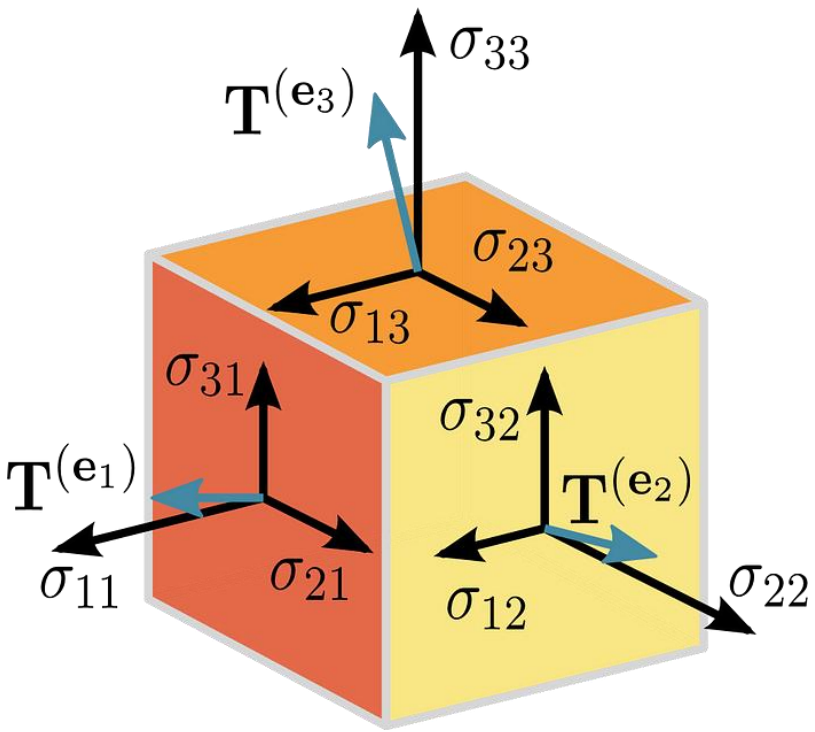
Practical 6

Aim : Introduction to TensorFlow.

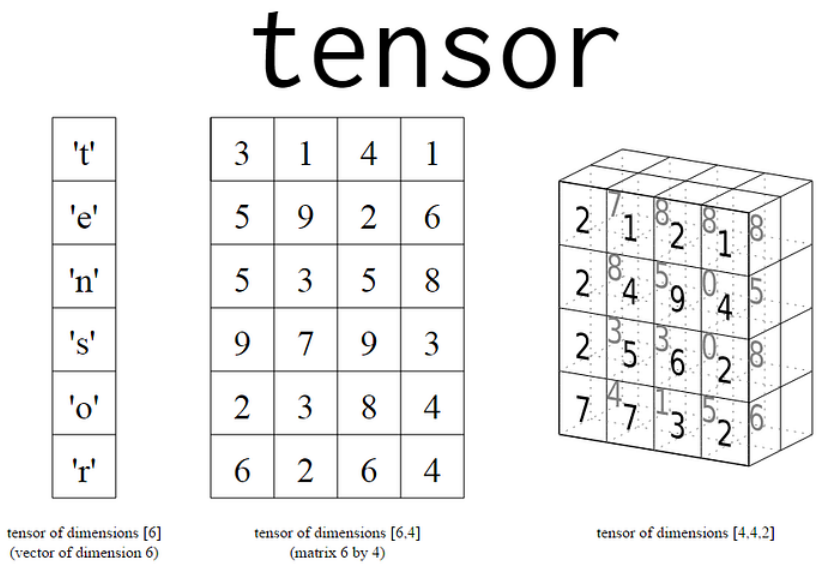
TensorFlow :

Tensorflow is one of the widely used libraries for implementing Machine learning and other algorithms involving large numbers of mathematical operations. Tensorflow was developed by Google and it's one of the most popular Machine Learning libraries on GitHub. Google uses Tensorflow for implementing Machine learning in almost all applications. For example, if you are using Google photos or Google voice search then you are using Tensorflow models indirectly, they work on large clusters of Google hardware and are powerful in perceptual tasks.The main aim of this article is to provide a beginner friendly introduction to TensorFlow, I assume that you already know a bit of python. The core component of TensorFlow is the computational graph and Tensors which traverse among all the nodes through edges. Let's have a brief introduction to each one of them.

Tensors:



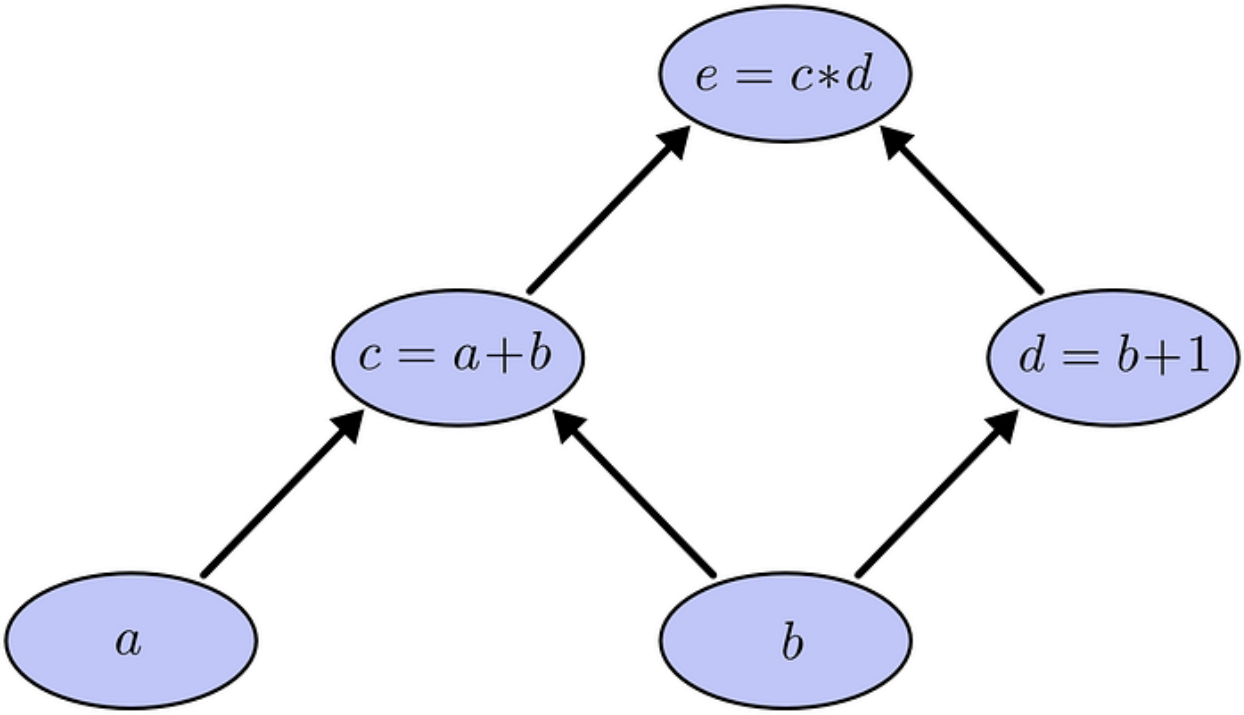
Mathematically a Tensor is a N-dimensional vector, meaning a Tensor can be used to represent N-dimensional datasets. The figure above is complex to understand. We'll look at it's simplified version



The above figure shows some simplified Tensors with minimum dimensions. As the dimensions keep on increasing, Data representation becomes more and more complex. For example if we take a Tensor of the form (3x3) then I can simply call it a matrix of 3 rows and columns. If I select another Tensor of the form (1000x3x3), I can call it a vector or set of 1000 3x3 matrices. Here we call (1000x3x3) the shape or Dimension of the resulting Tensor. Tensors can either be a constant or a variable.

Computational graphs (flow):

Now we understand what Tensor really means, and it's time to understand Flow. This flow refers to a computational graph or simply a graph, the graph can never be cyclic, each node in the graph represents an operation like addition, subtraction etc. And each operation results in the formation of a new Tensor.



The figure above shows a simple computational graph. The computational graph has the following properties:

The expression for above graph :

e = (a+b)x(b+1)

- Leaf vertices or start vertices are always Tensors. Means, An operation can never occur at the beginning of the graph and thus we can infer that each operation in the graph should accept a Tensor and produce a new Tensor. In the same way, A tensor cannot be present as a non-leaf node, meaning they should be always supplied as an input to the operation / node
  .
- A computational graph always represents a complex operation in a hierarchical order. The above expression can be organized in a hierarchical way, by representing a+b as c and b+1 as d. Therefore we can write e as:

e = (c)x(d) where c = a+b and d = b+1.

- Traversing the graph in reverse order results in the formation of sub expressions which are combined to form the final expression.
- When we traverse in forward direction, the vertex we encounter always becomes a dependency for the next vertex , for example c cannot be obtained without a and b, in the same way e cannot be obtained without solving for c and d.

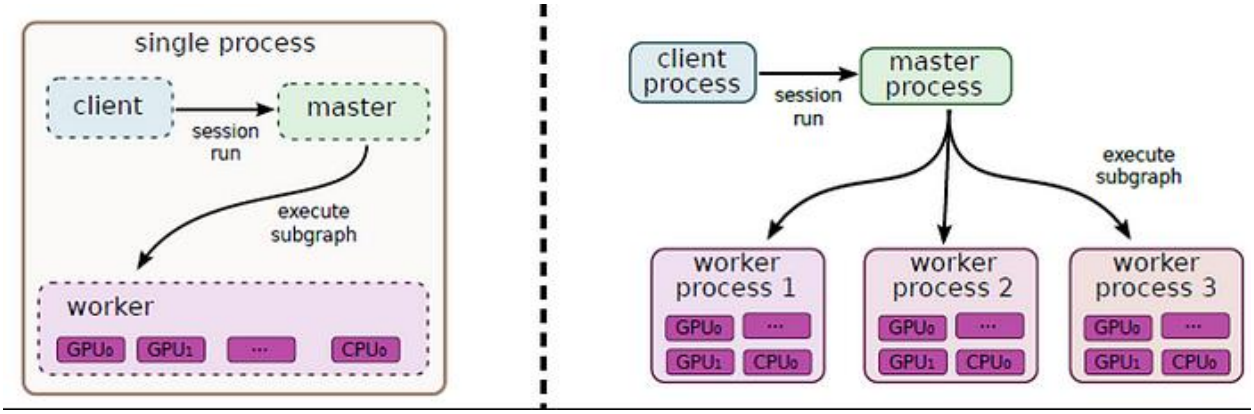
- **Operations in the nodes of the same level are independent of each other.** This is one of the important properties of Computational graph. When we construct a graph in the way shown in the figure, it is natural that nodes in the same level for example c and d are independent of each other, meaning there is no need to know c before evaluating d. Therefore they can be executed in parallel.

Parallelism in computational graphs:

Last property mentioned above is of course one of the most important properties, it clearly says that nodes at the same level are independent, meaning there is no need of sitting idle until c gets evaluated, you can parallel compute d while c is still being evaluated. Tensorflow greatly exploits this property.

Distributed Execution :

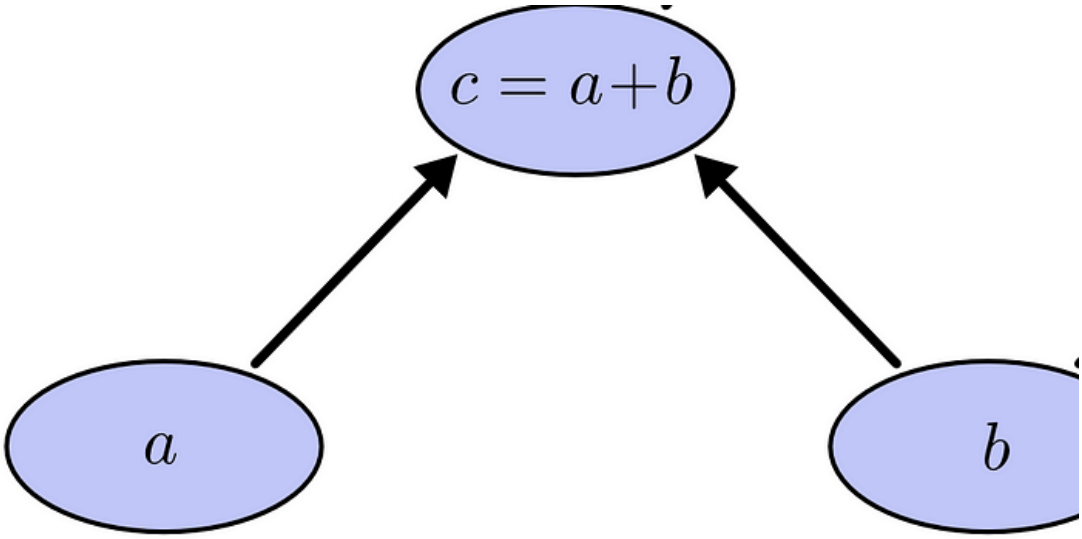
Tensorflow allows users to make use of parallel computing devices to perform operations faster. The nodes or operations of a computational system are automatically scheduled for parallel computing. This all happens internally, for example in the above graph, operation c can be scheduled on CPU and operation d can be scheduled on GPU. The figure below shows two perspectives of distributed execution :



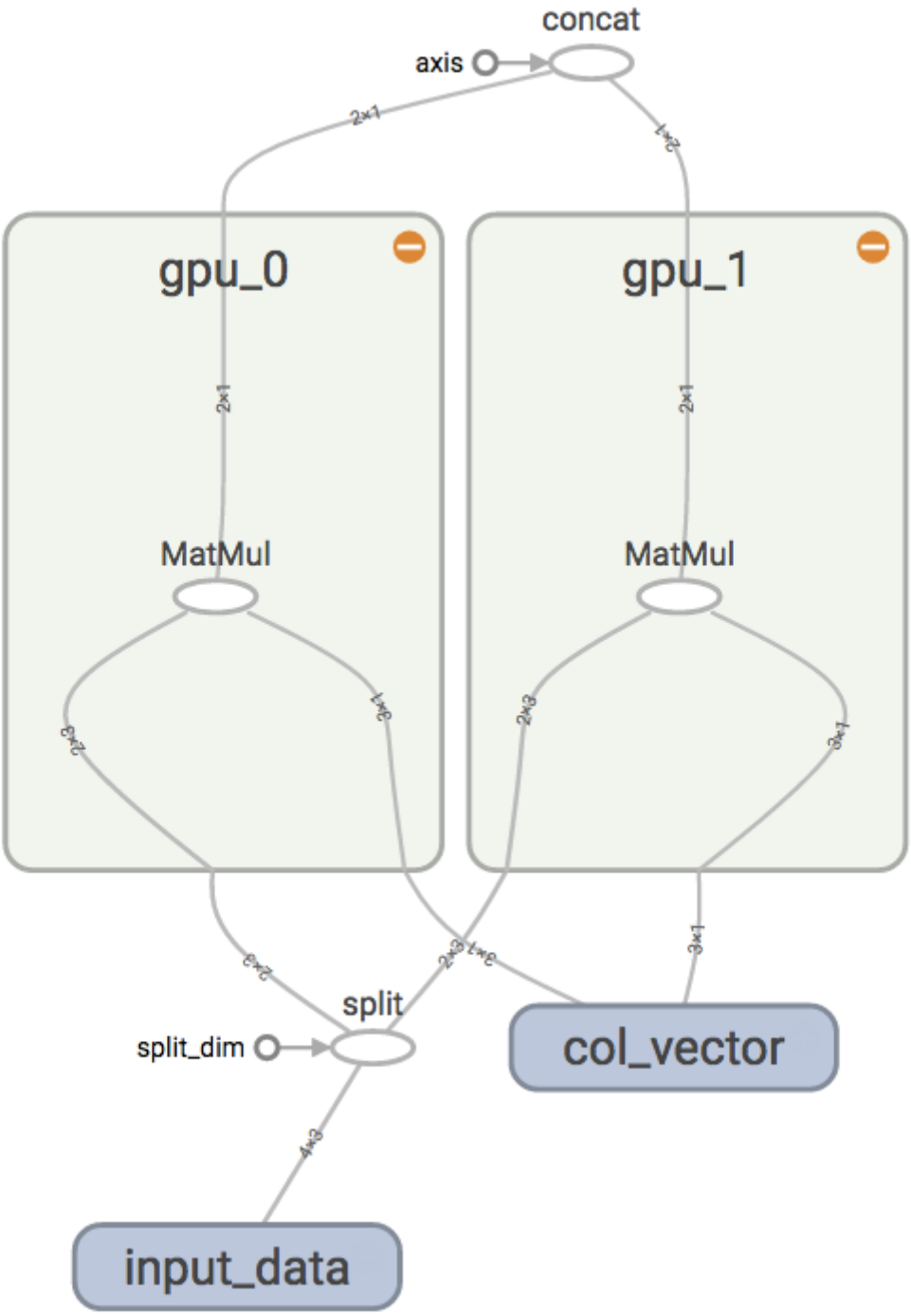
The first one, is a single system distributed execution where a single Tensorflow session(will be explained later) creates a single worker and the worker is responsible for scheduling tasks on various devices, in the second case, there are multiple workers , they can be on same machine or on different machines, each worker runs in its own context, in the above figure, worker process 1 runs on a separate Machine and schedules operations on all available devices.

Computational Subgraphs :

Subgraphs are the part of the main graph and are themselves computational graphs by nature. For example, in the above graph, we can obtain many Subgraphs, one of them is shown below



The graph above is a part of the main graph, from property 2 we can say that a Subgraph always represents a sub expression, as c is the subexpression of e. Subgraphs also satisfy the last property. Subgraphs in the same level are also independent of each other and can be executed in parallel. Therefore it's possible to schedule the entire Subgraph on a single device.



The above figure explains the parallel execution of Subgraphs. Here there are 2 Matrix multiplication operations, since both of them are at the same level, they are independent of each other, this holds good with the last property. The nodes are scheduled on different devices gpu_0 and gpu_1, this is because of the property of Independence.

Exchanging data between workers :

Now we know that Tensorflow distributes all its operations on different devices governed by workers. It is more common that, data in the form of Tensors are exchanged between workers, for example in the graph of e =(c)*(d) once c is calculated it is desirable to pass it on further to process e, therefore Tensor flows from node to node in upward direction. This movement is done as shown in the figure :

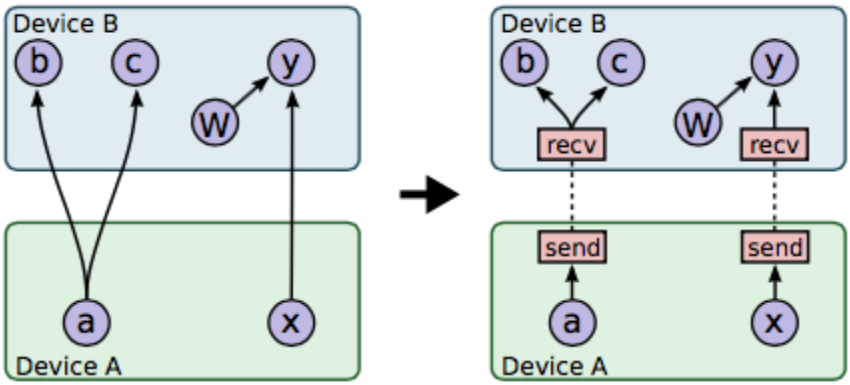


Figure 4: Before & after insertion of Send/Receive nodes

Here Tensors form device A has been passed on to device B. This induces some performance delays in a distributed system. The delays depend on an important property that is the size of a Tensor. Device B is in ideal mode until it receives input from Device A.

Need for compression :

Well, it's obvious that in computational graphs, Tensors flow between nodes. It's important to reduce the delays caused by the flow before it reaches the node where it can be processed. One such idea of reducing the size is by using Lossy compression.

The data type of tensors has a major role to play, let's understand why, it's obvious that we go for higher degrees of precision in Machine Learning operations, for example if we use float32 as the data type of a Tensor, them each value is represented using a 32 bit floating point number, so each value occupies a size of 32 bits, the same applies for 64 bit also. Assume a Tensor of shape (1000,440,440,3), the number of values that can be contained within the tensor will be 1000*440*440*3 If the data type is 32 bit then it's 32 times of this big number, it occupies a significant space in the memory and thus impose delays for the flow. Compression techniques can be used to reduce the size.

Lossy compression :

Lossy compression deals with compressing the size of data and does not care about its value, meaning its value may become corrupt or inaccurate during compression. But still if we have a 32 bit floating point number like 1.01010e-12, there is less importance that can be given to least significant digits. Changing or removing those values will not cause a much difference in our calculation. So Tensorflow automatically converts 32 bit floating point numbers to a 16 bit representation by ignoring all digits which are negligible, this reduces the size by almost half, if it's a 64 bit number, its compression to 16 bit will cause the reduction in size by almost 75%. Thus space occupied by Tensors can be minimized.

Once Tensor reaches the nodes, the 16 bit representation can be brought back to its original form just by appending 0s. Thus a 32 or 64 bit representation is bought back after it reaches the node for processing.

Practical - 7

Aim : Planar data classification with a hidden layer and Building of Deep Neural Network: step by step in any appropriate tool.

1 - Packages

Let's first import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- sklearn provides simple and efficient tools for data mining and data analysis.
- matplotlib is a library for plotting graphs in Python.
- testCases provides some test examples to assess the correctness of your functions
- planar_utils provide various useful functions used in this assignment

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
from testCases_v2 import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset,
load_extra_datasets
%matplotlib inline
np.random.seed(1) # set a seed so that the results are consistent
```

2 - Dataset

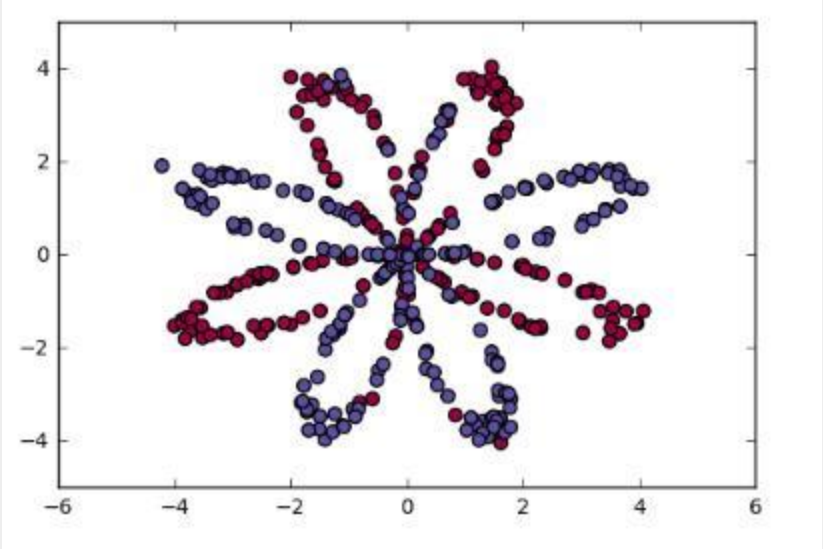
First, let's get the dataset you will work on. The following code will load a "flower" 2-class dataset into variables X and Y.

```
X, Y = load_planar_dataset()
X
```

```
X
Out[5]: array([[ 1.20444229e+00,  1.58709904e-01,  9.52471960e-02,
                 3.49178475e-01,  6.94150378e-01,  1.62065038e+00,
                 1.53856225e+00,  3.63085641e-02,  4.74591109e-01,
                 1.65695828e-01,  1.66446249e+00,  8.40285720e-01,
                 2.61695163e-01,  2.31614896e-01,  1.58013020e+00,
                 6.35509950e-03,  6.80610419e-01,  1.21400432e-01,
                 1.13281261e+00,  1.61505892e+00,  1.66454441e-01,
                 1.72438241e+00,  1.88667246e+00,  1.72327227e+00,
                 1.54661332e+00,  9.84590400e-01,  1.45313345e+00,
                 7.49043388e-01,  1.45048341e+00,  1.64287865e+00,
                 1.28141487e+00,  1.59574104e+00,  1.46298294e+00,
                 1.46629048e+00,  1.54348961e+00,  1.57013416e+00,
                 1.22995404e+00,  1.31142345e+00, -1.99364553e+00,
                 3.94564752e-01,  1.51715449e+00,  1.69169139e+00,
                 1.74186686e+00, -2.91373382e+00,  7.52150898e-01,
                 1.68537303e+00,  3.71160238e-01, -3.73033884e+00,
                 3.52484080e-01, -1.48694206e+00, -7.45290416e-01,
                 5.63807442e-01,  1.27093179e+00,  5.35133607e-01,
                -1.71330375e-01, -2.50197293e+00, -2.63275448e+00,
```

Visualize the dataset using matplotlib. The data looks like a "flower" with some red (label y=0) and some blue (y=1) points. Your goal is to build a model to fit this data.

```
# Visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



You have:

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

Let's first get a better sense of what our data is like.

Exercise: How many training examples do you have? In addition, what is the shape of the variables X and Y?

Hint: How do you get the shape of a numpy array? (help)

```
### START CODE HERE ### (≈ 3 lines of code)
shape_X = X.shape
shape_Y = Y.shape
m = shape_Y[1]  # training set size
### END CODE HERE ###
print ('The shape of X is: ' + str(shape_X))
print ('The shape of Y is: ' + str(shape_Y))
print ('I have m = %d training examples!' % (m))
```

```
The shape of X is: (2, 400)
The shape of Y is: (1, 400)
I have m = 400 training examples!
```

Expected Output:

shape of X	(2, 400)
shape of Y	(1, 400)
m	400

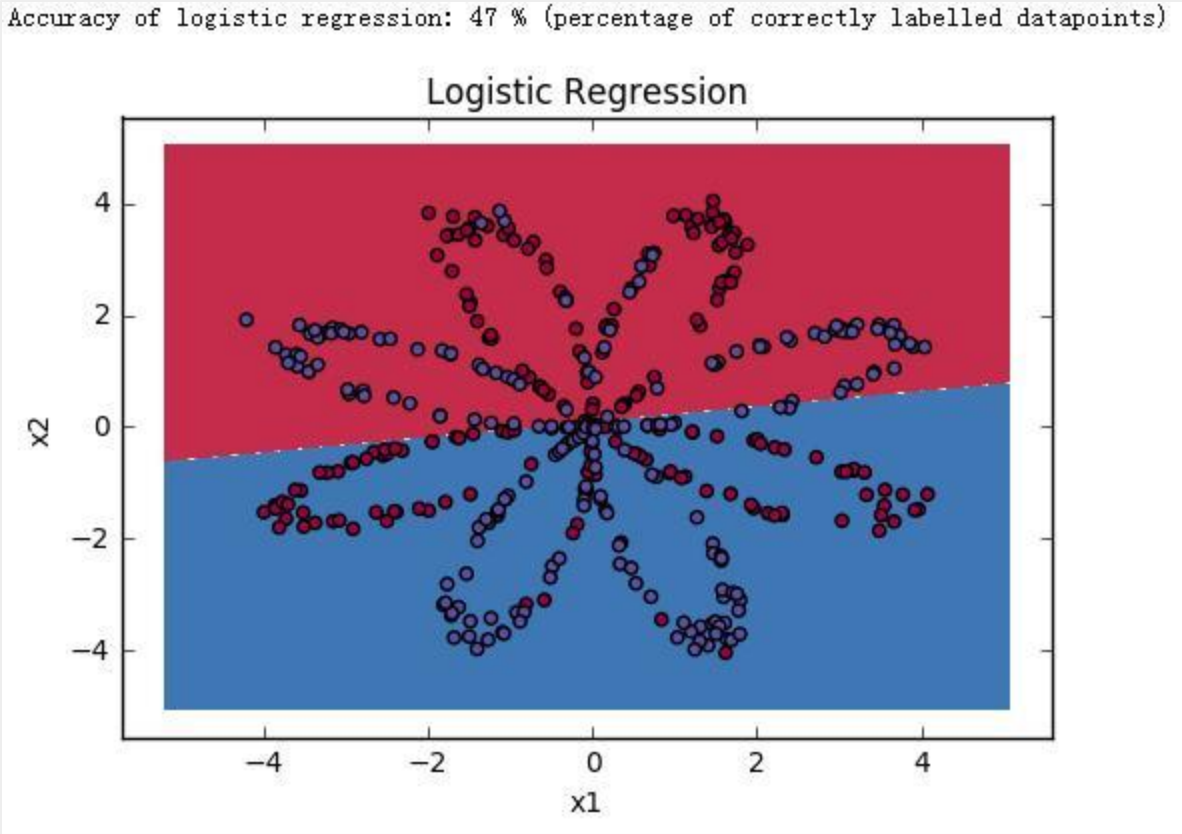
3 - Simple Logistic Regression

Before building a full neural network, lets first see how logistic regression performs on this problem. You can use sklearn's built-in functions to do that. Run the code below to train a logistic regression classifier on the dataset.

```
# Train the logistic regression classifier #Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

You can now plot the decision boundary of these models. Run the code below.

```
# Plot the decision boundary for logistic regression
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")
# Print accuracy
LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) + np.dot(1-Y,1-LR_predictions))/float(Y.size)*100) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

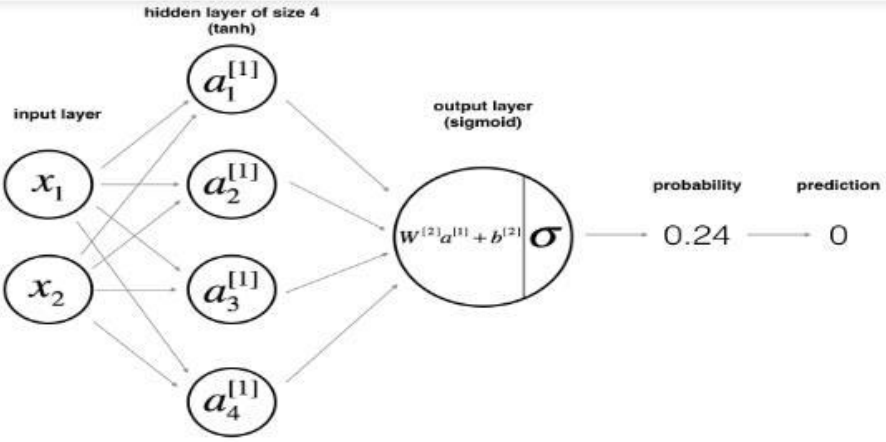


Interpretation: The dataset is not linearly separable, so logistic regression doesn't perform well. Hopefully a neural network will do better. Let's try this now!

4 - Neural Network model

Logistic regression did not work well on the "flower dataset". You are going to train a Neural Network with a single hidden layer.

Here is our model:



Mathematically:

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

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

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

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

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.

4.1 - Defining the neural network structure

Exercise: Define three variables:

- n_x: the size of the input layer
- n_h: the size of the hidden layer (set this to 4)
- n_y: the size of the output layer

```
# GRADED FUNCTION: layer_sizes
def layer_sizes(X, Y):
    """
    Arguments:
    X -- input dataset of shape (input size, number of examples)
    Y -- labels of shape (output size, number of examples)

    Returns:
    n_x -- the size of the input layer
    n_h -- the size of the hidden layer
    n_y -- the size of the output layer
    """
    ### START CODE HERE ### (≈ 3 lines of code)
    n_x = X.shape # size of input layer
    n_h = 4
    n_y = Y.shape # size of output layer
    ### END CODE HERE ###
    return (n_x, n_h, n_y)
X_assess, Y_assess = layer_sizes_test_case()
(n_x, n_h, n_y) = layer_sizes(X_assess, Y_assess)
print("The size of the input layer is: n_x = " + str(n_x))
print("The size of the hidden layer is: n_h = " + str(n_h))
print("The size of the output layer is: n_y = " + str(n_y))
```

```
The size of the input layer is: n_x = (5, 3)
The size of the hidden layer is: n_h = 4
The size of the output layer is: n_y = (2, 3)
```

Expected Output (these are not the sizes you will use for your network, they are just used to assess the function you've just coded).

n_x	5
n_h	4
n_y	2

4.2 - Initialize the model's parameters

Exercise: Implement the function initialize_parameters().

Instructions:

- Make sure your parameters' sizes are right. Refer to the neural network figure above if needed.
- You will initialize the weights matrices with random values.
    - Use: np.random.randn(a,b) * 0.01 to randomly initialize a matrix of shape (a,b).
- You will initialize the bias vectors as zeros.
    - Use: np.zeros((a,b)) to initialize a matrix of shape (a,b) with zeros.

```
# GRADED FUNCTION: initialize_parameters
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    params -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(2) # we set up a seed so that your output matches ours although the initialization is random.

    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros(shape=(n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros(shape=(n_y, 1))
    ### END CODE HERE ###

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
```

```
    "b1": b1,
    "W2": W2,
    "b2": b2}

  return parameters
n_x, n_h, n_y = initialize_parameters_test_case()
parameters = initialize_parameters(n_x, n_h, n_y)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-0.00416758 -0.00056267]
 [-0.02136196  0.01640271]
 [-0.01793436 -0.00841747]
 [ 0.00502881 -0.01245288]]
b1 = [[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
W2 = [[-0.01057952 -0.00909008  0.00551454  0.02292208]]
b2 = [[ 0.]]
```

Expected Output:

W1	[[-0.00416758 -0.00056267] [-0.02136196 0.01640271] [-0.01793436 -0.00841747] [0.00502881 -0.01245288]]
b1	[[0.] [0.] [0.] [0.]]
W2	[[-0.01057952 -0.00909008 0.00551454 0.02292208]]
b2	[[0.]]

4.3 - The Loop

Question: Implement `forward_propagation()`.

Instructions:

- Look above at the mathematical representation of your classifier.
- You can use the function `sigmoid()`. It is built-in (imported) in the notebook.
- You can use the function `np.tanh()`. It is part of the numpy library.
- The steps you have to implement are:
  1. Retrieve each parameter from the dictionary "parameters" (which is the output of `initialize_parameters()`) by using `parameters[".."]`.
  2. Implement Forward Propagation. Compute $Z^{[1]}, A^{[1]}, Z^{[2]}$ and $A^{[2]}$ (the vector of all your predictions on all the examples in the training set).
- Values needed in the backpropagation are stored in "`cache`". The `cache` will be given as an input to the backpropagation function.

```
# GRADED FUNCTION: forward_propagation
def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing your parameters (output of initialization function)

    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
        print("X:"+str(X.shape)) # 3 samples, 2 inputs
        print("W1:"+str(W1.shape)) # 4 neural nodes, 2 inputs
    print("b1:"+str(b1.shape))
        print("W2:"+str(W2.shape)) # 1 neural node, 4 inputs
    print("b2:"+str(b2.shape))
    ### END CODE HERE ###

    # Implement Forward Propagation to calculate A2 (probabilities)
    ### START CODE HERE ### (≈ 4 lines of code)
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    print("A2:"+str(A2.shape))  # A2:(1, 3)
    ### END CODE HERE ###

    assert(A2.shape == (1, X.shape[1]))

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

```

```
    return A2, cache
X_assess, parameters = forward_propagation_test_case()
A2, cache = forward_propagation(X_assess, parameters)
# Note: we use the mean here just to make sure that your output matches ours.
print(np.mean(cache['Z1'])
,np.mean(cache['A1']),np.mean(cache['Z2']),np.mean(cache['A2']))
```

```
X: (2, 3)
W1: (4, 2)
b1: (4, 1)
W2: (1, 4)
b2: (1, 1)
0.262818640198 0.091999045227 -1.30766601287 0.212877681719
```

Expected Output:

0.262818640198 0.091999045227 -1.30766601287 0.212877681719

Now that you have computed $A^{[2]}$ (in the Python variable "A2"), which contains $a^{[2](i)}$ for every example, you can compute the cost function as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right) \tag{13}$$

Exercise: Implement `compute_cost()` to compute the value of the cost J.

Instructions:

- There are many ways to implement the cross-entropy loss. To help you, we give you how we would have implemented $-\sum_{i=0}^{m} y^{(i)}\log(a^{[2](i)})$:

```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)                # no need to use a for loop!
```

(you can use either `np.multiply()` and then `np.sum()` or directly `np.dot()`).

```
# GRADED FUNCTION: compute_cost
def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)

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

    Returns:
    cost -- cross-entropy cost given equation (13)
    """

    m = Y.shape[1] # number of example
    # Compute the cross-entropy cost
    ### START CODE HERE ### (≈ 2 lines of code)
    logprobs = np.multiply(np.log(A2), Y) +  np.multiply(np.log(1 - A2), (1 - Y))
```

```
    cost = - np.sum(logprobs) / m
    ### END CODE HERE ###

    cost = np.squeeze(cost)     # makes sure cost is the dimension we expect.
                                # E.g., turns [[17]] into 17
    assert(isinstance(cost, float))

    return cost
A2, Y_assess, parameters = compute_cost_test_case()
print("cost = " + str(compute_cost(A2, Y_assess, parameters)))
```

```
cost = 0.693058761039
```

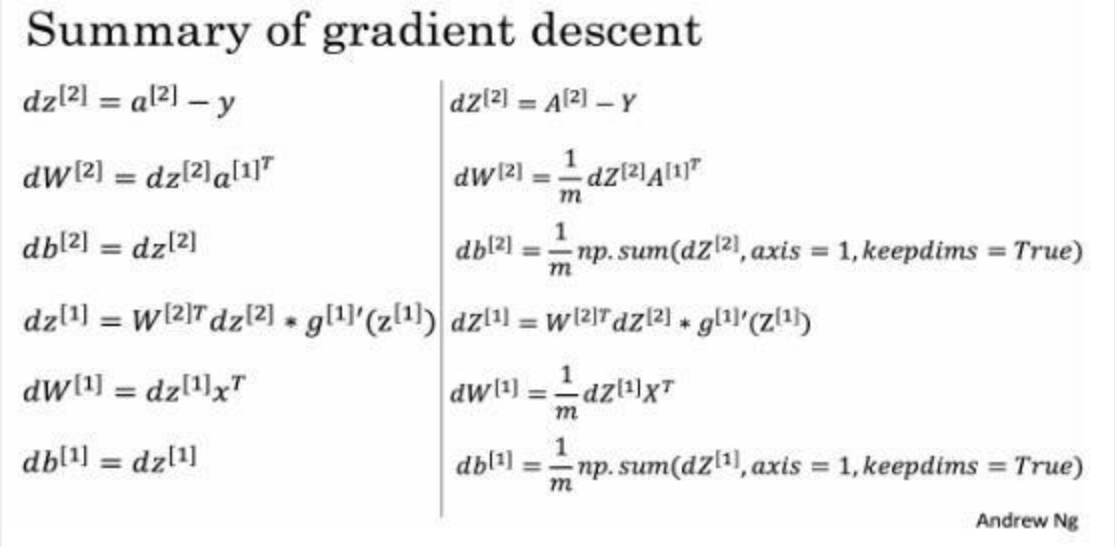
Expected Output:

cost	0.693058761...

Using the cache computed during forward propagation, you can now implement backward propagation.

Question: Implement the function backward_propagation().

Instructions: Backpropagation is usually the hardest (most mathematical) part in deep learning. To help you, here again is the slide from the lecture on backpropagation. You'll want to use the six equations on the right of this slide, since you are building a vectorized implementation.




```
# GRADED FUNCTION: backward_propagation
def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (2, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

    Returns:
    grads -- python dictionary containing your gradients with respect to different parameters
    """
    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary "parameters".
    ### START CODE HERE ### (≈ 2 lines of code)
    W1 = parameters['W1']
    W2 = parameters['W2']
    ### END CODE HERE ###

    # Retrieve also A1 and A2 from dictionary "cache".
    ### START CODE HERE ### (≈ 2 lines of code)
    A1 = cache['A1']
    A2 = cache['A2']
    ### END CODE HERE ###
```

```
    # Backward propagation: calculate dW1, db1, dW2, db2.
    ### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on slide above)
    # Z1 = np.dot(W1, X) + b1 # Z2 partial derivative of W1, dZ2/dW1 = (dZ2/dW2)*(dW2/dZ1)*(dZ1/dW1)= (dZ2/dW2)*( dW2/dZ1) * XT
        # A1 = np.tanh(Z1) # Z2 is partial derivative of Z1, dZ2/dZ1 = (dZ2/dW2)*(dW2/dZ1) = (dZ2/dW2) * dTanh = (dZ2/dW2) * (1- np.power(A1, 2))
        # Z2 = np.dot(W2, A1) + b2 # Z2 partial derivative of W2, dZ2/dW2 = dZ2 * A1.T
    # A2 = sigmoid(Z2)

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

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

    return grads
Parameters paramete , cache, X_assess, Y_assess = backward_propagation_test_case()
grads = backward_propagation(parameters, cache, X_assess, Y_assess)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dW2 = "+ str(grads["dW2"]))
print ("db2 = "+ str(grads["db2"]))
```

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: $\theta=\theta-\alpha\partial J\partial\theta\theta=\theta-\alpha\partial J\partial\theta$ where $\alpha\alpha$ is the learning rate and $\theta\theta$ 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.

```
# GRADED FUNCTION: update_parameters
def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Updates parameters using the gradient descent update rule given above

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients

    Returns:
    parameters -- python dictionary containing your updated parameters
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    ### END CODE HERE ###

    # Retrieve each gradient from the dictionary "grads"
    ### START CODE HERE ### (≈ 4 lines of code)
    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']
    ## END CODE HERE ###

    # Update rule for each parameter
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    ### END CODE HERE ###

    parameters = {"W1": W1,
                  "b1": b1,
```

```
    "W2": W2,
    "b2": b2}

  return parameters
parameters, grads = update_parameters_test_case()
parameters = update_parameters(parameters, grads)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-0.00643025  0.01936718]
 [-0.02410458  0.03978052]
 [-0.01653973 -0.02096177]
 [ 0.01046864 -0.05990141]]
b1 = [[ -1.02420756e-06]
 [  1.27373948e-05]
 [  8.32996807e-07]
 [ -3.20136836e-06]]
W2 = [[-0.01041081 -0.04463285  0.01758031  0.04747113]]
b2 = [[ 0.00010457]]
```

Expected Output:

W1	[[-0.00643025 0.01936718] [-0.02410458 0.03978052] [-0.01653973 -0.02096177] [0.01046864 -0.05990141]]
b1	[[-1.02420756e-06] [1.27373948e-05] [8.32996807e-07] [-3.20136836e-06]]
W2	[[-0.01041081 -0.04463285 0.01758031 0.04747113]]
b2	[[0.00010457]]

4.4 - Integrate parts 4.1, 4.2 and 4.3 in nn_model()

Question: Build your neural network model in nn_model().

Instructions: The neural network model has to use the previous functions in the right order.

```
# GRADED FUNCTION: nn_model
def nn_model(X, Y, n_h, num_iterations=10000, print_cost=False):
  """
  Arguments:
  X -- dataset of shape (2, number of examples)
  Y -- labels of shape (1, number of examples)
  n_h -- size of the hidden layer
  num_iterations -- Number of iterations in gradient descent loop
  print_cost -- if True, print the cost every 1000 iterations

  Returns:
  parameters -- parameters learnt by the model. They can then be used to predict.
```

```
    """

    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]

    # Initialize parameters, then retrieve W1, b1, W2, b2. Inputs: "n_x, n_h, n_y". Outputs = "W1, b1, W2, b2, parameters".
    ### START CODE HERE ### (≈ 5 lines of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    ### END CODE HERE ###

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        ### START CODE HERE ### (≈ 4 lines of code)
        # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
        A2, cache = forward_propagation(X, parameters)

        # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
        cost = compute_cost(A2, Y, parameters)
        # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
        grads = backward_propagation(parameters, cache, X, Y)
        # Gradient descent parameter update. Inputs: "parameters, grads". Outputs: "parameters".
        parameters = update_parameters(parameters, grads)

        ### END CODE HERE ###

        # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" % (i, cost))
    return parameters
X_assess, Y_assess = nn_model_test_case()
parameters = nn_model(X_assess, Y_assess, 4, num_iterations=10000, print_cost=True)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
```

```
    print("b2 = " + str(parameters["b2"]))
```

```
Cost after iteration 0: 0.692739
Cost after iteration 1000: 0.693147
Cost after iteration 2000: 0.693147
Cost after iteration 3000: 0.693147
Cost after iteration 4000: 0.693147
Cost after iteration 5000: 0.693147
Cost after iteration 6000: 0.693147
Cost after iteration 7000: 0.693147
Cost after iteration 8000: 0.693147
Cost after iteration 9000: 0.693147
W1 = [[-0.0071777   0.00690939]
 [-0.02394205  0.0228162 ]
 [-0.01636577 -0.01231018]
 [ 0.01155069 -0.02864301]]
b1 = [[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
W2 = [[ 0.  0.  0.  0.]]
b2 = [[ 0.]]
```

Expected Output:

cost after iteration 0	0.692739
⋮	⋮
W1	[[-0.65848169 1.21866811] [-0.76204273 1.39377573] [0.5792005 -1.10397703] [0.76773391 -1.41477129]]
b1	[[0.287592] [0.3511264] [-0.2431246] [-0.35772805]]
W2	[[-2.45566237 -3.27042274 2.00784958 3.36773273]]
b2	[[0.20459656]]

4.5 Predictions

Question: Use your model to predict by building predict(). Use forward propagation to predict results.

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

As an example, if you would like to set the entries of a matrix X to 0 and 1 based on a threshold you would do: `X_new = (X > threshold)`

```
    # GRADED FUNCTION: predict
    def predict(parameters, X):
        """
        Using the learned parameters, predicts a class for each example in X

        Arguments:
        parameters -- python dictionary containing your parameters
        X -- input data of size (n_x, m)

        Returns
        predictions -- vector of predictions of our model (red: 0 / blue: 1)
        """

        # Computes probabilities using forward propagation, and classifies to 0/1 using 0.5 as the threshold.
```

```
    ### START CODE HERE ### (≈ 2 lines of code)
    A2, cache = forward_propagation(X, parameters)
        # np.round round
    predictions = np.round(A2)
    ### END CODE HERE ###

    return predictions
parameters, X_assess = predict_test_case()
predictions = predict(parameters, X_assess)
print("predictions mean = " + str(np.mean(predictions)))
```

```
predictions mean = 0.666666666667
```

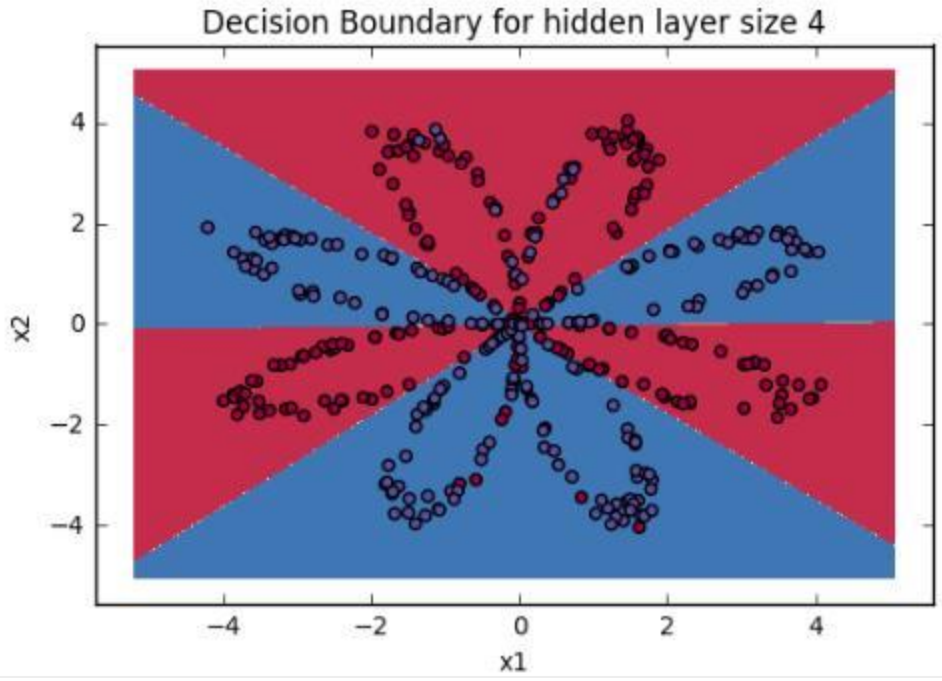
Expected Output:

predictions mean	0.666666666667

It is time to run the model and see how it performs on a planar dataset. Run the following code to test your model with a single hidden layer of n_h hidden units.

```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 4, num_iterations=10000, print_cost=True)
# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

```
Cost after iteration 0: 0.693048
Cost after iteration 1000: 0.288083
Cost after iteration 2000: 0.254385
Cost after iteration 3000: 0.233864
Cost after iteration 4000: 0.226792
Cost after iteration 5000: 0.222644
Cost after iteration 6000: 0.219731
Cost after iteration 7000: 0.217504
Cost after iteration 8000: 0.219454
Cost after iteration 9000: 0.218607
<matplotlib.text.Text at 0x7fe102f16668>
```




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

```
Accuracy: 90%
```

Expected Output:

Accuracy	90%

Accuracy is really high compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision boundaries, unlike logistic regression.

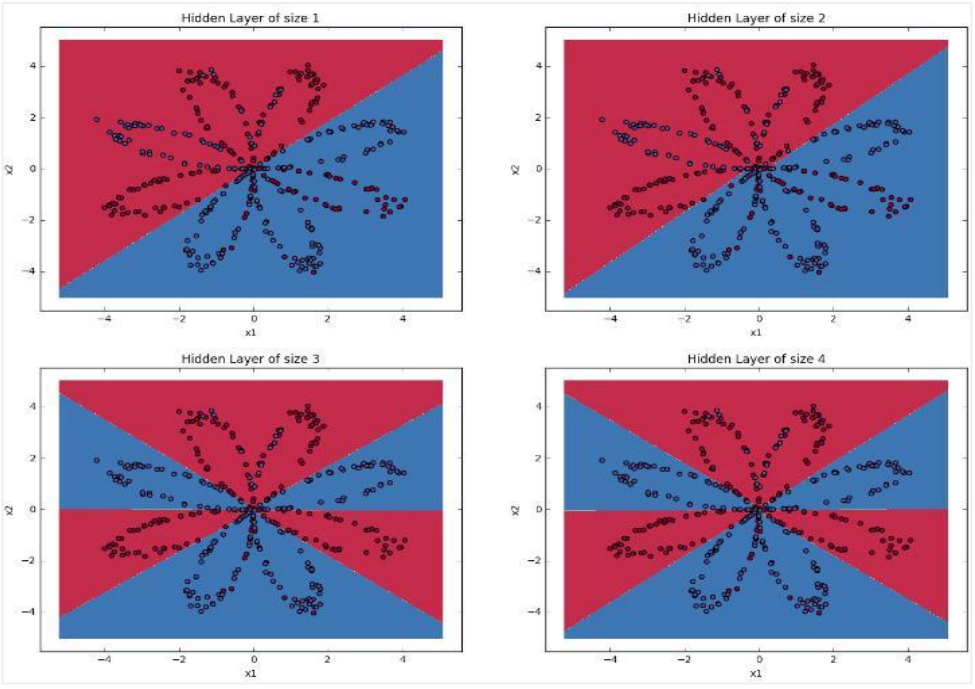
Now, let's try out several hidden layer sizes.

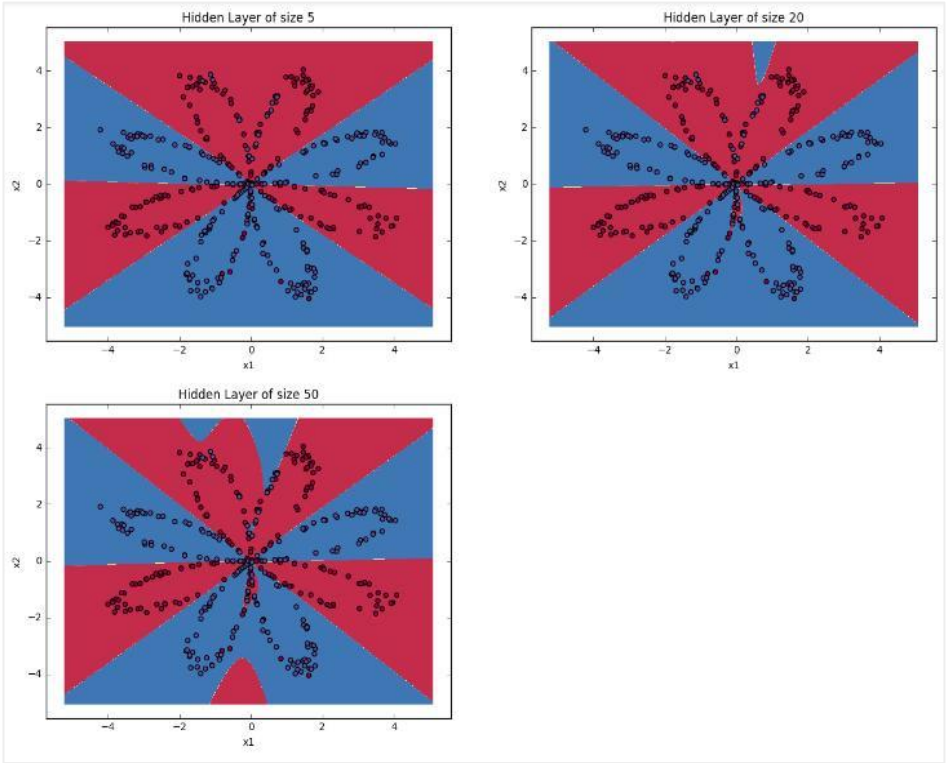
4.6 - Tuning hidden layer size (optional/ungraded exercise)

Run the following code. It may take 1-2 minutes. You will observe different behaviors of the model for various hidden layer sizes.

```
# This may take about 2 minutes to run
plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 5000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-
predictions.T))/float(Y.size)*100)
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

Accuracy for 1 hidden units: 67.5 %
Accuracy for 2 hidden units: 67.25 %
Accuracy for 3 hidden units: 90.75 %
Accuracy for 4 hidden units: 90.5 %
Accuracy for 5 hidden units: 91.25 %
Accuracy for 20 hidden units: 90.0 %
Accuracy for 50 hidden units: 90.25 %


Hidden Layer of size 1
Hidden Layer of size 2
Hidden Layer of size 3
Hidden Layer of size 4
x1
x2


Hidden Layer of size 5
Hidden Layer of size 20
Hidden Layer of size 50
x1
x2

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 noticeable overfitting.
- You will also learn later about regularization, which lets you use very large models (such as n_h = 50) without much overfitting.

Optional questions:

Note: Remember to submit the assignment but clicking the blue "Submit Assignment" button at the upper-right.

Some optional/ungraded questions that you can explore if you wish:

- What happens when you change the tanh activation for a sigmoid activation or a ReLU activation?
- Play with the learning_rate. What happens?
- What if we change the dataset? (See part 5 below!)

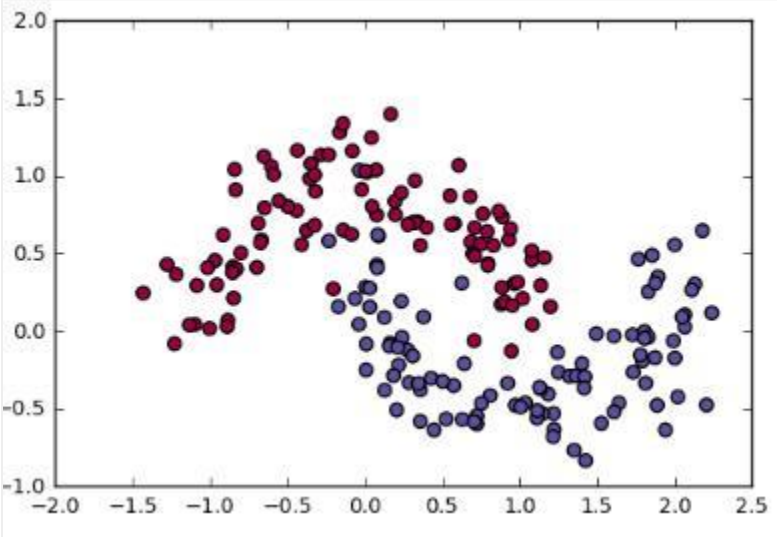
You've learnt to:

Build a complete neural network with a hidden layerMake a good use of a nonlinear unitImplemented forward propagation and backpropagation, and trained a neural networkSee the impact of varying the hidden layer size, including overfitting.

5) Performance on other datasets

```
# Datasets
noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure = load_extra_datasets()
datasets = {"noisy_circles": noisy_circles,
            "noisy_moons": noisy_moons,
            "blobs": blobs,
            "gaussian_quantiles": gaussian_quantiles}
### START CODE HERE ### (choose your dataset)
dataset = "noisy_moons"
### END CODE HERE ###
X, Y = datasets[dataset]
```

```
    X, Y = X.T, Y.reshape(1, Y.shape[0])
    # make blobs binary
    if dataset == "blobs":
        Y = Y%2
    # Visualize the data
    plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



Practical - 8

Aim : Demonstrate the implementation of CNN

Padding :

We have seen that convolving an input of 6 X 6 dimension with a 3 X 3 filter results in 4 X 4 output. We can generalize it and say that if the input is n X n and the filter size is f X f, then the output size will be (n-f+1) X (n-f+1):

- Input: n X n
- Filter size: f X f
- Output: (n-f+1) X (n-f+1)

There are primarily two disadvantages here:

- Every time we apply a convolutional operation, the size of the image shrinks
- Pixels present in the corner of the image are used only a few times during convolution as compared to the central pixels. Hence, we do not focus too much on the corners since that can lead to information loss

To overcome these issues, we can pad the image with an additional border, i.e., we add one pixel all around the edges. This means that the input will be an 8 X 8 matrix (instead of a 6 X 6 matrix). Applying convolution of 3 X 3 on it will result in a 6 X 6 matrix which is the original shape of the image. This is where padding comes to the fore:

- Input: n X n
- Padding: p
- Filter size: f X f
- Output: (n+2p-f+1) X (n+2p-f+1)

There are two common choices for padding:

- Valid: It means no padding. If we are using valid padding, the output will be (n-f+1) X (n-f+1)
- Same: Here, we apply padding so that the output size is the same as the input size, i.e., n+2p-f+1 = n
  So, p = (f-1)/2

We now know how to use padded convolution. This way we don't lose a lot of information and the image does not shrink either. Next, we will look at how to implement strided convolutions.

Strided Convolutions :

Suppose we choose a stride of 2. So, while convoluting through the image, we will take two steps – both in the horizontal and vertical directions separately. The dimensions for stride *s* will be:

- Input: n X n
- Padding: p
- Stride: s
- Filter size: f X f
- Output: [(n+2p-f)/s+1] X [(n+2p-f)/s+1]

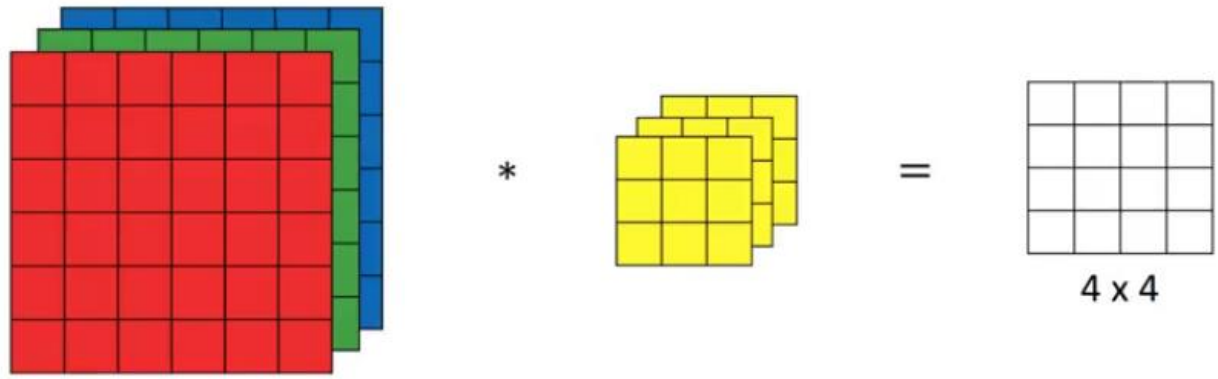
Stride helps to reduce the size of the image, a particularly useful feature.

Convolutions Over Volume :

Suppose, instead of a 2-D image, we have a 3-D input image of shape 6 X 6 X 3. How will we apply convolution on this image? We will use a 3 X 3 X 3 filter instead of a 3 X 3 filter. Let's look at an example:

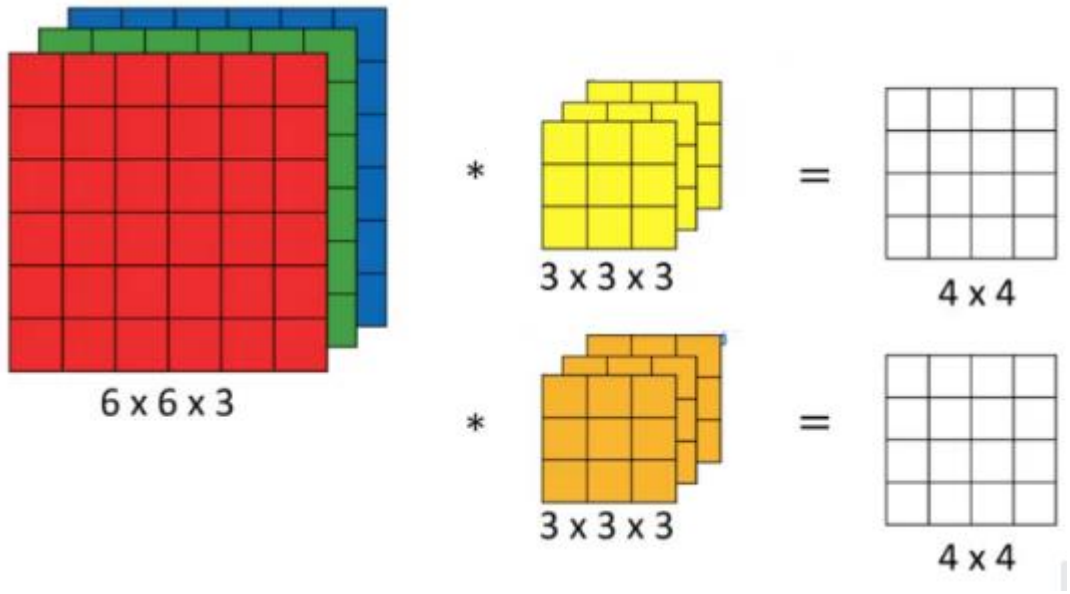
- Input: 6 X 6 X 3
- Filter: 3 X 3 X 3

The dimensions above represent the height, width and channels in the input and filter. *Keep in mind that the number of channels in the input and filter should be the same.* This will result in an output of 4 X 4. Let's understand it visually:



Since there are three channels in the input, the filter will consequently also have three channels. After convolution, the output shape is a 4 X 4 matrix. So, the first element of the output is the sum of the element-wise product of the first 27 values from the input (9 values from each channel) and the 27 values from the filter. After that we convolve over the entire image.

Instead of using just a single filter, we can use multiple filters as well. How do we do that? Let's say the first filter will detect vertical edges and the second filter will detect horizontal edges from the image. If we use multiple filters, the output dimension will change. So, instead of having a 4 X 4 output as in the above example, we would have a 4 X 4 X 2 output (if we have used 2 filters):



Generalized dimensions can be given as:

- Input: n X n X nc
- Filter: f X f X nc
- Padding: p
- Stride: s
- Output: [(n+2p-f)/s+1] X [(n+2p-f)/s+1] X nc'

Here, nc is the number of channels in the input and filter, while nc' is the number of filters

One Layer of a Convolutional Network :

Once we get an output after convolving over the entire image using a filter, we add a bias term to those outputs and finally apply an activation function to generate activations. *This is one layer of a convolutional network*. Recall that the equation for one forward pass is given by:

$$z[1] = w[1]*a[0] + b[1]$$

$$a[1] = g(z[1])$$

In our case, input (6 X 6 X 3) is a[0]and filters (3 X 3 X 3) are the weights w[1]. These activations from layer 1 act as the input for layer 2, and so on. Clearly, the number of parameters in case of convolutional neural networks is independent of the size of the image. It essentially depends on the filter size. Suppose we have 10 filters, each of shape 3 X 3 X 3. What will be the number of parameters in that layer? Let's try to solve this:

- Number of parameters for each filter = 3*3*3 = 27
- There will be a bias term for each filter, so total parameters per filter = 28
- As there are 10 filters, the total parameters for that layer = 28*10 = 280

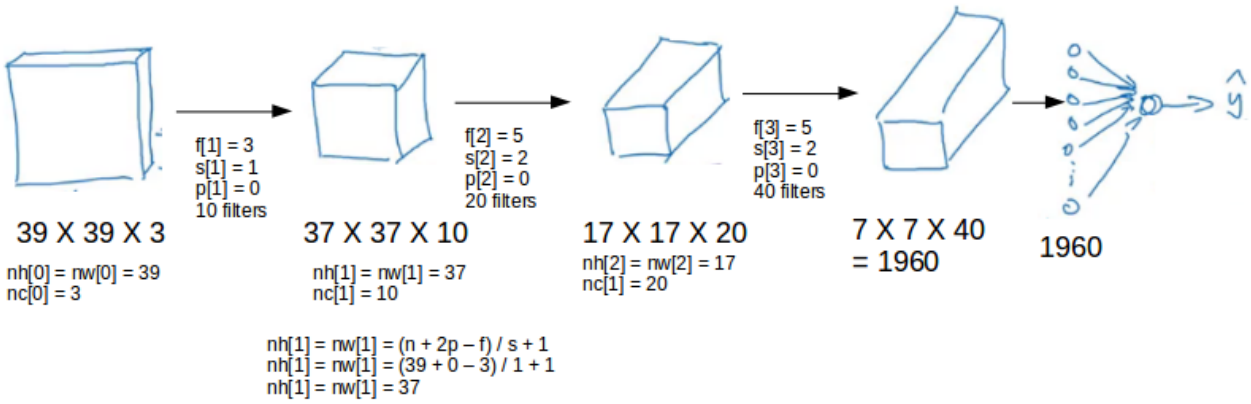
No matter how big the image is, the parameters only depend on the filter size. Awesome, isn't it? Let's have a look at the summary of notations for a convolution layer:

- f[l] = filter size
- p[l] = padding
- s[l] = stride
- n[c][l] = number of filters

Let's combine all the concepts we have learned so far and look at a convolutional network example.

Simple Convolutional Network Example :

This is how a typical convolutional network looks like:



We take an input image (size = 39 X 39 X 3 in our case), convolve it with 10 filters of size 3 X 3, and take the stride as 1 and no padding. This will give us an output of 37 X 37 X 10. We convolve this output further and get an output of 7 X 7 X 40 as shown above. Finally, we take all these numbers (7 X 7 X 40 = 1960), unroll them into a large vector, and pass them to a classifier that will make predictions. This is a microcosm of how a convolutional network works.

There are a number of hyperparameters that we can tweak while building a convolutional network. These include the number of filters, size of filters, stride to be used, padding, etc. We will look at each of these in detail later in this article. Just keep in mind that as we go deeper into the network, the size of the image shrinks whereas the number of channels usually increases.

In a convolutional network (ConvNet), there are basically three types of layers:

- Convolution layer

- Pooling layer
- Fully connected layer

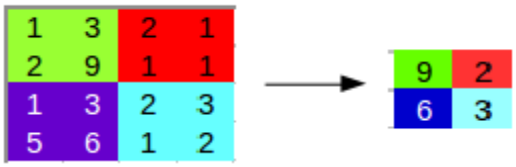
Let’s understand the pooling layer in the next section.

Pooling Layers :

Pooling layers are generally used to reduce the size of the inputs and hence speed up the computation. Consider a 4 X 4 matrix as shown below:

1	3	2	1
2	9	1	1
1	3	2	3
5	6	1	2

Applying max pooling on this matrix will result in a 2 X 2 output:



For every consecutive 2 X 2 block, we take the max number. Here, we have applied a filter of size 2 and a stride of 2. These are the hyperparameters for the pooling layer. Apart from max pooling, we can also apply average pooling where, instead of taking the max of the numbers, we take their average. In summary, the hyperparameters for a pooling layer are:

- Filter size
- Stride
- Max or average pooling

If the input of the pooling layer is nh X nw X nc, then the output will be [{(nh – f) / s + 1} X {(nw – f) / s + 1} X nc].

Program :

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential()
```

```
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

Here, x_train and y_train are the training data and labels, respectively, and x_test and y_test are the validation data and labels, respectively. We are training the model for 10 epochs and using the validation data to evaluate the performance of the model after each epoch.

Practical - 9

Aim : Building of a Recurrent Neural Network - Step by Step

Step 1: Create the Architecture for our RNN model

Our next task is defining all the necessary variables and functions we'll use in the RNN model. Our model will take in the input sequence, process it through a hidden layer of 100 units, and produce a single valued output:

```
learning_rate = 0.0001
nepoch = 25
T = 50                # length of sequence
hidden_dim = 100
output_dim = 1
bptt_truncate = 5
min_clip_value = -10
max_clip_value = 10
```

We will then define the weights of the network:

```
U = np.random.uniform(0, 1, (hidden_dim, T))
W = np.random.uniform(0, 1, (hidden_dim, hidden_dim))
V = np.random.uniform(0, 1, (output_dim, hidden_dim))
```

Here,

- U is the weight matrix for weights between input and hidden layers
- V is the weight matrix for weights between hidden and output layers
- W is the weight matrix for shared weights in the RNN layer (hidden layer)

Finally, we will define the activation function, sigmoid, to be used in the hidden layer:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Step 2: Train the Model

Now that we have defined our model, we can finally move on with training it on our sequence data. We can subdivide the training process into smaller steps, namely:

We need to repeat these steps until convergence. If the model starts to overfit, stop! Or simply pre-define the number of epochs.

Step 2.1: Check the loss on training data

We will do a forward pass through our RNN model and calculate the squared error for the predictions for all records in order to get the loss value.

```
for epoch in range(nepoch):
    # check loss on train
    loss = 0.0

    # do a forward pass to get prediction
    for i in range(Y.shape[0]):
        x, y = X[i], Y[i]                   # get input, output values of each record
        prev_s = np.zeros((hidden_dim, 1))   # here, prev-s is the value of the previous activation of
hidden layer; which is initialized as all zeroes
        for t in range(T):
            new_input = np.zeros(x.shape)    # we then do a forward pass for every timestep in the
sequence
            new_input[t] = x[t]              # for this, we define a single input for that timestep
            mulu = np.dot(U, new_input)
            mulw = np.dot(W, prev_s)
            add = mulw + mulu
            s = sigmoid(add)
            mulv = np.dot(V, s)
            prev_s = s

    # calculate error
        loss_per_record = (y - mulv)**2 / 2
        loss += loss_per_record
    loss = loss / float(y.shape[0])
```

Step 2.2: Check the loss on validation data

We will do the same thing for calculating the loss on validation data (in the same loop):

```
  # check loss on val
```

```
    val_loss = 0.0
    for i in range(Y_val.shape[0]):
        x, y = X_val[i], Y_val[i]
        prev_s = np.zeros((hidden_dim, 1))
        for t in range(T):
            new_input = np.zeros(x.shape)
            new_input[t] = x[t]
            mulu = np.dot(U, new_input)
            mulw = np.dot(W, prev_s)
            add = mulw + mulu
            s = sigmoid(add)
            mulv = np.dot(V, s)
            prev_s = s

        loss_per_record = (y - mulv)**2 / 2
        val_loss += loss_per_record
    val_loss = val_loss / float(y.shape[0])

    print('Epoch: ', epoch + 1, ', Loss: ', loss, ', Val Loss: ', val_loss)
```

You should get the below output:

```
Epoch:  1 , Loss:  [[101185.61756671]] , Val Loss:  [[50591.0340148]]
...
…
```

Step 2.3: Start actual training

We will now start with the actual training of the network. In this, we will first do a forward pass to calculate the errors and a backward pass to calculate the gradients and update them. Let me show you these step-by-step so you can visualize how it works in your mind.

Step 2.3.1: Forward Pass

In the forward pass:

- We first multiply the input with the weights between input and hidden layers
- Add this with the multiplication of weights in the RNN layer. This is because we want to capture the knowledge of the previous timestep
- Pass it through a sigmoid activation function

- Multiply this with the weights between hidden and output layers
- At the output layer, we have a linear activation of the values so we do not explicitly pass the value through an activation layer
- Save the state at the current layer and also the state at the previous timestep in a dictionary

Here is the code for doing a forward pass (note that it is in continuation of the above loop):

```
# train model
 for i in range(Y.shape[0]):
    x, y = X[i], Y[i]

    layers = []
    prev_s = np.zeros((hidden_dim, 1))
    dU = np.zeros(U.shape)
    dV = np.zeros(V.shape)
    dW = np.zeros(W.shape)

    dU_t = np.zeros(U.shape)
    dV_t = np.zeros(V.shape)
    dW_t = np.zeros(W.shape)

    dU_i = np.zeros(U.shape)
    dW_i = np.zeros(W.shape)

    # forward pass
    for t in range(T):
        new_input = np.zeros(x.shape)
        new_input[t] = x[t]
        mulu = np.dot(U, new_input)
        mulw = np.dot(W, prev_s)
        add = mulw + mulu
        s = sigmoid(add)
        mulv = np.dot(V, s)
        layers.append({'s':s, 'prev_s':prev_s})
        prev_s = s
```

Step 2.3.2 : Backpropagate Error

After the forward propagation step, we calculate the gradients at each layer, and backpropagate the errors. We will use truncated backpropagation through time (TBPTT), instead of vanilla backprop. It may sound complex but it's actually pretty straight forward.

The core difference in BPTT versus backprop is that the backpropagation step is done for all the time steps in the RNN layer. So if our sequence length is 50, we will backpropagate for all the timesteps previous to the current timestep.

If you have guessed correctly, BPTT seems very computationally expensive. So instead of backpropagating through all previous timesteps , we backpropagate till x timesteps to save computational power. Consider this ideologically similar to stochastic gradient descent, where we include a batch of data points instead of all the data points.

Here is the code for backpropagating the errors:

```
# derivative of pred
 dmulv = (mulv - y)

 # backward pass
 for t in range(T):
    dV_t = np.dot(dmulv, np.transpose(layers[t]['s']))
    dsv = np.dot(np.transpose(V), dmulv)

    ds = dsv
    dadd = add * (1 - add) * ds

    dmulw = dadd * np.ones_like(mulw)

    dprev_s = np.dot(np.transpose(W), dmulw)


    for i in range(t-1, max(-1, t-bptt_truncate-1), -1):
       ds = dsv + dprev_s
       dadd = add * (1 - add) * ds

       dmulw = dadd * np.ones_like(mulw)
       dmulu = dadd * np.ones_like(mulu)

       dW_i = np.dot(W, layers[t]['prev_s'])
       dprev_s = np.dot(np.transpose(W), dmulw)

       new_input = np.zeros(x.shape)
       new_input[t] = x[t]
       dU_i = np.dot(U, new_input)
       dx = np.dot(np.transpose(U), dmulu)
```

```
            dU_t += dU_i
            dW_t += dW_i

        dV += dV_t
        dU += dU_t
        dW += dW_t
```

Step 2.3.3 : Update weights

Lastly, we update the weights with the gradients of weights calculated. One thing we have to keep in mind is that the gradients tend to explode if you don't keep them in check.This is a fundamental issue in training neural networks, called the exploding gradient problem. So we have to clamp them in a range so that they don't explode. We can do it like this

```
        if dU.max() > max_clip_value:
            dU[dU > max_clip_value] = max_clip_value
        if dV.max() > max_clip_value:
            dV[dV > max_clip_value] = max_clip_value
        if dW.max() > max_clip_value:
            dW[dW > max_clip_value] = max_clip_value



        if dU.min() < min_clip_value:
            dU[dU < min_clip_value] = min_clip_value
        if dV.min() < min_clip_value:
            dV[dV < min_clip_value] = min_clip_value
        if dW.min() < min_clip_value:
            dW[dW < min_clip_value] = min_clip_value

    # update
    U -= learning_rate * dU
    V -= learning_rate * dV
    W -= learning_rate * dW
```

On training the above model, we get this output:

```
Epoch:  1 , Loss:  [[101185.61756671]] , Val Loss:  [[50591.0340148]]
Epoch:  2 , Loss:  [[61205.46869629]] , Val Loss:  [[30601.34535365]]
Epoch:  3 , Loss:  [[31225.3198258]] , Val Loss:  [[15611.65669247]]
Epoch:  4 , Loss:  [[11245.17049551]] , Val Loss:  [[5621.96780111]]
Epoch:  5 , Loss:  [[1264.5157739]] , Val Loss:  [[632.02563908]]
Epoch:  6 , Loss:  [[20.15654115]] , Val Loss:  [[10.05477285]]
```

```
Epoch:  7 , Loss:  [[17.13622839]] , Val Loss:  [[8.55190426]]
Epoch:  8 , Loss:  [[17.38870495]] , Val Loss:  [[8.68196484]]
Epoch:  9 , Loss:  [[17.181681]] , Val Loss:  [[8.57837827]]
Epoch:  10 , Loss:  [[17.31275313]] , Val Loss:  [[8.64199652]]
Epoch:  11 , Loss:  [[17.12960034]] , Val Loss:  [[8.54768294]]
Epoch:  12 , Loss:  [[17.09020065]] , Val Loss:  [[8.52993502]]
Epoch:  13 , Loss:  [[17.17370113]] , Val Loss:  [[8.57517454]]
Epoch:  14 , Loss:  [[17.04906914]] , Val Loss:  [[8.50658127]]
Epoch:  15 , Loss:  [[16.96420184]] , Val Loss:  [[8.46794248]]
Epoch:  16 , Loss:  [[17.017519]] , Val Loss:  [[8.49241316]]
Epoch:  17 , Loss:  [[16.94199493]] , Val Loss:  [[8.45748739]]
Epoch:  18 , Loss:  [[16.99796892]] , Val Loss:  [[8.48242177]]
Epoch:  19 , Loss:  [[17.24817035]] , Val Loss:  [[8.6126231]]
Epoch:  20 , Loss:  [[17.00844599]] , Val Loss:  [[8.48682234]]
Epoch:  21 , Loss:  [[17.03943262]] , Val Loss:  [[8.50437328]]
Epoch:  22 , Loss:  [[17.01417255]] , Val Loss:  [[8.49409597]]
Epoch:  23 , Loss:  [[17.20918888]] , Val Loss:  [[8.5854792]]
Epoch:  24 , Loss:  [[16.92068017]] , Val Loss:  [[8.44794633]]
Epoch:  25 , Loss:  [[16.76856238]] , Val Loss:  [[8.37295808]]
```

Looking good! Time to get the predictions and plot them to get a visual sense of what we've designed.

Step 3: Get predictions

We will do a forward pass through the trained weights to get our predictions:

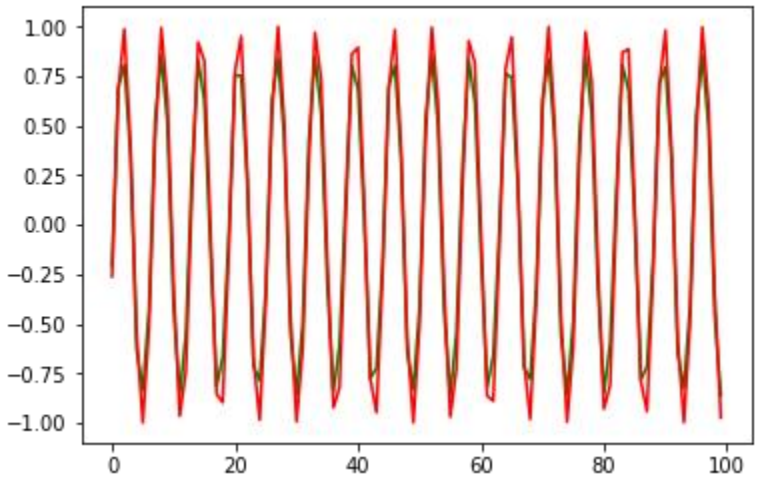
```
preds = []
for i in range(Y.shape[0]):
    x, y = X[i], Y[i]
    prev_s = np.zeros((hidden_dim, 1))
    # Forward pass
    for t in range(T):
        mulu = np.dot(U, x)
        mulw = np.dot(W, prev_s)
        add = mulw + mulu
        s = sigmoid(add)
        mulv = np.dot(V, s)
        prev_s = s

    preds.append(mulv)
```

```
preds = np.array(preds)
```

Plotting these predictions alongside the actual values:

```
plt.plot(preds[:, 0, 0], 'g')
plt.plot(Y[:, 0], 'r')
plt.show()
```



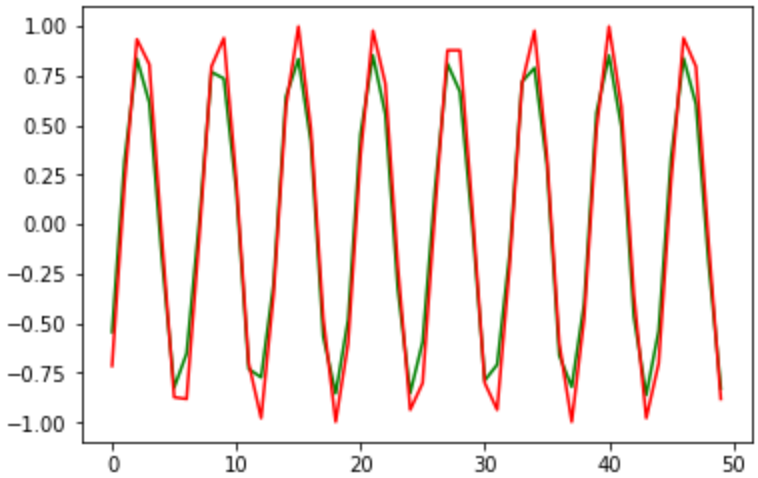
This was on the training data. How do we know if our model didn't overfit? This is where the validation set, which we created earlier, comes into play:

```
preds = []
for i in range(Y_val.shape[0]):
    x, y = X_val[i], Y_val[i]
    prev_s = np.zeros((hidden_dim, 1))
    # For each time step...
    for t in range(T):
        mulu = np.dot(U, x)
        mulw = np.dot(W, prev_s)
        add = mulw + mulu
        s = sigmoid(add)
        mulv = np.dot(V, s)
        prev_s = s

    preds.append(mulv)

preds = np.array(preds)

plt.plot(preds[:, 0, 0], 'g')
plt.plot(Y_val[:, 0], 'r')
plt.show()
```



Not bad. The predictions are looking impressive. The RMSE score on the validation data is respectable as well:

```
from sklearn.metrics import mean_squared_error

math.sqrt(mean_squared_error(Y_val[:, 0] * max_val, preds[:, 0, 0] * max_val))
```

0.127191931509431