, dW, db
```

7.3 L-Layer Backward Propagation

7.3.1 Non-Regularized

Now we implement the backward function for the whole network. Recall that when we implemented the `L_layer_forward` function, at each layer l, we stored a cache which contains $((A^{[l-1]}, W^{[l]}, b^{[l]}), Z^{[l]})$. In the back propagation module, we use those caches to compute the gradients. Therefore, we iterate through all the hidden layers backward, starting from layer L. On each step, we use the cached values for layer l to backpropagate through layer l.



Initializing backpropagation: To backpropagate through this network, we know that the output is, $Aout = A^{[L]} = \sigma(Z^{[L]})$. We thus need to compute `dAout` $= \frac{\partial \mathcal{L}}{\partial A^{[L]}}$. To do so, use this formula:

```
dAout = - (np.divide(Y, Aout) - np.divide(1 - Y, 1 - Aout)) # derivative of loss with respect to Aout
```

We can now feed in `dAout` into the LINEAR->SIGMOID backward function. After that, we have to use a `for` loop to iterate through all the other layers using the LINEAR->RELU backward function. We should store each dA, dW, and db in the grads dictionary.

Note: We actually don't use the $dAout$ formula above in the code; instead, we calculate $dZ^{[L]}$ for the output layer using $Aout - Y$. This is because $Aout$ could sometimes be 1 due to computer mathematical limitation, so np.divide(1 - Y, 1 - Aout) becomes np.divide(1 - Y, 0) which causes problems.

In [19]:

```
# whole backward propagation of L-Layer deep neural network
# The output layer use sigmoid activation, other layers use relu activation.

def L_layer_backward(Aout, Y, caches):
    """
    Arguments:
    Aout: output activation, i.e., the predicted y value, with the dimension of (1, number of examples)
    Y: true labels, the actual y values, with the dimension of (1, number of training examples)
    caches: the list of cache from L_layer_forward(X, params), i.e., list of ((A_prev, W, b), Z)
            (totally L - 1 of them since L counts the input layer, indexed from 0 to L - 2)

    Returns:
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWl'], ..., and grads['db1'], grads['db2'], ..., grads['dbl'], ...
    """


    grads = {}
    L = len(caches) + 1 # total number of layers including the input layer
    # gradient of the loss with respect to the output Aout
    # numpy.divide(): Divide arguments element-wise. Equivalent to x1 / x2 in terms of array-broadcasting.
    #dAout = - Y / Aout + (1 - Y) / (1 - Aout)
    dZout = Aout - Y

    # calculate gradients for the last layer using sigmoid activation
    cache = caches[-1]
    linear_cache, _ = cache
    #dA_prev, dW, db = activation_backward_from_dA(dAout, cache, "sigmoid")
    dA_prev, dW, db = linear_backward_from_dZ(dZout, linear_cache)
    grads['dW' + str(L - 1)] = dW
    grads['db' + str(L - 1)] = db

    # calculate gradients for all the previous layers using relu activation
    for l in range(L - 3, -1, -1):
        cache = caches[l]
        dA_prev, dW, db = activation_backward_from_dA(dA_prev, cache, "relu")
        grads['dW' + str(l + 1)] = dW
        grads['db' + str(l + 1)] = db

    return grads
```

7.3.2 L2 Regularization

In [20]:
```
# whole backward propagation of L-layer deep neural network, with L2 regularization
# The output layer use sigmoid activation, other layers use relu activation.

def L_layer_backward_L2_Regul(Aout, Y, caches, lambd):
    """
    Arguments:
    Aout: output activation, i.e., the predicted y value, with the dimension of (1, number of examples)
    Y: true labels, the actual y values, with the dimension of (1, number of training examples)
    caches: the list of cache from L_layer_forward(X, params), i.e., list of ((A_prev, W, b), Z)
            (totally L - 1 of them since L counts the input layer, indexed from 0 to L - 2)
    lambd: the lambda parameter for L2 regularization

    Returns:
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWL'], ..., and grads['db1'], grads['db2'], ..., grads['dbL'], ...
    """

    grads = {}
    L = len(caches) + 1 # total number of layers including the input layer
    # gradient of the cost with respect to the output Aout
    # dAout = - Y / Aout + (1 - Y) / (1 - Aout)
    dZout = Aout - Y

    # calculate gradients for the last layer using sigmoid activation
    cache = caches[-1]
    linear_cache, _ = cache
    #dA_prev, dW, db = activation_backward_from_dA_L2_Regul(dAout, cache, "sigmoid", lambd)
    dA_prev, dW, db = linear_backward_from_dZ_L2_Regul(dZout, linear_cache, lambd)
    grads['dW' + str(L - 1)] = dW
    grads['db' + str(L - 1)] = db

    # calculate gradients for all the previous layers using relu activation
    for l in range(L - 3, -1, -1):
        cache = caches[l]
        dA_prev, dW, db = activation_backward_from_dA_L2_Regul(dA_prev, cache, "relu", lambd)
        grads['dW' + str(l + 1)] = dW
        grads['db' + str(l + 1)] = db

    return grads
```

7.3.3 Inverted Dropout

We carry out 2 Steps:

1. We had previously shut down some neurons during forward propagation, by applying a mask $D^{[l]}$ to $A^{[l]}$. In backpropagation, we shut down the same neurons, by reapplying the same mask $D^{[1]}$ to $dA^{[l]}$.
2. During forward propagation, we had divided $A^{[l]}$ by `keep_prob`. In backpropagation, we have to divide $dA^{[l]}$ by `keep_prob` again (the calculus interpretation is that if $A^{[1]}$ is scaled by `keep_prob`, then its derivative $dA^{[1]}$ is also scaled by the same `keep_prob`).

In [21]:
```
# whole backward propagation of L-Layer deep neural network, with inverted dropout
# The output layer use sigmoid activation, other layers use relu activation.

def L_layer_backward_inverted_dropout(Aout, Y, caches, keep_prob, dropout_masks):
    """
    Arguments:
    Aout: output activation, i.e., the predicted y value, with the dimension of (1, number of examples)
    Y: true labels, the actual y values, with the dimension of (1, number of training examples)
    caches: the list of cache from L_layer_forward(X, params), i.e., list of ((A_prev, W, b), Z)
            (totally L - 1 of them since L counts the input layer, indexed from 0 to L - 2)
    keep_prob: parameter for inverted dropout,the probability of keeping a neuron
    dropout_masks: the mask matrices for shutting down neurons in each layer (except the input and output layers)

    Returns:
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWl'], ..., and grads['db1'], grads['db2'], ..., grads['dbl'], ...
    """

    grads = {}
    L = len(caches) + 1 # total number of layers including the input layer
    # gradient of the cost with respect to the output Aout
    # dAout = - Y / Aout + (1 - Y) / (1 - Aout)
    dZout = Aout - Y

    # calculate gradients for the last layer using sigmoid activation
    cache = caches[-1]
    linear_cache, _ = cache
    # dA_prev, dW, db = activation_backward_from_dA(dAout, cache, "sigmoid")
    dA_prev, dW, db = linear_backward_from_dZ(dZout, linear_cache)
    grads['dW' + str(L - 1)] = dW
    grads['db' + str(L - 1)] = db

    # calculate gradients for all the previous layers using relu activation
    for l in range(L - 3, -1, -1):
        cache = caches[l]

        mask = dropout_masks[l]
        assert(mask.shape == dA_prev.shape)
        dA_prev = mask * dA_prev
        dA_prev = dA_prev / keep_prob

        dA_prev, dW, db = activation_backward_from_dA(dA_prev, cache, "relu")
        grads['dW' + str(l + 1)] = dW
```

```
        grads['db' + str(l + 1)] = db

    return grads
```

8. Update Parameters

8.1 Gradient Descent

We update the parameters of the model, using gradient descent:

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]} \tag{21}$$

$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]} \tag{22}$$

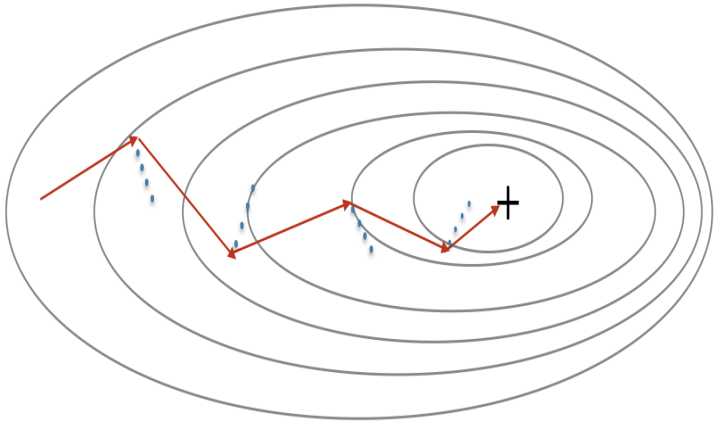
where α is the learning rate.

In [22]:
```
# Update the parameters including weight matrices and bias vectors

def update_params(params, grads, learning_rate):
    """
    Arguments:
    params: python dictionary containing weight matrix wl and bias vector bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWl'], ..., and grads['db1'], grads['db2'], ..., grads['dbl'], ...
    learning_rate: learning rate alpha

    Returns:
    updated params
    """
    L = len(params) // 2 + 1 # total number of layers including the input layer

    for l in range(1, L):
        params['W' + str(l)] -= learning_rate * grads['dW' + str(l)]
        params['b' + str(l)] -= learning_rate * grads['db' + str(l)]

    return params
```

8.2 Momentum

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 can also think of v as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.



The red arrows shows the direction taken by one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step. Rather than just following the gradient, we let the gradient influence v and then take a step in the direction of v.

The momentum update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases} \tag{23}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases} \tag{24}$$

where L is the number of layers, β is the momentum and α is the learning rate.

In [23]:
```
# Update the parameters including weight matrices and bias vectors, with Momentum optimization

def update_params_momentum(params, v, grads, learning_rate, beta):
    """
    Arguments:
    params: python dictionary containing weight matrix wl and bias vector bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
    v: python dictionary containing velocities for each layer,
            v['dW1'], v['dW2'], ..., v['dWl'], ..., v['db1'], v['db2'], ..., v['dbl'], ...
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWl'], ..., and grads['db1'], grads['db2'], ..., grads['dbl'], ...
    learning_rate: learning rate alpha
    beta: hyperparameter for momentum optimization

    Returns:
    updated params and velocities
    """
    L = len(params) // 2 + 1 # total number of layers including the input layer

    for l in range(1, L):
        v['dW' + str(l)] = beta * v['dW' + str(l)] + (1 - beta) * grads['dW' + str(l)]
        v['db' + str(l)] = beta * v['db' + str(l)] + (1 - beta) * grads['db' + str(l)]
        params['W' + str(l)] -= learning_rate * v['dW' + str(l)]
        params['b' + str(l)] -= learning_rate * v['db' + str(l)]

    return params, v
```

8.3 Adam Optimization

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp and Momentum.

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v^{corrected}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2) (\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s^{corrected}_{dW^{[l]}} = \frac{s_{dW^{[l]}}}{1 - (\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{dW^{[l]}}}{\sqrt{s^{corrected}_{dW^{[l]}}} + \varepsilon} \end{cases} \tag{25}$$

where:

- t counts the number of steps taken of Adam (starting from 1)
- 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

Some advantages of Adam include:

- Relatively low memory requirements (though higher than gradient descent and gradient descent with momentum)
- Usually works well even with little tuning of hyperparameters (except α)

In [24]:
```python
# Update the parameters including weight matrices and bias vectors, with Adam optimization

def update_params_adam(params, v, s, t, grads, learning_rate, beta1, beta2, epsilon):
    """
    Arguments:
    params: python dictionary containing weight matrix wl and bias vector bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
    v: python dictionary containing velocities for each layer,
            v['dW1'], v['dW2'], ..., v['dWl'], ..., v['db1'], v['db2'], ..., v['dbl'], ...
    s: python dictionary containing squared gradients for each layer,
            s['dW1'], s['dW2'], ..., s['dWl'], ..., s['db1'], s['db2'], ..., s['dbl'], ...
    t: current number of steps (starting from 1)
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWl'], ..., and grads['db1'], grads['db2'], ..., grads['bl'], ...
    learning_rate: learning rate alpha
    beta1, beta2: hyperparameters for Adam optimization
    epsilon: hyperparameter preventing division by zero in Adam updates

    Returns:
    updated params, exponentially weighted average of the past gradients and squares of the past gradients
    """
    L = len(params) // 2 + 1 # total number of layers including the input layer
    v_corrected = {}
    s_corrected = {}

    for l in range(1, L):
        v['dW' + str(l)] = beta1 * v['dW' + str(l)] + (1 - beta1) * grads['dW' + str(l)]
        v_corrected['dW' + str(l)] = v['dW' + str(l)] / (1 - beta1 ** t)

        v['db' + str(l)] = beta1 * v['db' + str(l)] + (1 - beta1) * grads['db' + str(l)]
        v_corrected['db' + str(l)] = v['db' + str(l)] / (1 - beta1 ** t)

        s['dW' + str(l)] = beta2 * s['dW' + str(l)] + (1 - beta2) * grads['dW' + str(l)] ** 2
        s_corrected['dW' + str(l)] = s['dW' + str(l)] / (1 - beta2 ** t)

        s['db' + str(l)] = beta2 * s['db' + str(l)] + (1 - beta2) * grads['db' + str(l)] ** 2
        s_corrected['db' + str(l)] = s['db' + str(l)] / (1 - beta2 ** t)

        params['W' + str(l)] -= learning_rate * v_corrected['dW' + str(l)] / (np.sqrt(s_corrected['dW' + str(l)]) + epsilon)
        params['b' + str(l)] -= learning_rate * v_corrected['db' + str(l)] / (np.sqrt(s_corrected['db' + str(l)]) + epsilon)

    return params, v, s
```

9. L-Layer Neural Network Complete Model

Now we use the helper functions implemented above to build an L-layer neural network with the following structure: *[LINEAR -> RELU]*$\times$*(L-1) -> LINEAR -> SIGMOID*.

Note:

- A **common mistake when using dropout** is to use it both in training and testing. We should use dropout (randomly eliminate nodes) only in training.

In [25]:
```
# Build the L-layer deep neural network model

def L_layer_model(X, Y, layer_dims, params_seed = 0, mini_batch_size = None, optimizer = "gd", learning_rate = 0.0075, beta =
0.9,
                  beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, iterations = 3300, print_cost_freq = 0, save_cost_freq = 100,
                  initialization = "he", regularization = "none", lambd = 0, keep_prob = 1, dropout_seed = 0):
    """
    Arguments:
    X: training set features, with the dimension of (number of features, number of examples)
    Y: training set labels, with the dimension of (1, number of examples)
    layer_dims: python array, layer_dims[l] is the number of units in the lth layer.
                l = 0 is the input layer, the last l is the output layer
    params_seed: random seed for initialization.
    mini_batch_size: size of each mini batch. By default it is the total number of examples, i.e., batch gradient descent
    optimizer: choose the optimization method:
               "gd": gradient descent
               "momentum": momentum optimization
               "adam": Adam optimization
    learning_rate: learning rate for gradient descent
    beta: hyperparameter for momentum optimization
    beta1, beta2: hyperparameters for Adam optimization
    epsilon: hyperparameter preventing division by zero in Adam updates
    iterations: number of iterations
    print_cost_freq: if > 0, print the cost value every print_cost_freq steps.
    save_cost_freq: save the cost value every save_cost_freq steps into costs, for ploting the learning curve.
    initialization: choose which initialization to use:
                "random_normal":  random values following standard normal distribution.
                "he": He initialization
                "xavier": Xavier initialization
    regularization: choose the regularization method:
                "none": no regularization
                "L2": L2 regularization
    lambd: the lambda parameter for L2 regularization
    keep_prob: keep_prob: parameter for inverted dropout,the probability of keeping a neuron. If it's 1, dropout is not used.
    dropout_seed: random seed for dropout.

    Returns:
    params: python dictionary containing weight matrix Wl and bias vector bl for the lth layer,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
            Wl has the dimension of (layer_dims[l], layer_dims[l - 1]).
            bl has the dimension of (layer_dims[l], 1)

    Note: in this code, dropout and L2 regularization are not used at the same time
    """
```

```
    assert(keep_prob == 1 or regularization == "none")
    assert(keep_prob <= 1)

    if mini_batch_size is None:
        mini_batch_size = X.shape[1]

    costs = []
    seed_mini_batches = 10 # random seed for creating mini batches
    t = 0 # count current number of steps

    # initialization
    params = init_params(layer_dims, initialization, params_seed)
    if optimizer == "gd":
        pass
    elif optimizer == "momentum":
        v = init_momentum(layer_dims)
    elif optimizer == "adam":
        v, s = init_adam(layer_dims)

    for i in range(1, iterations + 1):
        # create mini batches
        seed_mini_batches += 1
        mini_batches = random_mini_batches(X, Y, mini_batch_size, seed_mini_batches)

        for mini_batch in mini_batches:
            mini_batch_X, mini_batch_Y = mini_batch

            # forward propagation
            if keep_prob < 1:
                Aout, caches, dropout_masks = L_layer_forward_inverted_dropout(mini_batch_X, params, keep_prob, dropout_seed)
            else:
                Aout, caches = L_layer_forward(mini_batch_X, params)

            # cost
            if regularization == "L2":
                cost = cost_func_L2_Regul(Aout, mini_batch_Y, params, lambd)
            else:
                cost = cost_func(Aout, mini_batch_Y)

            # backward propagation
            if keep_prob < 1:
                grads = L_layer_backward_inverted_dropout(Aout, mini_batch_Y, caches, keep_prob, dropout_masks)
            elif regularization == "none":
                grads = L_layer_backward(Aout, mini_batch_Y, caches)
```

```
            elif regularization == "L2":
                grads = L_layer_backward_L2_Regul(Aout, mini_batch_Y, caches, lambd)

            # update weight matrices and bias vectors
            if optimizer == "gd":
                params = update_params(params, grads, learning_rate)
            elif optimizer == "momentum":
                params, v = update_params_momentum(params, v, grads, learning_rate, beta)
            elif optimizer == "adam":
                t += 1
                params, v, s = update_params_adam(params, v, s, t, grads, learning_rate, beta1, beta2, epsilon)

        if print_cost_freq > 0 and (i == 1 or i % print_cost_freq == 0):
            print("current iteration: " + str(i) + ", cost: " + str(cost))
        if save_cost_freq > 0 and (i == 1 or i % save_cost_freq == 0):
            costs.append(cost)

    # plot the cost
    plt.plot(costs)
    plt.ylabel('cost')
    plt.xlabel('iterations * ' + str(print_cost_freq))
    plt.title('learning rate = ' + str(learning_rate))
    plt.show()

    return params
```