Neural Networks and Deep Learning

- Understand the major trends driving the rise of deep learning.
- Be able to explain how deep learning is applied to supervised learning.
- Understand what are the major categories of models (such as CNNs and RNNs), and when they should be applied.
- Be able to recognize the basics of when deep learning will (or will not) work well.
- Build a logistic regression model, structured as a shallow neural network
- Implement the main steps of an ML algorithm, including making predictions, derivative computation, and gradient descent.
- Implement computationally efficient, highly vectorized, versions of models.
- Understand how to compute derivatives for logistic regression, using a backpropagation mindset.
- Become familiar with Python and Numpy
- Work with iPython Notebooks
- Be able to implement vectorization across multiple training examples
- Understand hidden units and hidden layers
- Be able to apply a variety of activation functions in a neural network.
- Build your first forward and backward propagation with a hidden layer
- Apply random initialization to your neural network
- Become fluent with Deep Learning notations and Neural Network Representations
- Build and train a neural network with one hidden layer.
- See deep neural networks as successive blocks put one after each other
- Build and train a deep L-layer Neural Network
- Analyze matrix and vector dimensions to check neural network implementations.
- Understand how to use a cache to pass information from forward propagation to back propagation.
- Understand the role of hyperparameters in deep learning

Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

- Recall that different types of initializations lead to different results
- Recognize the importance of initialization in complex neural networks.
- Recognize the difference between train/dev/test sets
- Diagnose the bias and variance issues in your model
- Learn when and how to use regularization methods such as dropout or L2 regularization.
- Understand experimental issues in deep learning such as Vanishing or Exploding gradients and learn how to deal with them
- Use gradient checking to verify the correctness of your backpropagation implementation
- Remember different optimization methods such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam
- Use random minibatches to accelerate the convergence and improve the optimization
- Know the benefits of learning rate decay and apply it to your optimization
- Master the process of hyperparameter tuning

Session 1 - Logistic Regression with a Neural Network mindset

You will learn to:

- Build the general architecture of a learning algorithm, including:
  - Initializing parameters
  - Calculating the cost function and its gradient
  - Using an optimization algorithm (gradient descent)
- Gather all three functions above into a main model function, in the right order.

Many software bugs in deep learning come from having matrix/vector dimensions that don't fit. If you can keep your matrix/vector dimensions straight you will go a long way toward eliminating many bugs.

- m_train (number of training examples)
- m_test (number of test examples)
- num_px (= height = width of a training image)

Remember that `train_set_x_orig` is a numpy-array of shape (m_train, num_px, num_px, 3). For instance, you can access `m_train` by writing `train_set_x_orig.shape[0]`.

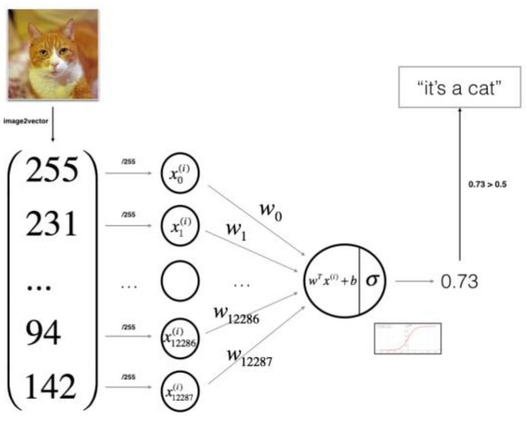
```
- a training set of m_train images labeled as cat (y=1) or non-cat (y=0)
- a test set of m_test images labeled as cat or non-cat - each image is
of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB). Thus,
each image is square (height = num_px) and (width = num_px).
```

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.

One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. 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 (the maximum value of a pixel channel).

<u>Common steps for pre-processing a new dataset are:</u>

- Figure out the dimensions and shapes of the problem (m_train, m_test, num_px, ...)
- Reshape the datasets such that each example is now a vector of size (num_px * num_px * 3, 1)
- "Standardize" the data



Following steps:

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

Building the parts of our algorithm

The main steps for building a Neural Network are:

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)

You often build 1-3 separately and integrate them into one function we call `model()`.

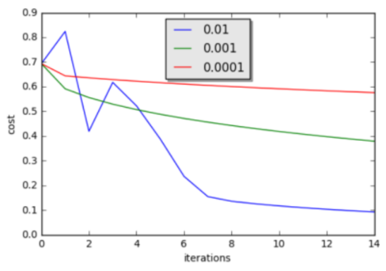
Forward and Backward propagation

Now that your parameters are initialized, you can do the "forward" and "backward" propagation steps for learning the parameters.

Optimization:

- You have initialized your parameters.
- You are also able to compute a cost function and its gradient.
- Now, you want to update the parameters using gradient descent.
  - Initialize (w,b)
  - Optimize the loss iteratively to learn parameters (w,b):
    - computing the cost and its gradient
    - updating the parameters using gradient descent
  - Use the learned (w,b) to predict the labels for a given set of examples

Reminder: In order for Gradient Descent to work you must choose the learning rate wisely. The learning rate α determines how rapidly we update the parameters. If the learning rate is too large we may "overshoot" the optimal value. Similarly, if it is too small we will need too many iterations to converge to the best values. That's why it is crucial to use a well-tuned learning rate.



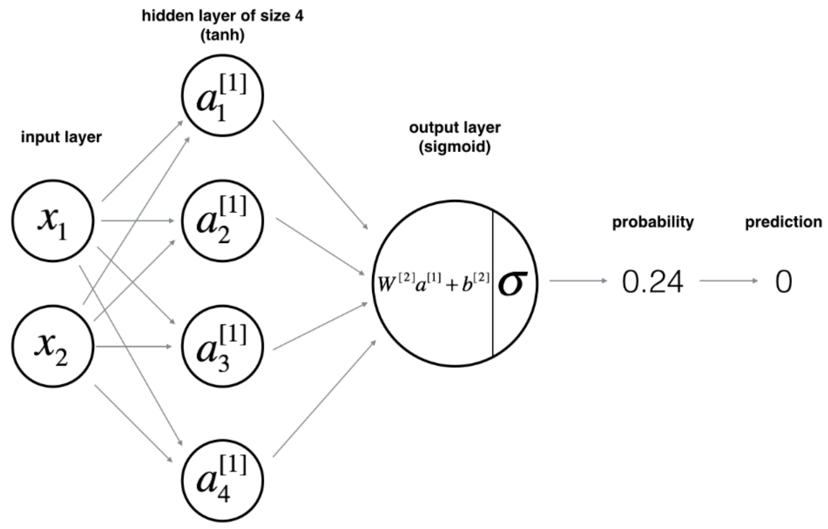
Interpretation:

- Different learning rates give different costs and thus different predictions results.
- If the learning rate is too large (0.01), the cost may oscillate up and down. It may even diverge (in this example, using 0.01 still evtl ends up at a good value for the cost).
- A lower cost doesn't mean a better model. You have to check if there is possibly overfitting. It happens when the training accuracy is a lot higher than the test accuracy.
- In deep learning, we usually recommend that you:
  - Choose the learning rate that better minimizes the cost function.
  - If your model overfits, use other techniques to reduce overfitting.

Session 2 - Planar data classification with one hidden layer

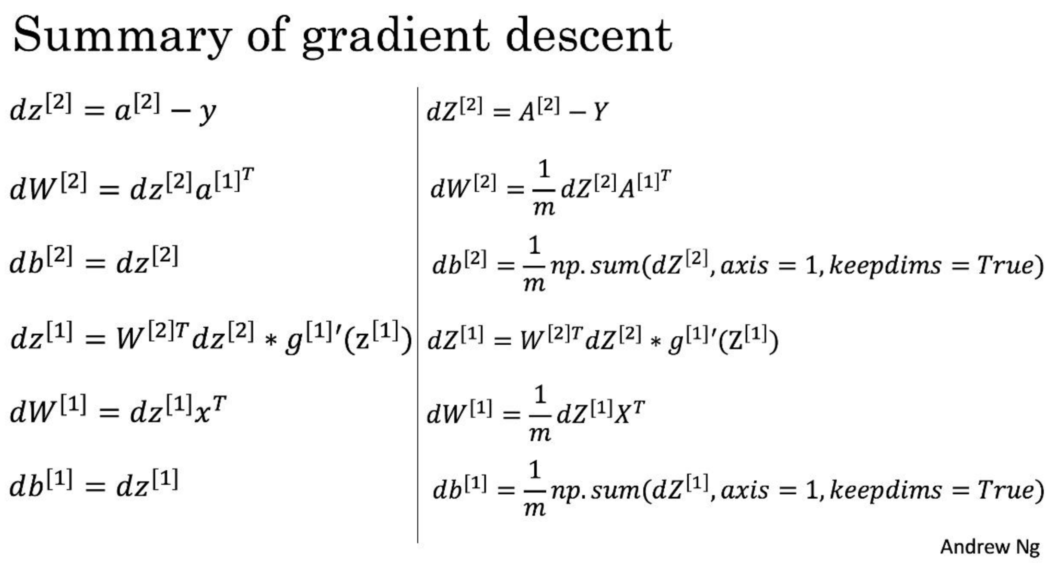
Neural Network model

If logistic regression don’t work well, train a Neural Network with a single hidden layer.



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

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.



Summary of gradient descent

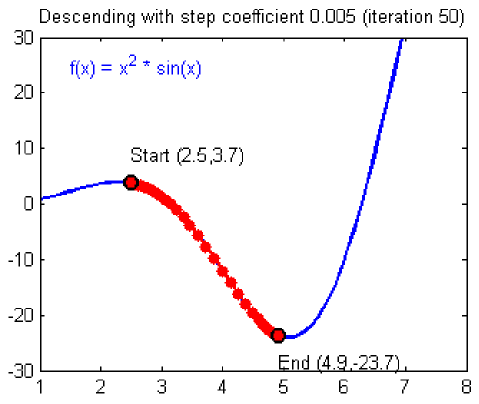
$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]}a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]}A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T}dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T}dZ^{[2]} *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

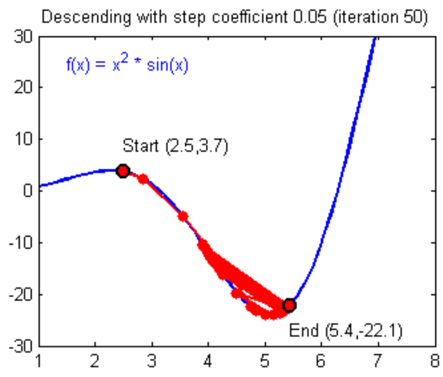
Andrew Ng

Question: Implement the update rule. Use gradient descent. You have to use (dW1, db1, dW2, db2) in order to update (W1, b1, W2, b2).

General gradient descent rule: $\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$ where *alpha* is the learning rate and *theta* represents a parameter.

Illustration: The gradient descent algorithm with a good learning rate (converging) and a bad learning rate (diverging). Images courtesy of Adam Harley.





Interpretation:

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

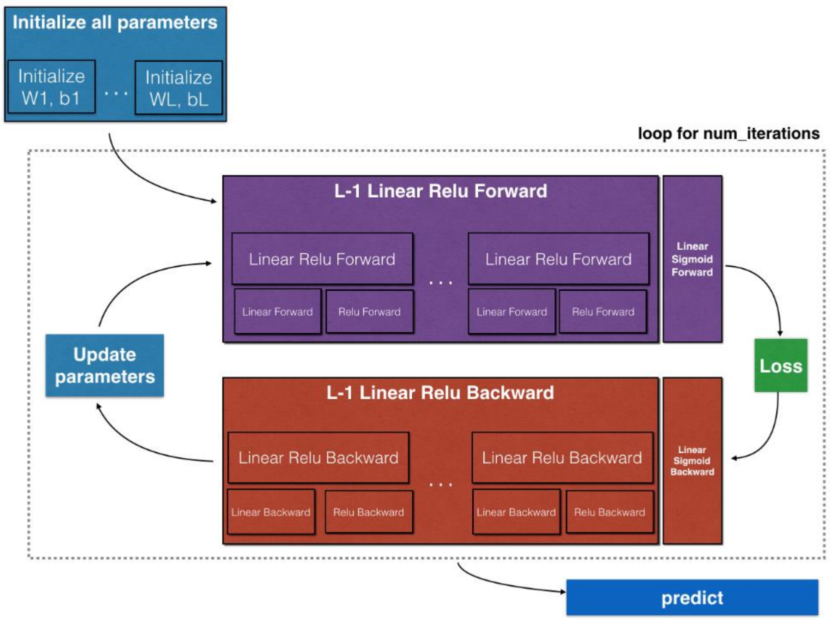
Session 3 - Building your Deep Neural Network: Step by Step

After this assignment you will be able to:

- Use non-linear units like ReLU to improve your model
- Build a deeper neural network (with more than 1 hidden layer)
- Implement an easy-to-use neural network class

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).



L-layer Neural Network

The initialization for a deeper L-layer neural network is more complicated because there are many more weight matrices and bias vectors. When completing the `initialize_parameters_deep`, you should make sure that your 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	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
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:

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

Then $WX + b$ will be:

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

Exercise: Implement initialization for an L-layer Neural Network.

Instructions:

- The model's structure is *[LINEAR -> RELU]* $\times$ *(L-1) -> LINEAR -> SIGMOID*. I.e., it has $L-1$ layers using a ReLU activation function followed by an output layer with a sigmoid activation function.
- Use random initialization for the weight matrices. Use `np.random.rand(shape) * 0.01`.
- Use zeros initialization for the biases. Use `np.zeros(shape)`.
- 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. Thus 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!
- Here is the implementation for $L = 1$ (one layer neural network). It should inspire you to implement the general case (L-layer neural network).

Linear Activation Forward

In this notebook, you will use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1 + e^{-(WA + b)}}$. We have provided you with the `sigmoid` function. This function returns **two** items: the activation value "a" and a "`cache`" that contains "`Z`" (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = sigmoid(Z)
```

- **ReLU**: The mathematical formula for ReLu is $A = RELU(Z) = max(0, Z)$. We have provided you with the `relu` function. This function returns **two** items: the activation value "`A`" and a "`cache`" that contains "`Z`" (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = relu(Z)
```

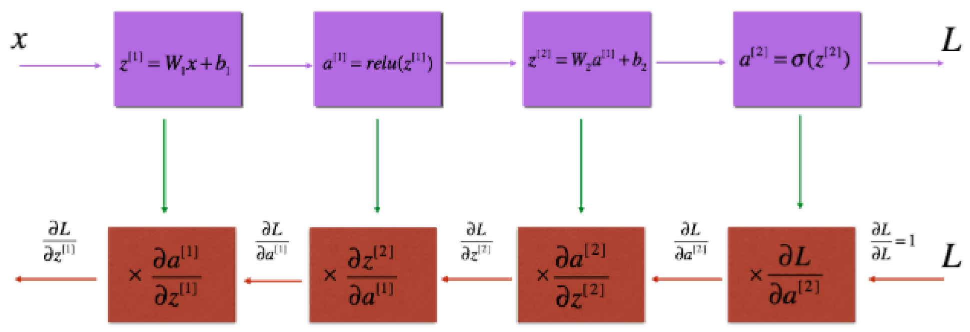
For more convenience, you are going to group two functions (Linear and Activation) into one function (LINEAR->ACTIVATION). Hence, you will implement a function that does the LINEAR forward step followed by an ACTIVATION forward step.

Exercise: Implement the forward propagation of the *LINEAR->ACTIVATION* layer. 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(). Use linear_forward() and the correct activation function.

Backward Propagation Module

Just like with forward propagation, you will implement helper functions for backpropagation. Remember that back propagation is used to calculate the gradient of the loss function with respect to the parameters.

Reminder:


x
$z^{[1]} = W_1 x + b_1$
$a^{[1]} = relu(z^{[1]})$
$z^{[2]} = W_2 a^{[1]} + b_2$
$a^{[2]} = \sigma(z^{[2]})$
L
$\frac{\partial L}{\partial z^{[1]}}$
$\times \frac{\partial a^{[1]}}{\partial z^{[1]}}$
$\frac{\partial L}{\partial a^{[1]}}$
$\times \frac{\partial z^{[2]}}{\partial a^{[1]}}$
$\frac{\partial L}{\partial z^{[2]}}$
$\times \frac{\partial a^{[2]}}{\partial z^{[2]}}$
$\frac{\partial L}{\partial a^{[2]}}$
$\times \frac{\partial L}{\partial a^{[2]}}$
$\frac{\partial L}{\partial L} = 1$
L

Now, similar to forward propagation, you are going to build the backward propagation in three steps:

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

Linear-Activation backward

Next, you will create a function that merges the two helper functions: **`linear_backward`** and the backward step for the activation **`linear_activation_backward`**.

To help you implement `linear_activation_backward`, we provided two backward functions:

- **`sigmoid_backward`**: Implements the backward propagation for SIGMOID unit. You can call it as follows:

```
dZ = sigmoid_backward(dA, activation_cache)
```

- **`relu_backward`**: Implements the backward propagation for RELU unit. You can call it as follows:

```
dZ = relu_backward(dA, activation_cache)
```

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

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

Linear-Activation backward

Initializing backpropagation: To backpropagate through this network, we know that the output is, $A^{[L]} = \sigma(Z^{[L]})$. Your code thus needs to compute `dAL` $= \frac{\partial \mathcal{L}}{\partial A^{[L]}}$. To do so, use this formula (derived using calculus which you don't need in-depth knowledge of):

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) # derivative of cost with respect to AL
```

You can then use this post-activation gradient `dAL` to keep going backward. As seen in Figure 5, you can now feed in `dAL` into the LINEAR->SIGMOID backward function you implemented (which will use the cached values stored by the L_model_forward function). After that, you will have to use a `for` loop to iterate through all the other layers using the LINEAR->RELU backward function. You should store each dA, dW, and db in the grads dictionary. To do so, use this formula :

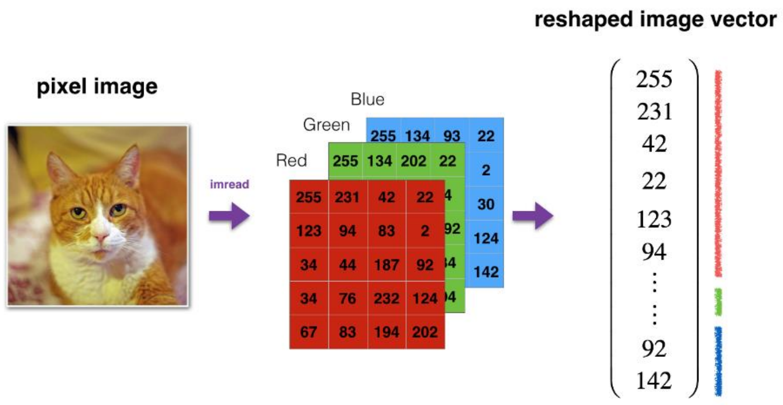
$$grads[\text{Unknown character}dW\text{Unknown character} + str(l)] = dW^{[l]}$$

For example, for $l = 3$ this would store $dW^{[l]}$ in `grads["dW3"]`.

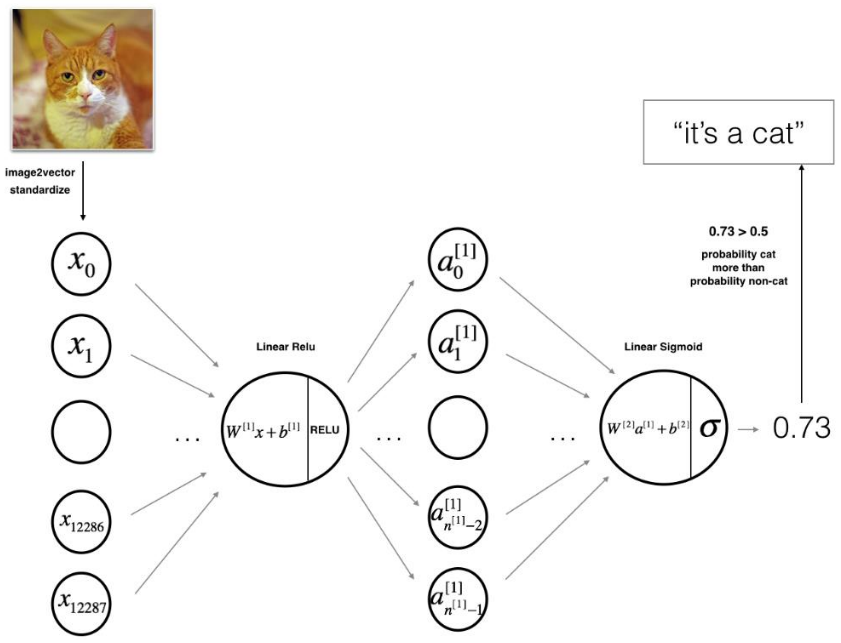
Session 4 - Deep Neural Network for Image Classification

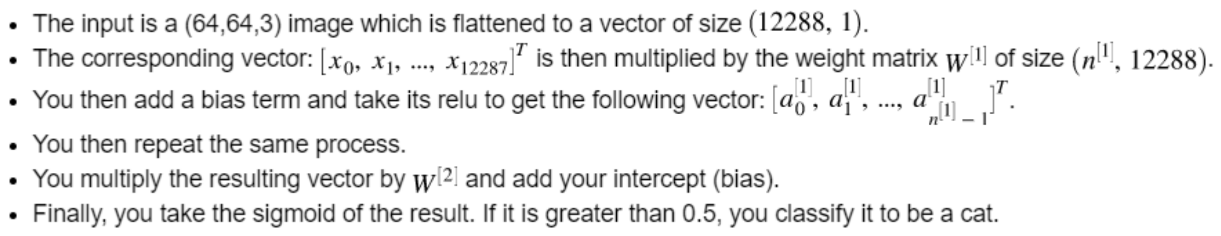
- Build and apply a deep neural network to supervised learning (2- an L-layer)

As usual, you reshape and standardize the images before feeding them to the network.



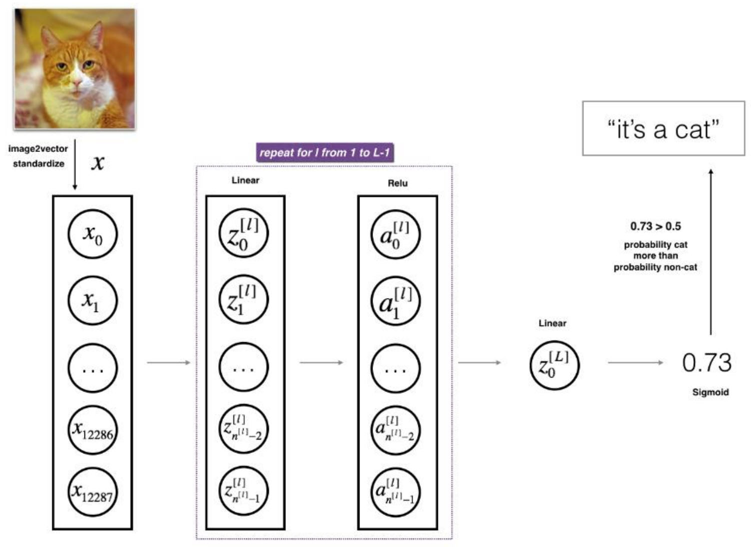
2-layer neural network





- The input is a (64,64,3) image which is flattened to a vector of size $(12288, 1)$.
- The corresponding vector: $[x_0, x_1, ..., x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ of size $(n^{[1]}, 12288)$.
- You then add a bias term and take its relu to get the following vector: $[a_0^{[1]}, a_1^{[1]}, ..., a_{n^{[1]}-1}^{[1]}]^T$.
- 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.

L-layer deep neural network



General methodology

As usual you will follow the Deep Learning 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

A few type of images the model tends to do poorly on include:

- Cat body in an unusual position
- Cat appears against a background of a similar color
- Unusual cat color and species
- Camera Angle
- Brightness of the picture
- Scale variation (cat is very large or small in image)

Session 5 – Initialization

A well chosen initialization can:

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

- *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.

Zero initialization

There are two types of parameters to initialize in a neural network:

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

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 you might as well be training a neural network with n^[l] = 1 for every layer, and the network is no more powerful than a linear classifier such as logistic regression:

What you should remember:

- The weights W^[l] should be initialized randomly to break symmetry.
- It is however okay to initialize the biases b^[l] to zeros. Symmetry is still broken so long as W^[l] is initialized randomly.

Random initialization

To break symmetry, lets intialize the weights randomly. Following random initialization, each neuron can then proceed to learn a different function of its inputs. In this exercise, you will see what happens if the weights are intialized randomly, but to very large values.

Implement the following function to initialize your weights to large random values (scaled by *10) and your biases to zeros. Use np.random.randn(..,..) * 10 for weights and np.zeros((.., ..)) for biases. We are using a fixed np.random.seed(..) to make sure your "random" weights match ours, so don't worry if running several times your code gives you always the same initial values for the parameters.

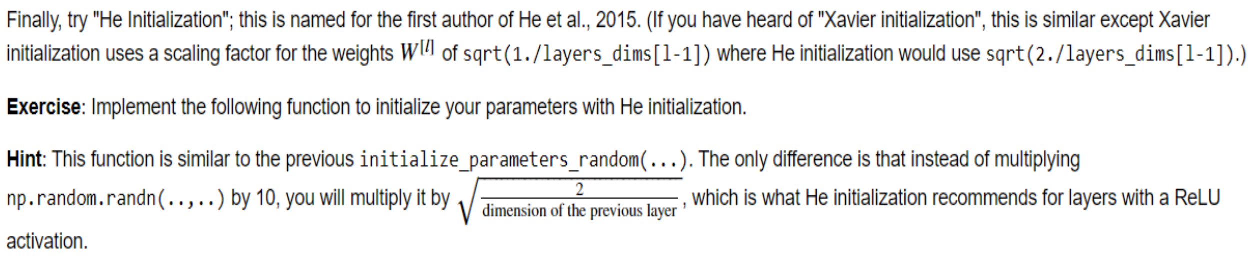
Observations:

- The cost starts very high. This is because with large random-valued weights, the last activation (sigmoid) outputs results that are very close to 0 or 1 for some examples, and when it gets that example wrong it incurs a very high loss for that example. Indeed, when $\log(a^{[3]}) = \log(0)$, the loss goes to infinity.
- Poor initialization can lead to vanishing/exploding gradients, which also slows down the optimization algorithm.
- If you train this network longer you will see better results, but initializing with overly large random numbers slows down the optimization.

In summary:

- Initializing weights to very large random values does not work well.
- Hopefully intializing with small random values does better. The important question is: how small should be these random values be? Lets find out in the next part!

He initialization



Finally, try "He Initialization"; this is named for the first author of He et al., 2015. (If you have heard of "Xavier initialization", this is similar except Xavier initialization uses a scaling factor for the weights $W^{[l]}$ of `sqrt(1./layers_dims[l-1])` where He initialization would use `sqrt(2./layers_dims[l-1])`.)

Exercise: Implement the following function to initialize your parameters with He initialization.

Hint: This function is similar to the previous `initialize_parameters_random(...)`. The only difference is that instead of multiplying `np.random.randn(..,..)` by 10, you will multiply it by $\sqrt{\frac{2}{\text{dimension of the previous layer}}}$, which is what He initialization recommends for layers with a ReLU activation.

Observations:

- The model with He initialization separates the blue and the red dots very well in a small number of iterations.

You have seen three different types of initializations. For the same number of iterations and same hyperparameters the comparison is:

Model	Train accuracy	Problem/Comment
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

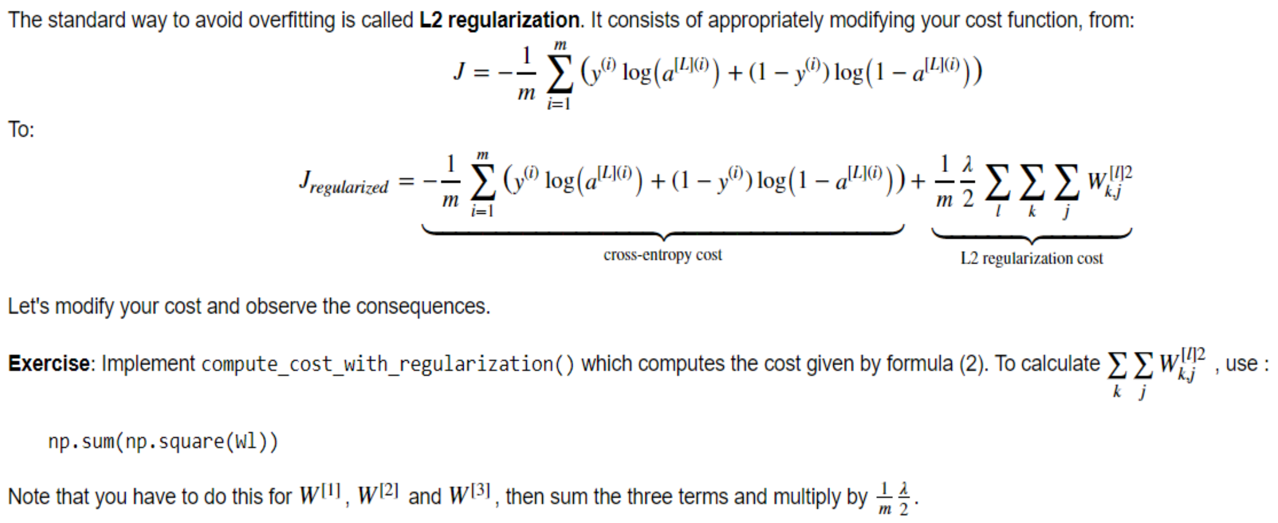
What you should remember from this notebook:

- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large
- He initialization works well for networks with ReLU activations.

Session 5 – Regularization

A non-regularized model can overfitting the training set. It is fitting the noisy points! Lets now look at two techniques to reduce overfitting.

L2 Regularization



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

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

To:

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

Let's modify your cost and observe the consequences.

Exercise: Implement `compute_cost_with_regularization()` which computes the cost given by formula (2). To calculate $\sum_k \sum_j W_{k,j}^{[l]2}$, use :

```
np.sum(np.square(Wl))
```

Note that you have to do this for $W^{[1]}$, $W^{[2]}$ and $W^{[3]}$, then sum the three terms and multiply by $\frac{1}{m} \frac{\lambda}{2}$.

Observations:

- The value of λ is a hyperparameter that you can tune using a dev set.
- 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.

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 you 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.

What you should remember -- the implications of L2-regularization on:

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

Dropout

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−keep_prob1−keep_prob or keep it with probability keep_probkeep_prob (50% here). 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.

Forward propagation with dropout

Instructions: You would like to shut down some neurons in the first and second layers. To do that, you are going to carry out 4 Steps:

1. In lecture, we dicussed creating a variable d[1] with the same shape as a[1] using np.random.rand() to randomly get numbers between 0 and 1. Here, you will use a vectorized implementation, so create a random matrix D[1]=[d[1](1)d[1](2)...d[1](m)] of the same dimension as A[1].
2. Set each entry of D[1] to be 1 with probability (keep_prob), and 0 otherwise.

Hint: Let's say that keep_prob = 0.8, which means that we want to keep about 80% of the neurons and drop out about 20% of them. We want to generate a vector that has 1's and 0's, where about 80% of them are 1 and about 20% are 0. This python statement:
X = (X < keep_prob).astype(int) is conceptually the same as this if-else statement (for the simple case of a one-dimensional array):

```
for i,v in enumerate(x):
  if v < keep_prob:
    x[i] = 1
  else: # v >= keep_prob
    x[i] = 0
```

Note that the X = (X < keep_prob).astype(int) works with multi-dimensional arrays, and the resulting output preserves the dimensions of the input array.
Also note that without using .astype(int), the result is an array of booleans True and False, which Python automatically converts to 1 and 0 if we multiply it with numbers. (However, it's better practice to convert data into the data type that we intend, so try using .astype(int).)

1. Set A[1] to A[1]∗D[1]. (You are shutting down some neurons). You can think of D[1] as a mask, so that when it is multiplied with another matrix, it shuts down some of the values.
2. Divide A[1] by keep_prob. By doing this you 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).

Backward propagation with dropout

Exercise: Implement the backward propagation with dropout. As before, you 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.

Instruction: Backpropagation with dropout is actually quite easy. You will have to carry out 2 Steps:

1. You had previously shut down some neurons during forward propagation, by applying a mask D[1] to A1. In backpropagation, you will have to shut down the same neurons, by reapplying the same mask D[1] to dA1.

2. During forward propagation, you had divided A1 by keep_prob. In backpropagation, you'll therefore have to divide dA1 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).

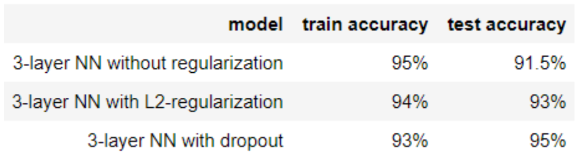
Note:

- A **common mistake** when using dropout is to use it both in training and testing. You should use dropout (randomly eliminate nodes) only in training.
- Deep learning frameworks like tensorflow, PaddlePaddle, keras or caffe come with a dropout layer implementation. Don't stress - you will soon learn some of these frameworks.

What you should remember about dropout:

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

Here are the results of our three models:



model	train accuracy	test accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

Note that regularization hurts training set performance! This is because it limits the ability of the network to overfit to the training set. But since it ultimately gives better test accuracy, it is helping your system.

What we want you to remember from this notebook:

- Regularization will help you reduce overfitting.
- Regularization will drive your weights to lower values.
- L2 regularization and Dropout are two very effective regularization techniques.

Session 6 – Gradient Checking

How does gradient checking work?

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.

Because forward propagation is relatively easy to implement, you're confident you got that right, and so you're almost 100% sure that you're computing the cost J correctly. Thus, 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):

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon} \tag{1}$$

If you're not familiar with the "$\lim_{\varepsilon \to 0}$" notation, it's just a way of saying "when ε is really really small."

We know the following:

- $\frac{\partial J}{\partial \theta}$ is what you want to make sure you're computing correctly.
- You can compute $J(\theta + \varepsilon)$ and $J(\theta - \varepsilon)$ (in the case that θ is a real number), since you're confident your implementation for J is correct.

Lets use equation (1) and a small value for ε to convince your CEO that your code for computing $\frac{\partial J}{\partial \theta}$ is correct!

1-dimensional gradient checking

Consider a 1D linear function $J(\theta) = \theta x$. The model contains only a single real-valued parameter θ, and takes x as input.

You will implement code to compute $J(.)$ and its derivative $\frac{\partial J}{\partial \theta}$. You will then use gradient checking to make sure your derivative computation for J is correct.

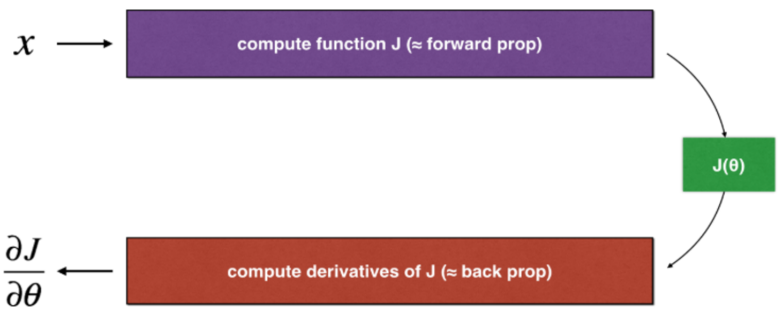

x
compute function J (≈ forward prop)
J(θ)
compute derivatives of J (≈ back prop)
$\frac{\partial J}{\partial \theta}$

Figure 1 : **1D linear model**

The diagram above shows the key computation steps: First start with x, then evaluate the function $J(x)$ ("forward propagation"). Then compute the derivative $\frac{\partial J}{\partial \theta}$ ("backward propagation").

Exercise: To show that the `backward_propagation()` function is correctly computing the gradient $\frac{\partial J}{\partial \theta}$, let's implement gradient checking.

Instructions:

- First compute "gradapprox" using the formula above (1) and a small value of ε. Here are the Steps to follow:
  1. $\theta^{+} = \theta + \varepsilon$
  2. $\theta^{-} = \theta - \varepsilon$
  3. $J^{+} = J(\theta^{+})$
  4. $J^{-} = J(\theta^{-})$
  5. $gradapprox = \frac{J^{+} - J^{-}}{2\varepsilon}$
- Then compute the gradient using backward propagation, and store the result in a variable "grad"
- Finally, compute the relative difference between "gradapprox" and the "grad" using the following formula:
$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2}$$
  You will need 3 Steps to compute this formula:
  - 1'. compute the numerator using np.linalg.norm(...)
  - 2'. compute the denominator. You will need to call np.linalg.norm(...) twice.
  - 3'. divide them.
- If this difference is small (say less than 10^{-7}), you can be quite confident that you have computed your gradient correctly. Otherwise, there may be a mistake in the gradient computation.

Congrats if the difference is smaller than the 10−7 threshold. So you can have high confidence that you've correctly computed the gradient in backward_propagation().
Now, in the more general case, your cost function JJ has more than a single 1D input. When you are training a neural network, θ actually consists of multiple matrices W[l] and biases b[l]! It is important to know how to do a gradient check with higher-dimensional inputs.

N-dimensional gradient checking

The following figure describes the forward and backward propagation of your fraud detection model.

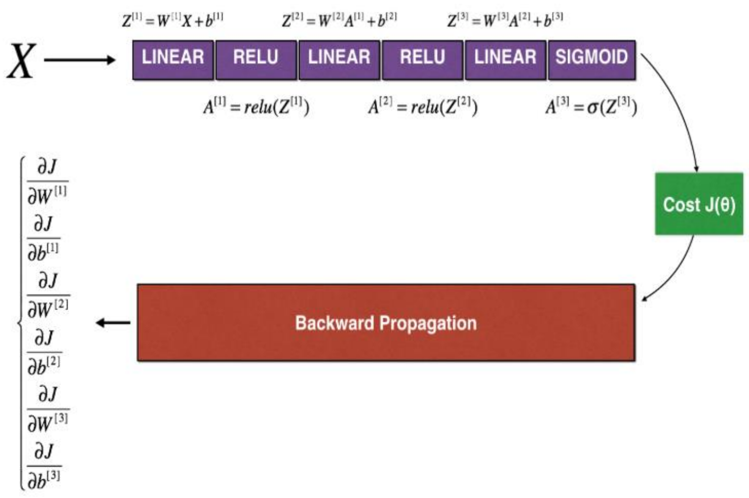

$Z^{[1]} = W^{[1]}X + b^{[1]}$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$
X
LINEAR
RELU
LINEAR
RELU
LINEAR
SIGMOID
$A^{[1]} = relu(Z^{[1]})$
$A^{[2]} = relu(Z^{[2]})$
$A^{[3]} = \sigma(Z^{[3]})$
Cost J(θ)
Backward Propagation
$\frac{\partial J}{\partial W^{[1]}}$ $\frac{\partial J}{\partial b^{[1]}}$ $\frac{\partial J}{\partial W^{[2]}}$ $\frac{\partial J}{\partial b^{[2]}}$ $\frac{\partial J}{\partial W^{[3]}}$ $\frac{\partial J}{\partial b^{[3]}}$

Figure 2 : **deep neural network**
LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID

How does gradient checking work?.

As in 1) and 2), you want to compare "gradapprox" to the gradient computed by backpropagation. The formula is still:

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

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 (W1, b1, W2, b2, W3, b3) into vectors and concatenating them.

The inverse function is "vector_to_dictionary" which outputs back the "parameters" dictionary.

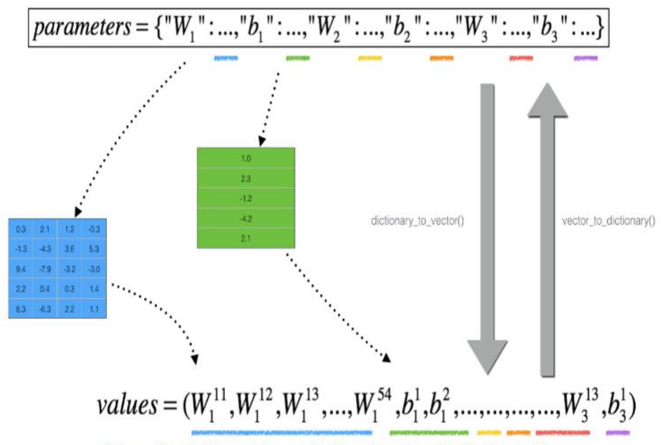

$parameters = \{"W_1":...,"b_1":...,"W_2":...,"b_2":...,"W_3":...,"b_3":...\}$
dictionary_to_vector()
vector_to_dictionary()
$values = (W_1^{11}, W_1^{12}, W_1^{13}, ..., W_1^{54}, b_1^1, b_1^2, ..., ..., ..., ..., W_3^{13}, b_3^1)$

Figure 2 : **dictionary_to_vector() and vector_to_dictionary()**
You will need these functions in gradient_check_n()

We have also converted the "gradients" dictionary into a vector "grad" using gradients_to_vector(). You don't need to worry about that.

Exercise: Implement gradient_check_n().

Instructions: Here is pseudo-code that will help you implement the gradient check.

For each i in num_parameters:

- To compute `J_plus[i]`:
  1. Set θ^{+} to `np.copy(parameters_values)`
  2. Set θ_i^{+} to $\theta_i^{+} + \varepsilon$
  3. Calculate J_i^{+} using to `forward_propagation_n(x, y, vector_to_dictionary(`θ^{+} `))`.
- To compute `J_minus[i]`: do the same thing with θ^{-}
- Compute $gradapprox[i] = \frac{J_i^{+} - J_i^{-}}{2\varepsilon}$

Thus, you get a vector gradapprox, where gradapprox[i] is an approximation of the gradient with respect to `parameter_values[i]`. You can now compare this gradapprox vector to the gradients vector from backpropagation. Just like for the 1D case (Steps 1', 2', 3'), compute:

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2} \tag{3}$$

Note

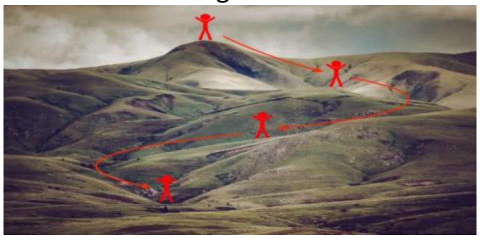
- Gradient Checking is slow! Approximating the gradient with $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$ is computationally costly. For this reason, we don't run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.
- Gradient Checking, at least as we've presented it, 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.

What you should remember from this notebook:

- Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation).
- Gradient checking is slow, so we don't run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.

Session 7 – Optimization Methods

Until now, you've always used Gradient Descent to update the parameters and minimize the cost. In this notebook, 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. Gradient descent goes "downhill" on a cost function J. Think of it as trying to do this:



At each step of the training, you update your parameters following a certain direction to try to get to the lowest possible point. **Notations**: As usual, ∂J/∂a = da for any variable a.

Gradient Descent

A simple optimization method in machine learning is gradient descent (GD). When you take gradient steps with respect to all mm examples on each step, it is also called Batch Gradient Descent.
Warm-up exercise: Implement the gradient descent update rule. The gradient descent rule is, for l=1,...,L:

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

where L is the number of layers and αα is the learning rate. All parameters should be stored in the parameters dictionary. Note that the iterator l starts at 0 in the for loop while the first parameters are W[1] and b[1]. You need to shift l to l+1 when coding.

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. The update rule that you have just implemented does not change. What changes is that you would be computing gradients on just one training example at a time, rather than on the whole training set. The code examples below illustrate the difference between stochastic gradient descent and (batch) gradient descent.

- **(Batch) Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost += compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

- **Stochastic Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost += compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

In Stochastic Gradient Descent, you use only 1 training example before updating the gradients. When the training set is large, SGD can be faster. But the parameters will "oscillate" toward the minimum rather than converge smoothly. Here is an illustration of this:

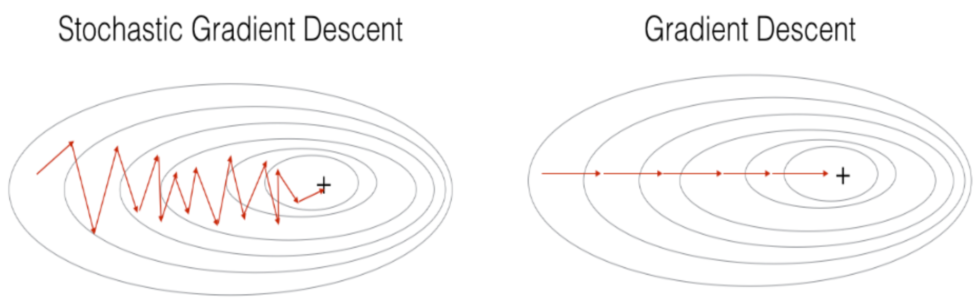


Figure 1 : **SGD vs GD**
"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

Note also that 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 $(W^{[L]}, b^{[L]})$)

In practice, you'll often get faster results if you do not 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 mini-batch gradient descent, you loop over the mini-batches instead of looping over individual training examples.

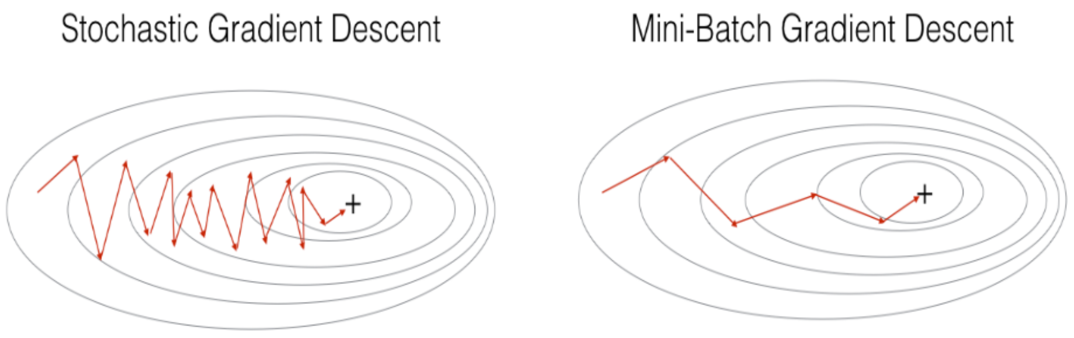


Figure 2 : **SGD vs Mini-Batch GD**
"+" denotes a minimum of the cost. Using mini-batches in your optimization algorithm often leads to faster optimization.

What you should remember:

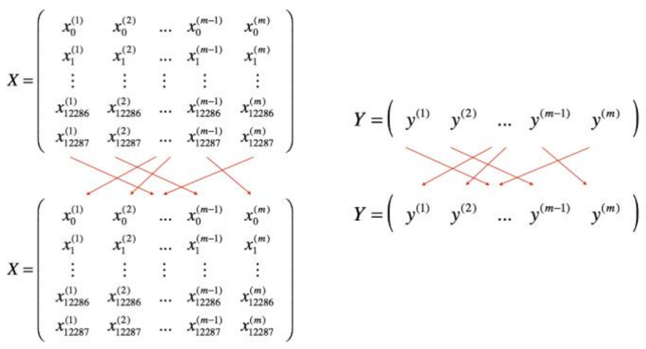
- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter αα.
- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

Mini-Batch Gradient Descent

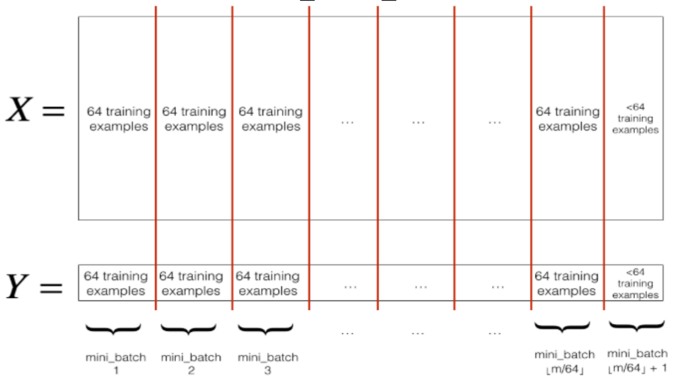
Let's learn how to build mini-batches from the training set (X, Y).

There are two steps:

- **Shuffle**: Create a shuffled version of the training set (X, Y) as shown below. Each column of X and Y represents a training example. Note that the random shuffling is done synchronously between X and Y. Such that after the shuffling the ith column of X is the example corresponding to the ith 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 (here 64). 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. When the final mini-batch is smaller than the full mini_batch_size, it will look like this:



Exercise: Implement `random_mini_batches`. We coded the shuffling part for you. To help you with the partitioning step, we give you the following code that selects the indexes for the 1^{st} and 2^{nd} mini-batches:

```
first_mini_batch_X = shuffled_X[:, 0 : mini_batch_size]
second_mini_batch_X = shuffled_X[:, mini_batch_size : 2 * mini_batch_size]
...
```

Note that the last mini-batch might end up smaller than `mini_batch_size=64`. Let $\lfloor s \rfloor$ represents s rounded down to the nearest integer (this is `math.floor(s)` in Python). If the total number of examples is not a multiple of `mini_batch_size=64` then there will be $\lfloor \frac{m}{mini\_batch\_size} \rfloor$ mini-batches with a full 64 examples, and the number of examples in the final mini-batch will be $(m - mini\_batch\_size \times \lfloor \frac{m}{mini\_batch\_size} \rfloor)$.

What you should remember:

- Shuffling and Partitioning are the two steps required to build mini-batches
- Powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

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. You 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.

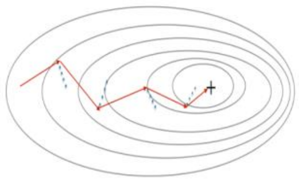


Figure 3: The red arrows shows the direction taken by one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step. Rather than just following the gradient, we let the gradient influence v and then take a step in the direction of v.

Exercise: Initialize the velocity. The velocity, vv, is a python dictionary that needs to be initialized with arrays of zeros. Its keys are the same as those in the grads dictionary, that is: for l=1, ..., L:

v["dW" **+** str(l**+**1)] = ... *#(numpy array of zeros with same shape as parameters["W" + str(l+1)])*
v["db" **+** str(l**+**1)] = ... *#(numpy array of zeros with same shape as parameters["b" + str(l+1)])*

Note that the iterator l starts at 0 in the for loop while the first parameters are v["dW1"] and v["db1"] (that's a "one" on the superscript). This is why we are shifting l to l+1 in the for loop.
Exercise: Now, implement the parameters update with momentum. The momentum update rule is, for l=1, ..., 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 αα is the learning rate. All parameters should be stored in the `parameters` dictionary. Note that the iterator `l` starts at 0 in the `for` loop while the first parameters are W[1] and b[1] (that's a "one" on the superscript). So you will need to shift `l` to `l+1` when coding.

Note that:

- The velocity is initialized with zeros. So the algorithm will take a few iterations to "build up" velocity and start to take bigger steps.
- If β=0, then this just becomes standard gradient descent without momentum.

How do you choose β?

- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, β=0.9 is often a reasonable default.
- Tuning the optimal β for your model might need trying several values to see what works best in term of reducing the value of the cost function J.

What you should remember:

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.
- You have to tune a momentum hyperparameter β and a learning rate α.

Adam

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp (described in lecture) 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 vcorrected (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables ss (before bias correction) and scorrected (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is l = 1, ..., L:

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

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β1 and β2 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

As usual, we will store all parameters in the parameters dictionary

Model with different optimization algorithms

Mini-Batch Gradient Descent with Momentum

Because this example is relatively simple, the gains from using momemtum are small; but for more complex problems you might see bigger gains.

Mini-Batch Gradient Descent with Adam Mode

Summary

optimization method	accuracy	cost shape
Gradient descent	79.7%	oscillations
Momentum	79.7%	oscillations
Adam	94%	smoother

Momentum usually helps, but given the small learning rate and the simplistic dataset, its impact is almost negligeable. Also, the huge oscillations you see in the cost come from the fact that some minibatches are more difficult thans 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.

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)

Session 8 – TensorFlow

Machine learning frameworks like TensorFlow, PaddlePaddle, Torch, Caffe, Keras, and many others can speed up your machine learning development significantly. All of these frameworks also have a lot of documentation, which you should feel free to read. In this assignment, you will learn to do the following in TensorFlow:

- Initialize variables
- Start your own session
- Train algorithms
- Implement a Neural Network

Programing frameworks can not only shorten your coding time, but sometimes also perform optimizations that speed up your code.

Exploration

Writing and running programs in TensorFlow has the following steps:

1. Create Tensors (variables) that are not yet executed/evaluated.
2. Write operations between those Tensors.
3. Initialize your Tensors.
4. Create a Session.
5. Run the Session. This will run the operations you'd written above.

Therefore, when we created a variable for the loss, we simply defined the loss as a function of other quantities but did not evaluate its value. To evaluate it, we had to run init=tf.global_variables_initializer(). That initialized the loss variable, and in the last line we were finally able to evaluate the value of loss and print its value.

To summarize, **remember to initialize your variables, create a session and run the operations inside the session**.

Next, you'll also have to know about placeholders. A placeholder is an object whose value you can specify only later. To specify values for a placeholder, you can pass in values by using a "feed

dictionary" (feed_dict variable). Below, we created a placeholder for x. This allows us to pass in a number later when we run the session.
When you first defined x you did not have to specify a value for it. A placeholder is simply a variable that you will assign data to only later, when running the session. We say that you **feed data** to these placeholders when running the session.
Here's what's happening: When you specify the operations needed for a computation, you are telling TensorFlow how to construct a computation graph. The computation graph can have some placeholders whose values you will specify only later. Finally, when you run the session, you are telling TensorFlow to execute the computation graph.

Linear Function

Let's start this programming exercise by computing the following equation: Y=WX+b,
where W and X are random matrices and b is a random vector.
Exercise: Compute WX+b where W, X and b are drawn from a random normal distribution. W is of shape (4, 3), X is (3,1) and b is (4,1). As an example, here is how you would define a constant X that has shape (3,1):

```
X = tf.constant(np.random.randn(3,1), name = "X")
```

You might find the following functions helpful:

- tf.matmul(..., ...) to do a matrix multiplication
- tf.add(..., ...) to do an addition
- np.random.randn(...) to initialize randomly

Computing the sigmoid

Great! You just implemented a linear function. Tensorflow offers a variety of commonly used neural network functions like tf.sigmoid and tf.softmax. For this exercise lets compute the sigmoid function of an input.
You will do this exercise using a placeholder variable x. When running the session, you should use the feed dictionary to pass in the input z. In this exercise, you will have to (i) create a placeholder x, (ii) define the operations needed to compute the sigmoid using tf.sigmoid, and then (iii) run the session.

Exercise: Implement the sigmoid function below. You should use the following:

- `tf.placeholder(tf.float32, name = "...")`
- `tf.sigmoid(...)`
- `sess.run(..., feed_dict = {x: z})`

Note that there are two typical ways to create and use sessions in tensorflow:

Method 1:

```
sess = tf.Session()
# Run the variables initialization (if needed), run the opera
tions
result = sess.run(..., feed_dict = {...})
sess.close() # Close the session
```

Method 2:

```
    with tf.Session() as sess:
        # run the variables initialization, run the operations
        result = sess.run(..., feed_dict = {...})
        # This takes care of closing the session for you
```

To summarize, you how know how to:

1. Create placeholders
2. Specify the computation graph corresponding to operations you want to compute
3. Create the session
4. Run the session, using a feed dictionary if necessary to specify placeholder variables' values.

Computing the sigmoid

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...m:

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

you can do it in one line of code in tensorflow!

Exercise: Implement the cross entropy loss. The function you will use is:

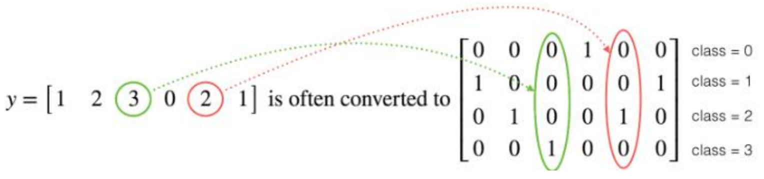
- `tf.nn.sigmoid_cross_entropy_with_logits(logits = ...,  labels = ...)`

Your code should input `z`, compute the sigmoid (to get `a`) and then compute the cross entropy cost J. All this can be done using one call to `tf.nn.sigmoid_cross_entropy_with_logits`, which computes

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

Using One Hot encodings

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:



$$y = \begin{bmatrix} 1 & 2 & 3 & 0 & 2 & 1 \end{bmatrix} \text{ is often converted to } \begin{array}{cl} \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} & \begin{matrix} \text{class} = 0 \\ \text{class} = 1 \\ \text{class} = 2 \\ \text{class} = 3 \end{matrix} \end{array}$$

This is called a "one hot" encoding, because in the converted representation exactly one element of each column is "hot" (meaning set to 1). To do this conversion in numpy, you might have to write a few lines of code. In tensorflow, you can use one line of code:

- tf.one_hot(labels, depth, axis)

Exercise: Implement the function below to take one vector of labels and the total number of classes C, and return the one hot encoding. Use tf.one_hot() to do this.

Initialize with zeros and ones

Now you will learn how to initialize a vector of zeros and ones. The function you will be calling is tf.ones(). To initialize with zeros you could use tf.zeros() instead. These functions take in a shape and return an array of dimension shape full of zeros and ones respectively.

Exercise: Implement the function below to take in a shape and to return an array (of the shape's dimension of ones).

- tf.ones(shape)

Building your first neural network in tensorflow

In this part of the assignment you will build a neural network using tensorflow. Remember that there are two parts to implement a tensorflow model:

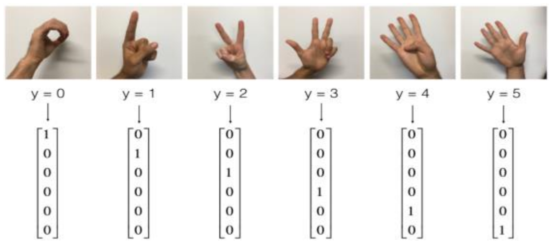
- Create the computation graph
- Run the graph

Problem statement: SIGNS Dataset

It's now your job to build an algorithm that would facilitate communications from a speech-impaired person to someone who doesn't understand sign language.

- **Training set**: 1080 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (180 pictures per number).
- **Test set**: 120 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (20 pictures per number).

Note that this is a subset of the SIGNS dataset. The complete dataset contains many more signs. Here are examples for each number, and how an explanation of how we represent the labels. These are the original pictures, before we lowered the image resolution to 64 by 64 pixels.



Note that 12288 comes from 64×64×3. 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.
Your 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). It is a great occasion to compare your numpy implementation to the tensorflow one.

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.

Create placeholders

Implement the function below to create the placeholders in tensorflow.

Initializing the parameters

Your second task is to initialize the parameters in tensorflow.

Exercise: Implement the function below to initialize the parameters in tensorflow. You are going use Xavier Initialization for weights and Zero Initialization for biases. The shapes are given below. As an example, to help you, for W1 and b1 you could use:

```
W1 = tf.get_variable("W1", [25,12288], initializer = tf.contrib.layers.xavier_initializer(seed = 1))
b1 = tf.get_variable("b1", [25,1], initializer = tf.zeros_initializer())
```

Please use seed = 1 to make sure your results match ours.

Forward propagation in tensorflow

You will now implement the forward propagation module in tensorflow. The function will take in a dictionary of parameters and it will complete the forward pass. The functions you will be using are:

- tf.add(...,...) to do an addition
- tf.matmul(...,...) to do a matrix multiplication
- tf.nn.relu(...) to apply the ReLU activation

Question: Implement the forward pass of the neural network. We commented for you the numpy equivalents so that you can compare the tensorflow implementation to numpy. 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. Therefore, you don't need a3!

You may have noticed that the forward propagation doesn't output any cache. You will understand why below, when we get to backpropagation.

Compute cost

As seen before, it is very easy to compute the cost using:

```
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = ..., labels = ...))
```

Question: Implement the cost function below.

- It is important to know that the "logits" and "labels" inputs of tf.nn.softmax_cross_entropy_with_logits are expected to be of shape (number of examples, num_classes). We have thus transposed Z3 and Y for you.
- Besides, tf.reduce_mean basically does the summation over the examples.

Backward propagation & parameter updates

This is where you become grateful to programming frameworks. All the backpropagation and the parameters update is taken care of 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.

For instance, for gradient descent the optimizer would be:

optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(cost)

To make the optimization you would do:

_ , c = sess.run([optimizer, cost], feed_dict={X: minibatch_X, Y: minibatch_Y})

This computes the backpropagation by passing through the tensorflow graph in the reverse order. From cost to inputs.

Note When coding, we often use _ as a "throwaway" variable to store values that we won't need to use later. Here, _ takes on the evaluated value of optimizer, which we don't need (and c takes the value of the cost variable).

Building the model

Now, you will bring it all together!

Insights:

- Your model seems big enough to fit the training set well. However, given the difference between train and test accuracy, you could try to add L2 or dropout regularization to reduce overfitting.
- Think about the session as a block of code to train the model. Each time you run the session on a minibatch, it trains the parameters. In total you have run the session a large number of times (1500 epochs) until you obtained well trained parameters.

What you should remember:

- Tensorflow is a programming framework used in deep learning
- The two main object classes in tensorflow are Tensors and Operators.
- When you code in tensorflow you have to take the following steps:
    - Create a graph containing Tensors (Variables, Placeholders ...) and Operations (tf.matmul, tf.add, ...)
    - Create a session
    - Initialize the session
    - Run the session to execute the graph
- You can execute the graph multiple times as you've seen in model()
- The backpropagation and optimization is automatically done when running the session on the "optimizer" object.