

Neural Networks (Shallow/Deep Learning) from Scratch in Python with a Geoscience Example

Ryan (Ali) Mardani

Ryan.Mardani@dataenergy.ca

July 2021

Motivation

Started deep learning using Tensorflow with 12 lines of code!

```
1 model = tf.keras.Sequential(  
2     tf.keras.layers.Dense(128, activation='relu', input_shape=(train_features.shape[-1]),  
3     tf.keras.layers.Dense(128, activation='sigmoid'),  
4     tf.keras.layers.Dense(10))  
5  
6 model.compile(optimizer='adam',  
7     loss=tf.keras.losses.CategoricalCrossentropy(),  
8     metrics=['accuracy'])  
9  
10 model.fit(train_features, train_labels, epochs=10)  
11  
12 model.predict(test_features)
```

Functions and mechanics behind?

Agenda

- Introduction
 - Prior knowledge
 - Database & Problem
 - Artificial Neural Networks
 - Computation Graphs & Derivatives
 - Logistic Regression cost function
- Shallow Learning
 - Parameter Initialization
 - Forward Propagation
 - Compute Loss
 - Backward Propagation
 - Update Parameters
- Deep Learning
 - Similar to shallow learning...
- Prediction and Visualization

To be comfortable with this work:

Fundamental knowledge of:

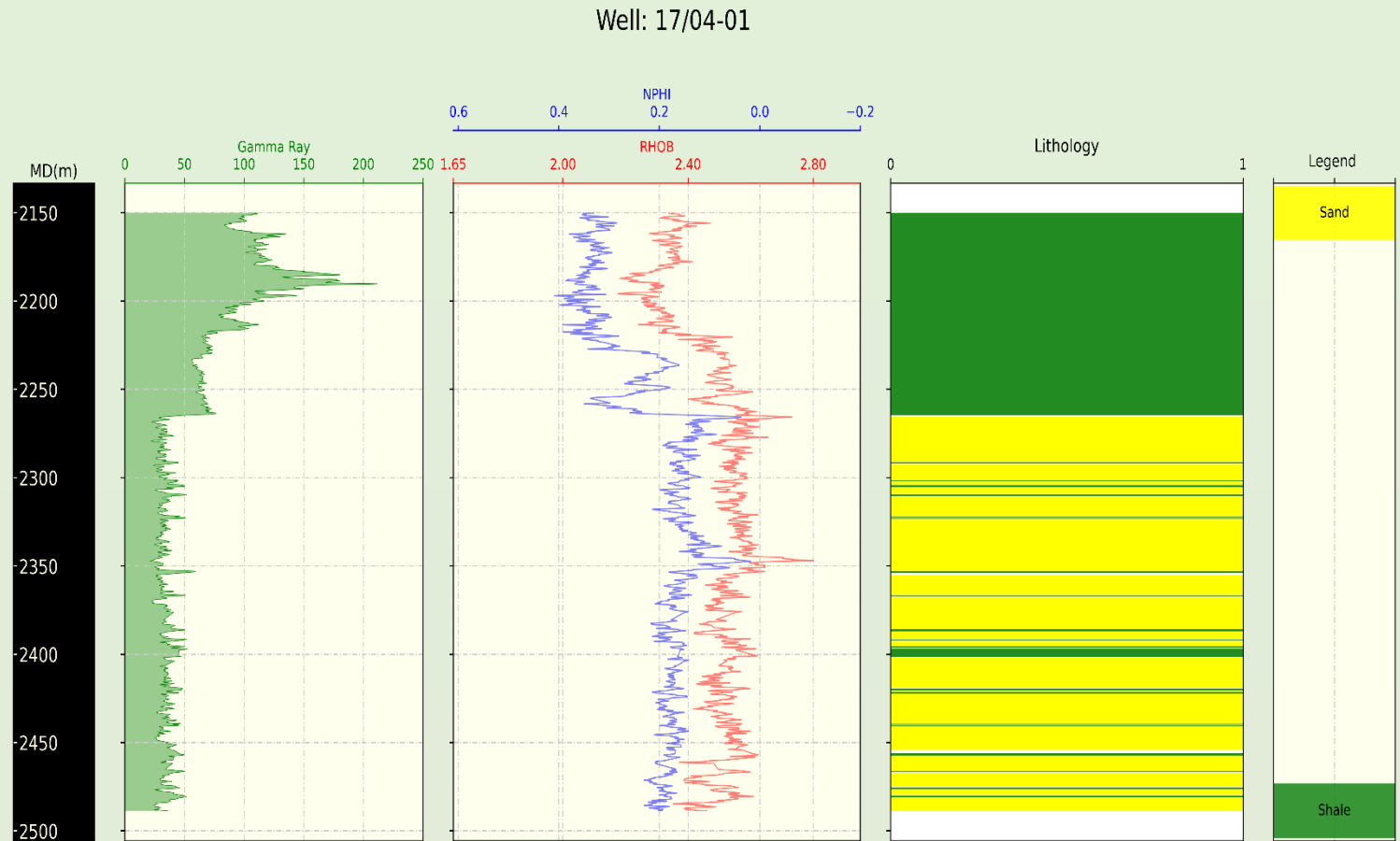
- Linear Algebra (Matrix multiplication)
- Multivariate Calculus (Derivation & Chain rule)
- Python 3 & Numpy
- Neural Networks terminology

Dataset & Problem

For simplicity, binary classification problem (shale/sand) prediction from well logs (FORCE2020 completion data)

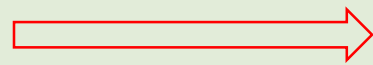
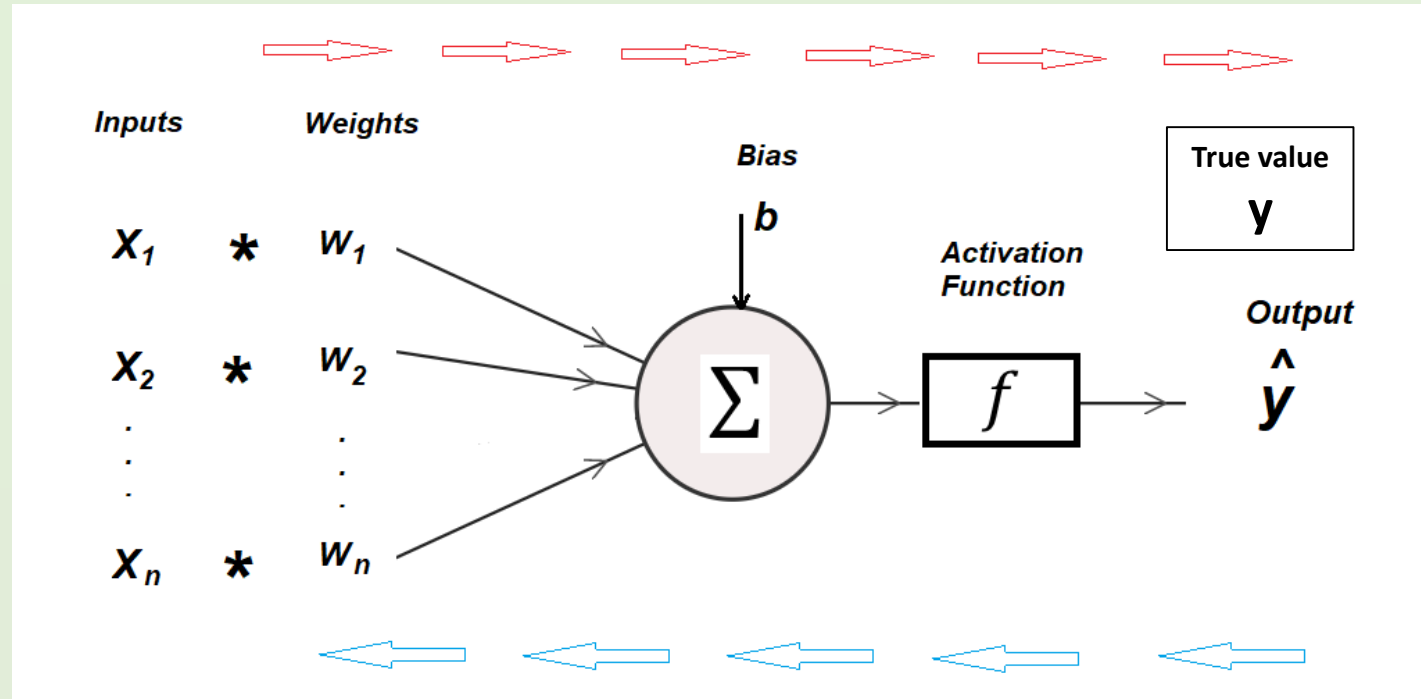
- 3 Wells for Training
- 1 Well for Testing

	DEPTH	WELL	GR	NPHI	RHOB	lithology_name	lithology_num
0	700.116001	15/09-14	76.240852	0.432834	2.137022	Shale	65000
1	700.268001	15/09-14	74.573189	0.426413	2.138363	Shale	65000
2	700.420001	15/09-14	71.091011	0.412909	2.146594	Shale	65000
3	700.572001	15/09-14	68.820503	0.404825	2.156368	Shale	65000
4	700.724001	15/09-14	66.995461	0.384928	2.154258	Shale	65000

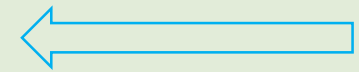


Artificial Neural Networks

- Info (X) receive
- Some operation (W,b) (Sum, Activation)
- Output(\hat{y}) Vs. Target(y)
- Minimize error (\hat{y} , y) adjusting (W,b)
- Learning from Loop



Forward Propagation



Backward Propagation

Computation Graphs & Derivatives

Suppose we have a function J:

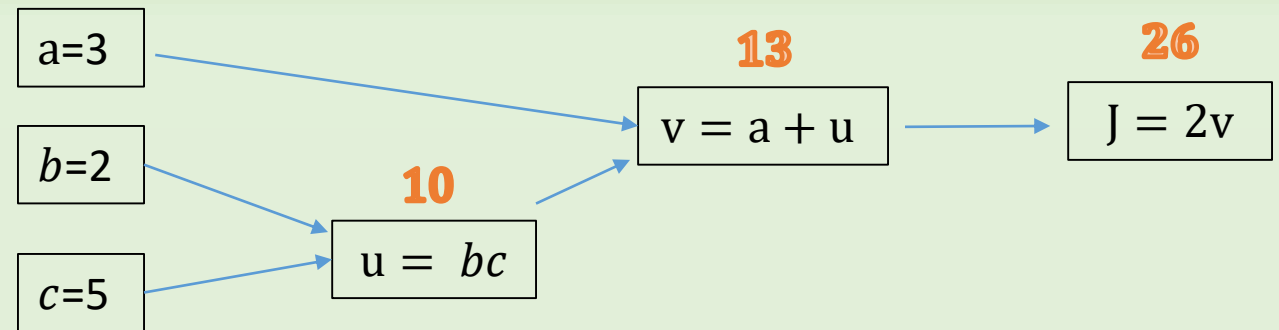
$$J(a, b, c) = 2(a + bc)$$

3 steps to compute:

$$u = bc$$

$$v = a + u$$

$$J = 2v$$



Minimizing J (back propagation) with respect to:

$$\frac{dJ}{dv} = 2$$

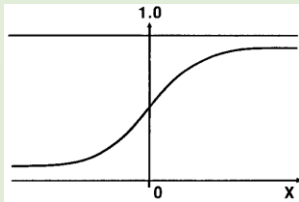
$$\frac{dJ}{da} = \frac{dJ}{dv} \times \frac{dv}{da} = 2 * 1 = 2$$

In NN we usually try to minimize loss function with respect to (W, b) parameters and for convention we will write dw , instead of $\frac{dJ}{dw}$

Logistic Regression cost function

For this binary problem(**shale:0**, **Sand:1**) the most convenient algorithm is logistic regression for classification

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



Loss Function in theory: $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

Cross Entropy Loss Function: $L(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$

- If $y = 1$, $L(\hat{y}, y) = -\log \hat{y}$ and want $\log \hat{y}$ large, (\hat{y}) must be large (1)
- If $y = 0$, $L(\hat{y}, y) = -\log(1 - \hat{y})$ and want large, (\hat{y}) must be small (0)

for entire training example:

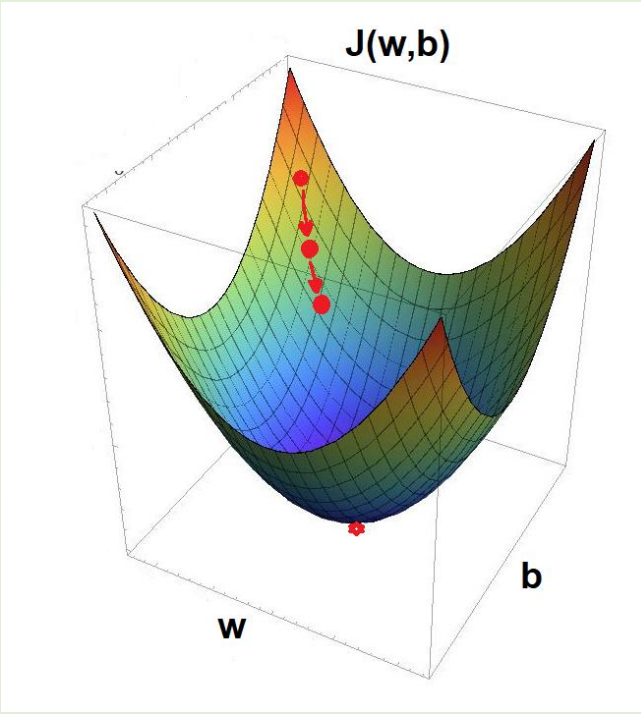
$$\text{Cost Function: } J(\mathbf{w}, \mathbf{b}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

Gradient Descent

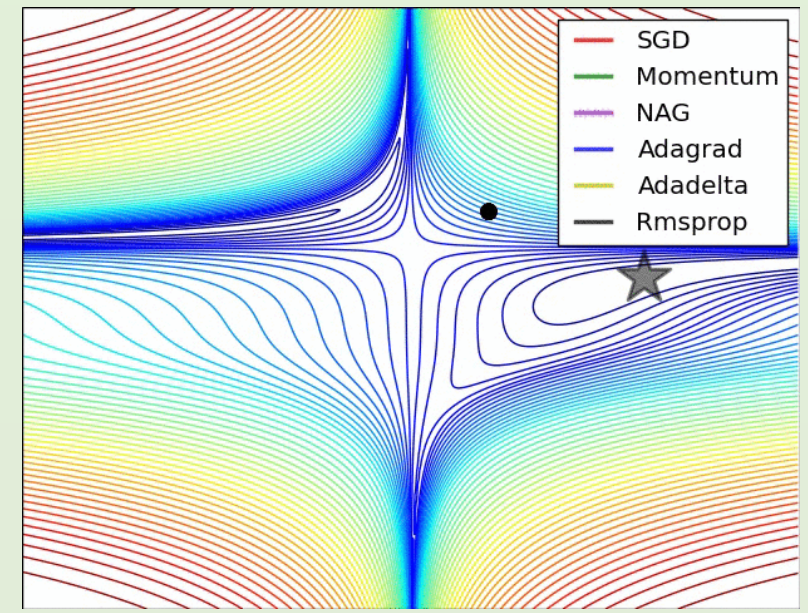
Cost Function: $J(\mathbf{w}, \mathbf{b}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$

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

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



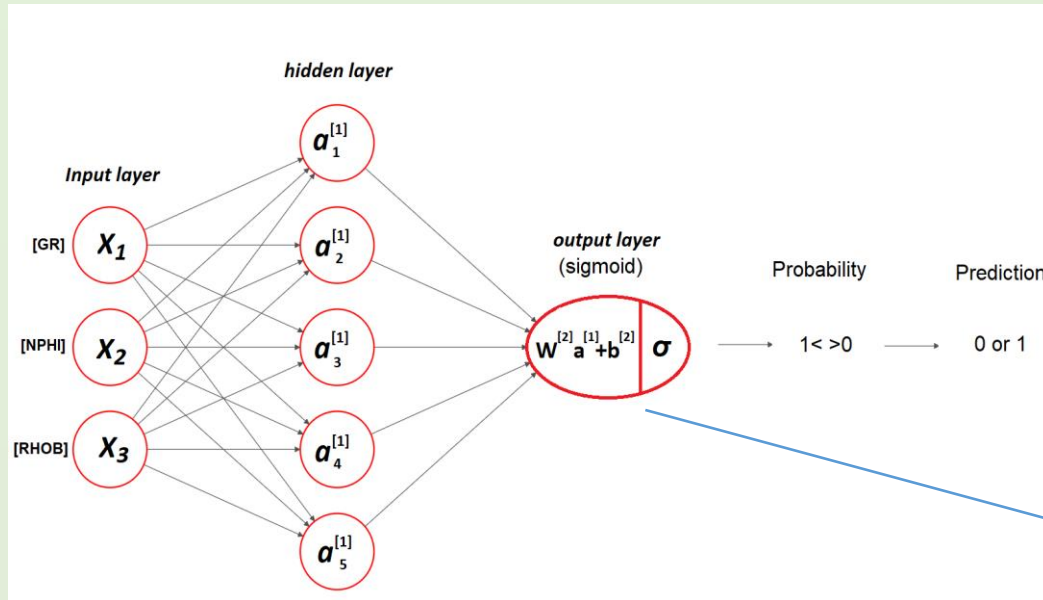
1 iteration, 1 step



Ref: <https://lnkd.in/eyVzZgR>

Shallow Learning

Mainly, shallow network consists of a single hidden layer



$$\begin{array}{c}
 \overbrace{\begin{array}{ccc} W^{[1]} & X & b^{[1]} \end{array}} \\
 \left[\begin{array}{ccc} \cdots & & \\ \vdots & \ddots & \vdots \\ & \cdots & \end{array} \right] \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \\
 (5 \times 3) \quad (3 \times 1) \quad (5 \times 1)
 \end{array}
 \quad
 \begin{array}{c}
 \overbrace{\begin{array}{ccc} W^{[2]} & a^{[1]} & b^{[2]} \end{array}} \\
 \left[\begin{array}{ccc} \cdots & & \end{array} \right] \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \\
 (1 \times 5) \quad (5 \times 1) \quad (1 \times 1)
 \end{array}$$

$$\begin{array}{ccccccc}
 X & \rightarrow & Z^{[1]} = W^{[1]} x + b^{[1]} & \rightarrow & a^{[1]} = \sigma(z^{[1]}) & \rightarrow & Z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} & \rightarrow & a^{[2]} = \sigma(z^{[2]}) & \rightarrow & L(a^{[2]}, y) \\
 W^{[1]} & \nearrow & & \nwarrow & & \nwarrow & W^{[2]} & \nearrow & & \nwarrow & \\
 b^{[1]} & \nearrow & & \nwarrow & & \nwarrow & b^{[2]} & \nearrow & & \nwarrow & \\
 & & dz^{[1]} & & da^{[1]} & & dz^{[2]} & & da^{[2]} & &
 \end{array}$$

Implementation in Python

- Defining the neural network structure
 - layer_sizes

```
1 def layer_sizes(X, Y):
2     """
3     Arguments:
4     X -- input dataset of shape (input size, number of examples)
5     Y -- labels of shape (output size, number of examples)
6
7     Returns:
8     nx -- the size of the input layer
9     nh -- the size of the hidden layer
10    ny -- the size of the output layer
11    """
12    nx = X.shape[0]
13    nh = 5
14    ny = Y.shape[0]
15
16    return (nx, nh, ny)
```

Initialize the model's parameters

```

1 def initialize_parameters(nx, nh, ny):
2     """
3     Argument:
4     nx -- size of the input layer
5     nh -- size of the hidden layer
6     ny -- size of the output layer
7
8     Returns:
9     params -- python dictionary containing your parameters:
10                W1 -- weight matrix of shape (nh, nx)
11                b1 -- bias vector of shape (nh, 1)
12                W2 -- weight matrix of shape (ny, nh)
13                b2 -- bias vector of shape (ny, 1)
14    """
15
16    np.random.seed(2)
17
18    W1 = np.random.randn(nh, nx) * 0.01
19    b1 = np.zeros((nh, 1))
20    W2 = np.random.randn(ny, nh) * 0.01
21    b2 = np.zeros((ny, 1))
22
23    parameters = {"W1": W1,
24                  "b1": b1,
25                  "W2": W2,
26                  "b2": b2}
27
28    return parameters

```

```

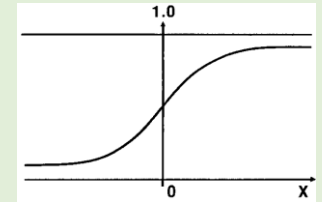
In [13]: 1 initialize_parameters(3,5,1)
Out[13]: {'W1': array([[ -0.00416758, -0.00056267, -0.02136196],
                        [ 0.01640271, -0.01793436, -0.00841747],
                        [ 0.00502881, -0.01245288, -0.01057952],
                        [-0.00909008,  0.00551454,  0.02292208],
                        [ 0.00041539, -0.01117925,  0.00539058]]),
          'b1': array([[0.],
                        [0.],
                        [0.],
                        [0.],
                        [0.])),
          'W2': array([[ -5.96159700e-03, -1.91304965e-04,  1.17500122e-02,
                        -7.47870949e-03,  9.02525097e-05]]),
          'b2': array([[0.]])}

```

Forward propagation

- Define and use the function sigmoid()

$$\text{Sigmoid}(z) = \frac{1}{1 + e^{-z}}$$



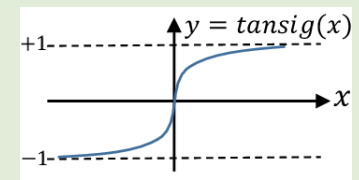
```

1 def sigmoid(x):
2     """
3     Compute sigmoid of x
4
5     Arguments:
6     x : scalar or numpy array
7     """
8     s = 1/(1+np.exp(-x))
9
10    return s

```

- Use the function tanh() from Numpy

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



np.tanh(Z1)

Forward propagation

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} \\ A^{[1]} &= \tanh(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ \hat{Y} &= A^{[2]} = \sigma(Z^{[2]}) \end{aligned}$$

- Retrieve each parameter from the dictionary "parameters" (output of *initialize_parameters()*)
- Implement Forward Propagation. Compute $Z1$, $A1$, $Z2$ and $A2$
- Values needed in the backpropagation are stored in "cache". The cache will be given as an input to the backpropagation function.

```

1 def forward_propagation(X, parameters):
2     """
3     arguments:
4     X : input data size of(nx, m) , m: training samples
5     parameters: dictionary containing parameters(from initialization function)
6
7     returns:
8     A2 : the sigmoid output of the second activation
9     cache: a dict conatining Z1, A1, Z2, and A2
10    """
11    W1 = parameters["W1"]
12    b1 = parameters["b1"]
13    W2 = parameters["W2"]
14    b2 = parameters["b2"]
15
16    #4 steps to calculate A2
17
18    Z1 = np.dot(W1, X) + b1
19    A1 = np.tanh(Z1)
20    Z2 = np.dot(W2, A1) + b2
21    A2 = sigmoid(Z2)
22
23    #store parameters
24
25    cache = {"Z1": Z1,
26            "A1": A1,
27            "Z2": Z2,
28            "A2": A2}
29    return A2, cache

```

Compute the Cost

$$\text{Cost Function: } J(\mathbf{w}, \mathbf{b}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(a^{[2](i)}) + (1 - y^{(i)}) \log(1 - (a^{[2](i)}))]$$

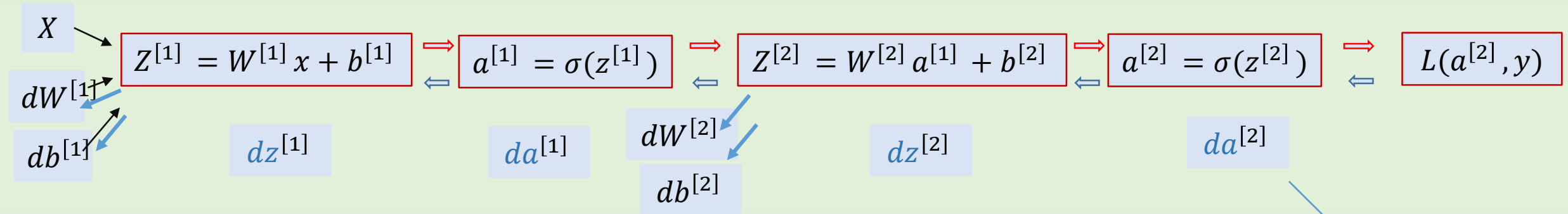
- You can use `np.squeeze()` to remove redundant dimensions (in the case of single float, this will be reduced to a zero-dimension array).

```

1 def compute_cost(A2, Y):
2     """
3     arguments:
4     A2: the sigmoid output of second activation, shape(1, number of examples)
5     Y: the label or target, shape(1, number of examples)
6
7     returns:
8     cost : cross-entropy cost given equation above
9     """
10    m = Y.shape[1]
11
12    log_probs = np.multiply(np.log(A2), Y) + np.multiply((1-Y), np.log(1-A2))
13    cost = - np.sum(log_probs) / m
14
15    cost = float(np.squeeze(cost)) # makes sure cost is the dimension we expect.
16
17    return cost

```

Implement Backpropagation



Summary of gradient descent

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]}a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

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

$$dW^{[1]} = dz^{[1]}x^T$$

$$db^{[1]} = dz^{[1]}$$

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

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

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

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

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

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

Andrew Ng

$$da = \frac{y}{a} - \frac{1-y}{1-a}$$

Backpropagation

To compute dZ^1 we'll need to compute $g^{[1]'}(Z^{[1]})$. Since $g^{[1]}$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]'}(z) = 1 - a^2$. So we can compute $g^{[1]'}(Z^{[1]})$ using $(1 - \text{np.power}(A1, 2))$.

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

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

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

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

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

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

```

1 def backward_propagation(parameters, cache, X, Y):
2     """
3     Arguments:
4     parameters -- python dictionary containing our parameters
5     cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
6     X -- input data of shape (3, number of examples)
7     Y -- "true" labels vector of shape (1, number of examples)
8
9     Returns:
10    grads -- python dictionary containing your gradients with respect to different parameters
11    """
12    m = X.shape[1]
13    # First, retrieve W1 and W2 from the dictionary "parameters".
14    W1 = parameters['W1']
15    W2 = parameters['W2']
16
17    # Retrieve also A1 and A2 from dictionary "cache".
18    A1 = cache['A1']
19    A2 = cache['A2']
20
21    # Backward propagation: calculate dw1, db1, dw2, db2.
22    dZ2 = A2 - Y
23    dW2 = np.dot(dZ2, A1.T)/m
24    db2 = np.sum(dZ2, axis=1, keepdims=True)/m
25    dZ1 = np.dot(W2.T, dZ2)*(1 - np.power(A1, 2))
26    dW1 = np.dot(dZ1, X.T)/m
27    db1 = np.sum(dZ1, axis=1, keepdims=True)/m
28
29
30    grads = {"dw1": dW1,
31            "db1": db1,
32            "dw2": dW2,
33            "db2": db2}
34
35    return grads

```

Update Parameters

General gradient descent rule:

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

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

```

1 def update_parameters(parameters, grads, learning_rate = 1.2):
2     """
3     Arguments:
4     parameters -- python dictionary containing your parameters
5     grads -- python dictionary containing your gradients
6
7     Returns:
8     parameters -- python dictionary containing your updated parameters
9     """
10    # Retrieve each parameter from the dictionary "parameters"
11    w1 = parameters["w1"]
12    b1 = parameters["b1"]
13    w2 = parameters["w2"]
14    b2 = parameters["b2"]
15
16    # Retrieve each gradient from the dictionary "grads"
17    dw1 = grads["dw1"]
18    db1 = grads["db1"]
19    dw2 = grads["dw2"]
20    db2 = grads["db2"]
21
22    # Update rule for each parameter
23    w1 = w1 - learning_rate * dw1
24    b1 = b1 - learning_rate * db1
25    w2 = w2 - learning_rate * dw2
26    b2 = b2 - learning_rate * db2
27
28    parameters = {"w1": w1,
29                  "b1": b1,
30                  "w2": w2,
31                  "b2": b2}
32
33    return parameters

```

NN Model Function

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

```

1 def nn_model(X, Y, nh, num_iterations = 1000, print_cost=False):
2     """
3     Arguments:
4     X -- dataset of shape (3, number of examples)
5     Y -- labels of shape (1, number of examples)
6     nh -- size of the hidden layer
7     num_iterations -- Number of iterations in gradient descent loop
8     print_cost -- if True, print the cost every 1000 iterations
9
10    Returns:
11    parameters -- parameters learnt by the model. They can then be used to predict.
12    """
13    np.random.seed(3)
14    nx = layer_sizes(X, Y)[0]
15    ny = layer_sizes(X, Y)[2]
16
17    # Initialize parameters
18    parameters = initialize_parameters(nx, nh, ny)
19
20    # Loop (gradient descent)
21    for i in range(0, num_iterations):
22
23        # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
24        A2, cache = forward_propagation(X, parameters)
25
26        # Cost function. Inputs: "A2, Y". Outputs: "cost".
27        cost = compute_cost(A2, Y)
28
29        # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
30        grads = backward_propagation(parameters, cache, X, Y)
31
32        # Gradient descent parameter update. Inputs: "parameters, grads". Outputs: "parameters".
33        parameters = update_parameters(parameters, grads, learning_rate = 1.2)
34
35        # Print the cost every 1000 iterations
36        if print_cost and i % 1000 == 0:
37            print ("Cost after iteration %i: %f" % (i, cost))
38
39    return parameters

```

Shallow Learning Progress

It seems shallow network model cannot capture complexity of dataset.

```
1 parameters = nn_model(X, Y, 5, num_iterations=10000, print_cost=True)
2 print("W1 = " + str(parameters["W1"]))
3 print("b1 = " + str(parameters["b1"]))
4 print("W2 = " + str(parameters["W2"]))
5 print("b2 = " + str(parameters["b2"]))
```

Cost after iteration 0: 0.695391

Cost after iteration 1000: 0.681194

Cost after iteration 2000: 0.681220

Cost after iteration 3000: 0.681209

Cost after iteration 4000: 0.681212

Cost after iteration 5000: 0.681211

Cost after iteration 6000: 0.681209

Cost after iteration 7000: 0.681208

Cost after iteration 8000: 0.681210

Cost after iteration 9000: 0.681210

W1 = [[0.21862145 -0.00559108 -0.05404105]
[-0.20732471 -0.03159421 -0.09176682]
[0.20823183 -0.00233738 0.05082625]
[0.17895134 0.01195833 0.06376946]
[-0.31918833 -0.0068341 0.03319447]]

b1 = [[-0.01688963]
[-0.04186761]
[0.0307843]
[0.0203578]
[0.01457726]]

W2 = [[-0.19581836 -0.12901404 0.13005298 -0.13160344 0.18943112]]

b2 = [[-0.05312095]]

Test the Shallow Model

```
1 pred_train_shallow = predict(X, Y, parameters)
```

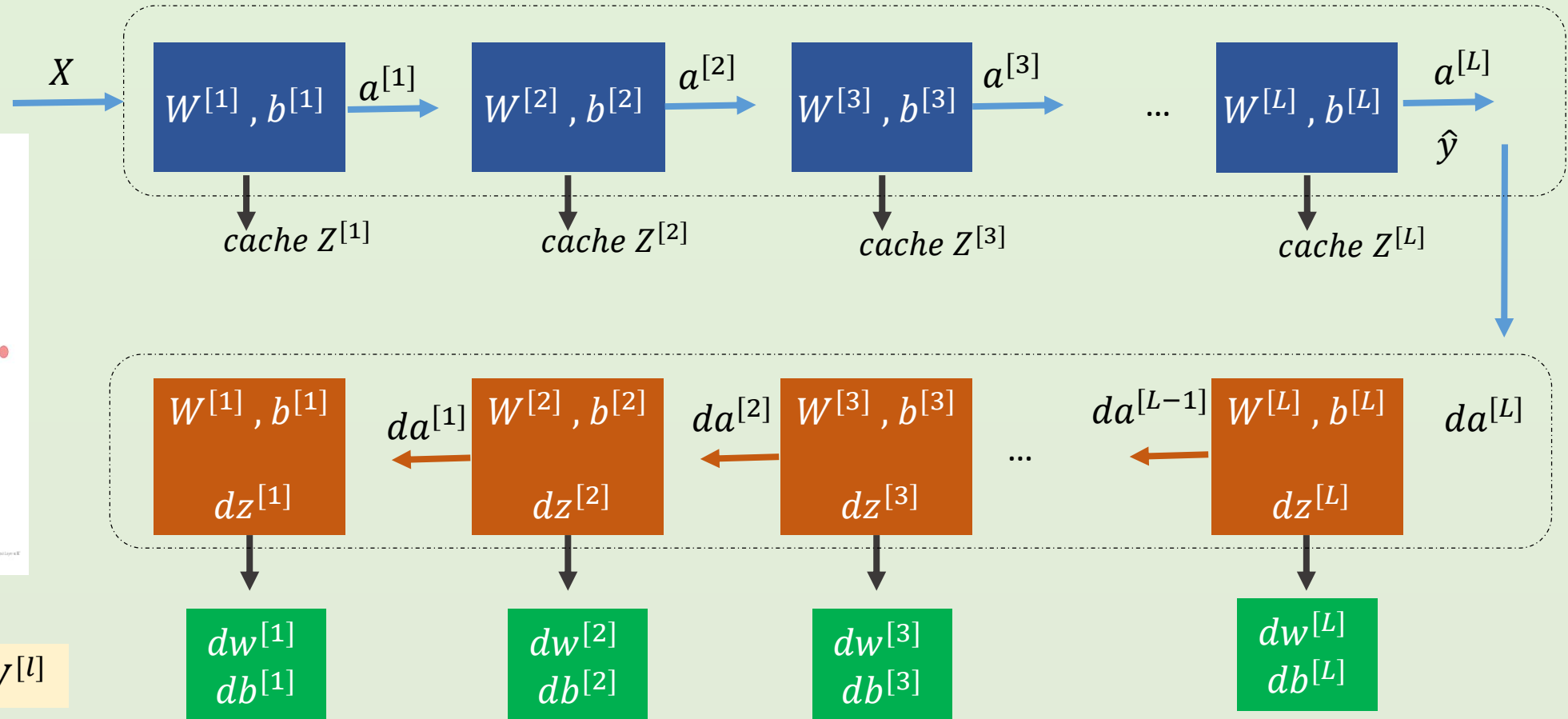
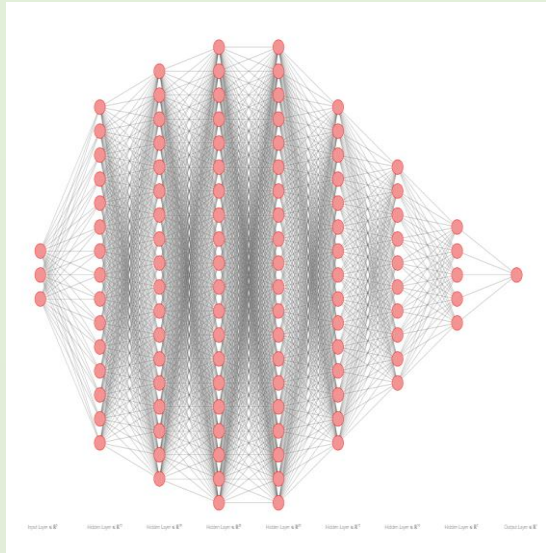
Accuracy: 0.5771010023130301

```
1 pred_test_shallow = predict(XX, YY, parameters)
```

Accuracy: 0.40293225480283107

```
1 def predict(X, y, parameters):
2     """
3     This function is used to predict the results of a L-layer neural network.
4
5     Arguments:
6     X -- data set of examples you would like to label
7     parameters -- parameters of the trained model
8
9     Returns:
10    p -- predictions for the given dataset X
11    """
12    m = X.shape[1]
13    n = len(parameters) // 2 # number of layers in the neural network
14    p = np.zeros((1,m))
15
16    # Forward propagation
17    probas, caches = forward_propagation(X, parameters)
18
19    # convert probas to 0/1 predictions
20    for i in range(0, probas.shape[1]):
21        if probas[0,i] > 0.5:
22            p[0,i] = 1
23        else:
24            p[0,i] = 0
25
26    print("Accuracy: " + str(np.sum((p == y)/m)))
27
28    return p
```

Deep Learning

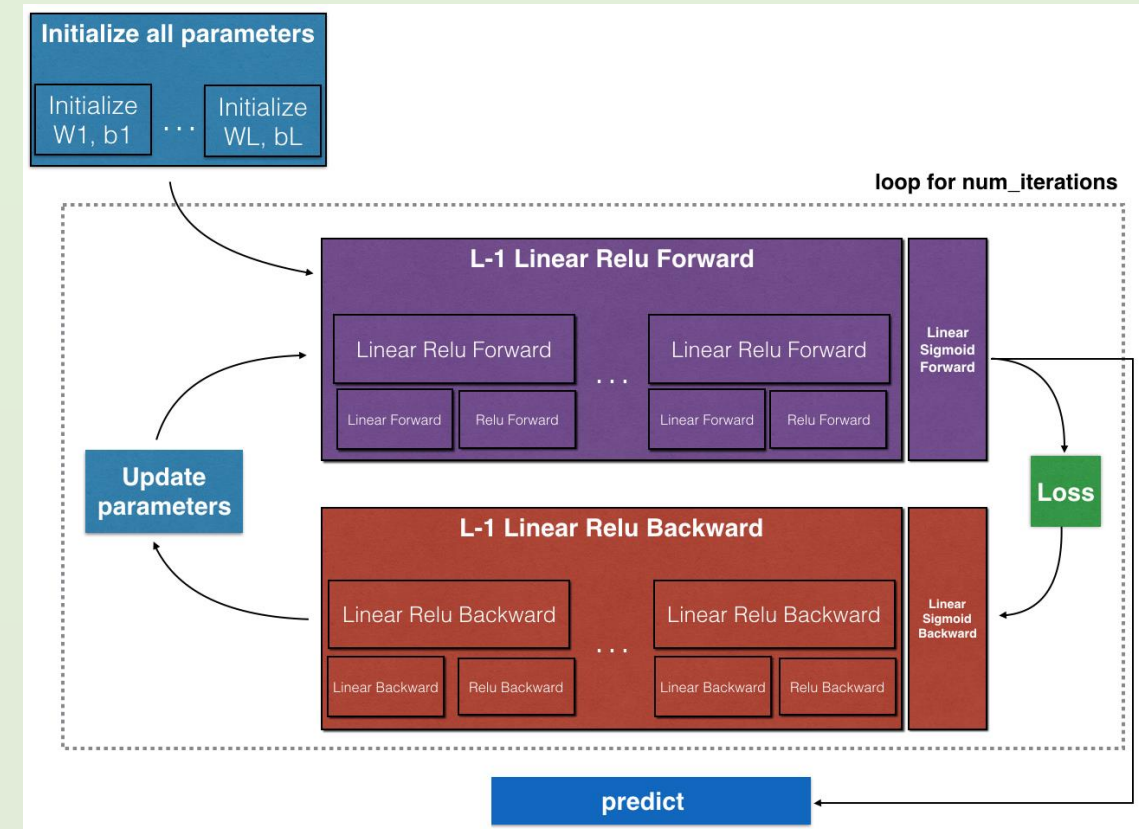


$$W^{[l]} := W^{[l]} - \alpha dW^{[l]}$$

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

Deep Learning

- Initialize the parameters for an L -layer neural network
- Forward Propagation
 - Complete linear part (Resulting $Z^{[1]}$)
 - Activation function(ReLU/Sigmoid)
 - Combine two previous into LINEAR->ACTIVATION forward function
 - Stack the forward function $L-1$ time and add a LINEAR-> SIGMOID at the end
- Compute the loss
- Backward propagation
 - Complete linear part
 - The gradient of the ACTIVATE function (relu_backward/sigmoid_backward)
 - Combine two previous into LINEAR->ACTIVATION backward function
 - Stack the backward function $L-1$ time and add a LINEAR-> SIGMOID at the end
- Update the Parameters



Initialize the parameters for an L -layer NN

- Initialization for a deeper L -layer NN is more complicated because there are many more weight matrices and bias vectors

$n^{[l]}$: number of nodes(units) in layer l

m : number of features (3 here)

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, 3)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 3)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 3)$
\vdots	\vdots	\vdots	\vdots	\vdots
Layer $L-1$	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 3)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 3)$

Initialize the parameters for an L -layer NN

```

1 def initialize_parameters_deep(layer_dims):
2     """
3     Arguments:
4     layer_dims -- python array (list) containing the dimensions of each layer in our network
5
6     Returns:
7     parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
8         W1 -- weight matrix of shape (layer_dims[1], layer_dims[1-1])
9         b1 -- bias vector of shape (layer_dims[1], 1)
10    """
11
12    np.random.seed(1)
13    parameters = {}
14    L = len(layer_dims)          # number of layers in the network
15
16    for l in range(1, L):
17        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) / np.sqrt(layer_dims[l-1])
18        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))
19
20
21    return parameters

```

For 3 input features, 5 nodes in first hidden layer, 3 nodes in second layer, and 1 output

```

1 parameters = initialize_parameters_deep([3,5,3,1])
2
3 print("W1 = " + str(parameters["W1"]))
4 print("b1 = " + str(parameters["b1"]))
5 print("W2 = " + str(parameters["W2"]))
6 print("b2 = " + str(parameters["b2"]))
7 print("W3 = " + str(parameters["W3"]))
8 print("b3 = " + str(parameters["b3"]))

```

W1 = [[0.93781623 -0.35319773 -0.3049401]
[-0.61947872 0.49964333 -1.32879399]
[1.00736754 -0.43948301 0.18419731]
[-0.14397405 0.84414841 -1.18942279]
[-0.18614766 -0.22173389 0.65458209]]
b1 = [[0.]
[0.]
[0.]
[0.]]
W2 = [[-0.49188633 -0.07711224 -0.39259022 0.01887856 0.26064289]
[-0.49221186 0.51193601 0.40320363 0.2247223 0.40287503]
[-0.30577239 -0.05495818 -0.41848881 -0.11980319 0.23718218]]
b2 = [[0.]
[0.]
[0.]]
W3 = [[-0.39933052 -0.22906576 -0.39673934]]
b3 = [[0.]]

Linear Forward

Now, we'll complete three functions in this order:

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

```
1 def linear_forward(A, W, b):
2     """
3     Implement the linear part of a layer's forward propagation.
4
5     Arguments:
6     A -- activations from previous layer (or input data): (size of previous layer, number of examples)
7     W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
8     b -- bias vector, numpy array of shape (size of the current layer, 1)
9
10    Returns:
11    Z -- the input of the activation function, also called pre-activation parameter
12    cache -- a python dictionary containing "A", "W" and "b" ; stored for computing the backward pass efficiently
13    """
14
15    Z = W.dot(A) + b
16
17    cache = (A, W, b)
18
19    return Z, cache
```

Activation Functions

Sigmoid: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$

ReLU is $A = \text{RELU}(Z) = \max(0, Z)$.

```

1 def sigmoid(Z):
2     """
3     Implements the sigmoid activation in numpy
4
5     Arguments:
6     Z -- numpy array of any shape
7
8     Returns:
9     A -- output of sigmoid(z), same shape as Z
10    cache -- returns Z as well, useful during backpropagation
11    """
12    A = 1/(1+np.exp(-Z))
13    cache = Z
14
15    return A, cache
16
17
18 def relu(Z):
19     """
20     Implement the RELU function.
21
22     Arguments:
23     Z -- Output of the linear layer, of any shape
24
25     Returns:
26     A -- Post-activation parameter, of the same shape as Z
27     cache -- a python dictionary containing "A" ; stored for computing the backward pass efficiently
28     """
29    A = np.maximum(0,Z)
30
31    cache = Z
32    return A, cache

```

Linear-Activation Forward

Now, we'll complete three functions in this order:

- ~~LINEAR~~
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- ~~[LINEAR -> RELU] x (L-1) -> LINEAR -> SIGMOID (whole model)~~

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

```

1 def linear_activation_forward(A_prev, W, b, activation):
2     """
3     Implement the forward propagation for the LINEAR->ACTIVATION layer
4
5     Arguments:
6     A_prev -- activations from previous layer (or input data): (size of previous layer, number of examples)
7     W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
8     b -- bias vector, numpy array of shape (size of the current layer, 1)
9     activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"
10
11     Returns:
12     A -- the output of the activation function, also called the post-activation value
13     cache -- a python dictionary containing "linear_cache" and "activation_cache";
14             stored for computing the backward pass efficiently
15     """
16
17     if activation == "sigmoid":
18         # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
19         Z, linear_cache = linear_forward(A_prev, W, b)
20         A, activation_cache = sigmoid(Z)
21
22     elif activation == "relu":
23         # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
24         Z, linear_cache = linear_forward(A_prev, W, b)
25         A, activation_cache = relu(Z)
26
27     cache = (linear_cache, activation_cache)
28
29     return A, cache

```

L-Layer Model

Now, we'll complete three functions in this order:

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- [LINEAR -> RELU] \times (L-1) -> LINEAR -> SIGMOID (whole model)

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

```

1 def L_model_forward(X, parameters):
2     """
3     Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation
4
5     Arguments:
6     X -- data, numpy array of shape (input size, number of examples)
7     parameters -- output of initialize_parameters_deep()
8
9     Returns:
10    AL -- last post-activation value
11    caches -- list of caches containing:
12                every cache of linear_relu_forward() (there are L-1 of them, indexed from 0 to L-2)
13                the cache of linear_sigmoid_forward() (there is one, indexed L-1)
14    """
15
16    caches = []
17    A = X
18    L = len(parameters) // 2          # number of layers in the neural network
19
20    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
21    for l in range(1, L):
22        A_prev = A
23        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)], parameters['b' + str(l)], activation = "relu")
24        caches.append(cache)
25
26    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
27    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], parameters['b' + str(L)], activation = "sigmoid")
28    caches.append(cache)
29
30
31
32    return AL, caches

```

Cost Function

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

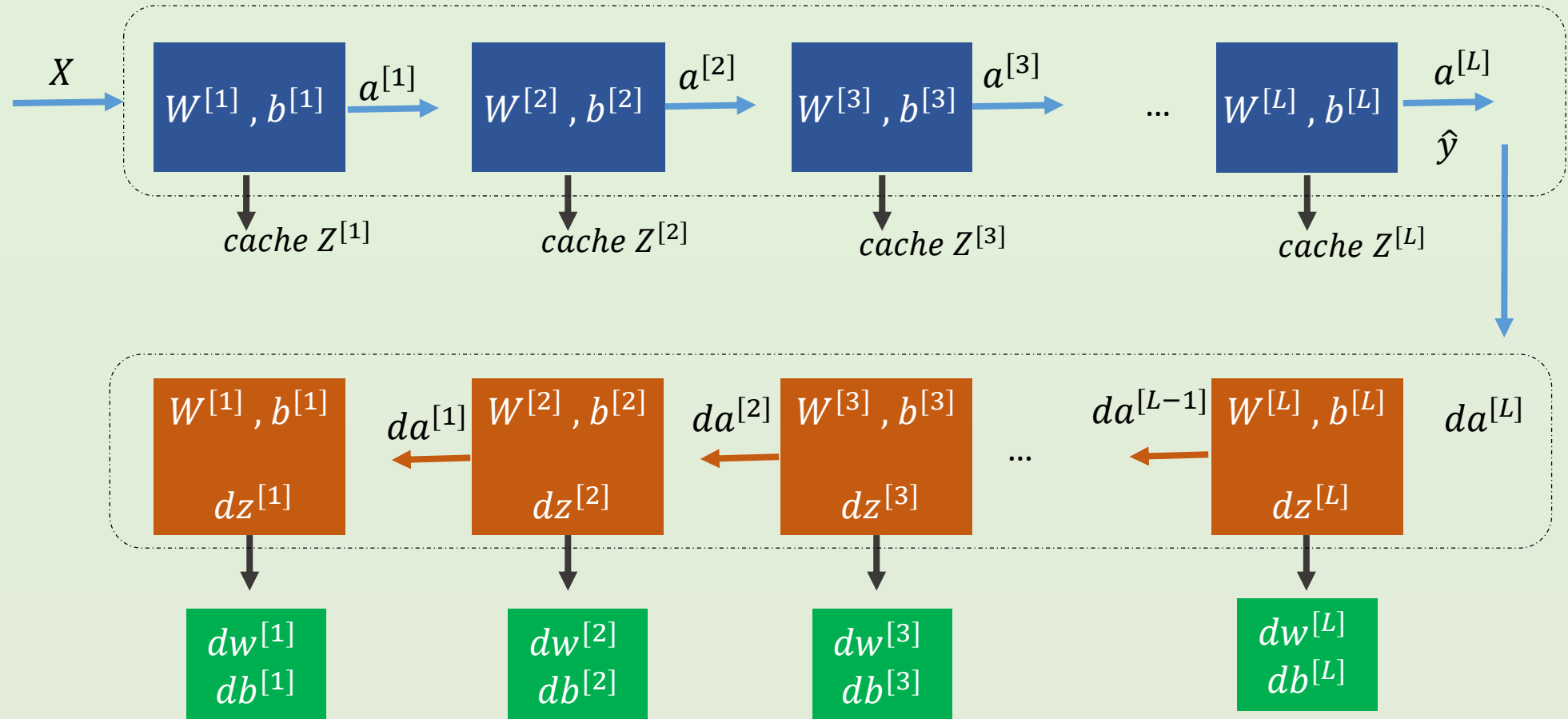
$$\text{Cost Function: } J(\mathbf{w}, \mathbf{b}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(a^{[L](i)}) + (1 - y^{(i)}) \log(1 - (a^{[L](i)})]$$

```

1 def compute_cost(AL, Y):
2     """
3     Implement the cost function defined by equation (7).
4
5     Arguments:
6     AL -- probability vector corresponding to your label predictions, shape (1, number of examples)
7     Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number of examples)
8
9     Returns:
10    cost -- cross-entropy cost
11    """
12
13    m = Y.shape[1]
14
15    # Compute loss from aL and y.
16    cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))
17
18    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).
19
20    return cost

```

Backward Propagation Module



In backpropagation, we will calculate gradient of the loss function with respect to parameters(W, b)

Linear Backward

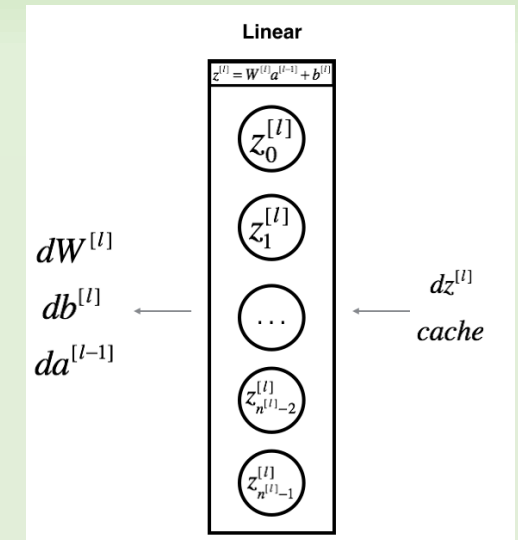
Now, we'll complete three functions in this order:

- LINEAR backward
- ~~LINEAR -> ACTIVATION backward where ACTIVATION will be either ReLU or Sigmoid.~~
- ~~[LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID backward (whole model)~~

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

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

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



```

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

```


Linear-Activation Backward

- LINEAR backward
- LINEAR -> ACTIVATION backward where ACTIVATION will be either ReLU or Sigmoid.
- ~~[LINEAR -> RELU] × (L - 1) -> LINEAR -> SIGMOID backward (whole model)~~

```

1 def relu_backward(dA, cache):
2     """
3     Implement the backward propagation for a single RELU unit.
4
5     Arguments:
6     dA -- post-activation gradient, of any shape
7     cache -- 'Z' where we store for computing backward propagation efficiently
8
9     Returns:
10    dZ -- Gradient of the cost with respect to Z
11    """
12    Z = cache
13    dZ = np.array(dA, copy=True) # just converting dz to a correct object.
14
15    # When z <= 0, you should set dz to 0 as well.
16    dZ[Z <= 0] = 0
17
18    return dZ
19
20 def sigmoid_backward(dA, cache):
21     """
22     Implement the backward propagation for a single SIGMOID unit.
23
24     Arguments:
25     dA -- post-activation gradient, of any shape
26     cache -- 'Z' where we store for computing backward propagation efficiently
27
28     Returns:
29     dZ -- Gradient of the cost with respect to Z
30     """
31    Z = cache
32    s = 1/(1+np.exp(-Z))
33    dZ = dA * s * (1-s)
34
35    return dZ

```

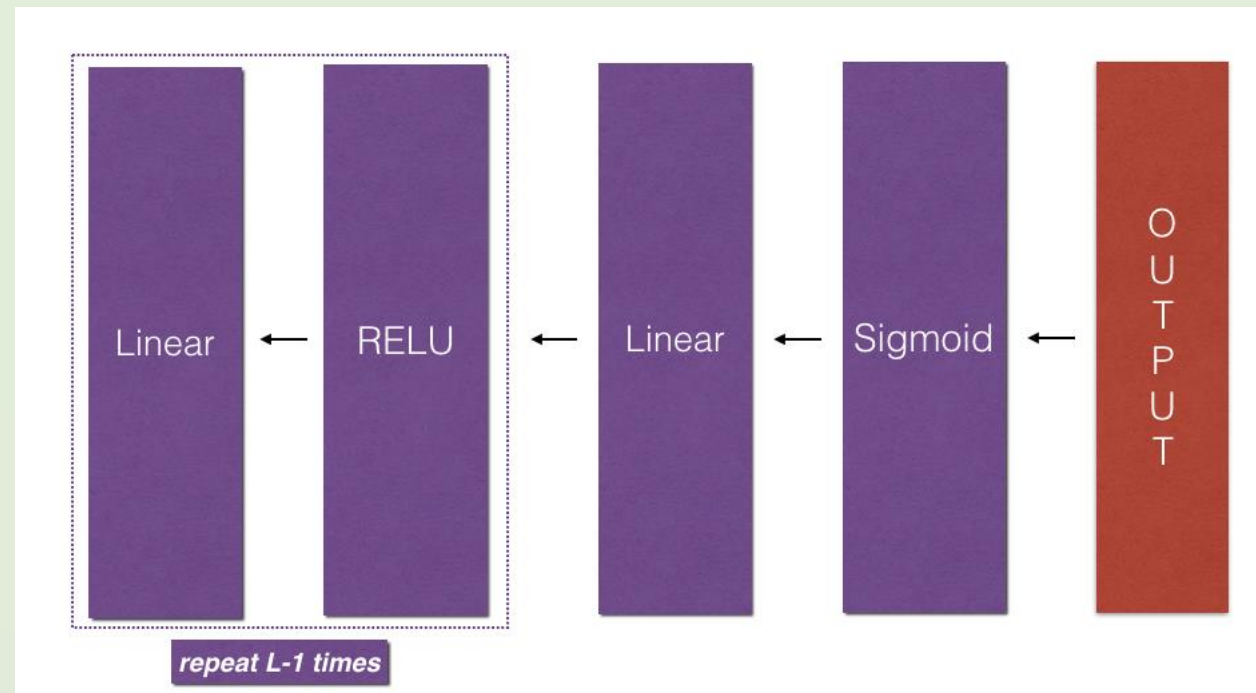
```

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

```

L-Model Backward

- Now we will implement the backward function for the whole network!



L_model_backward

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

```

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

```

Update Parameters

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

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

```

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

```

L_layer_model

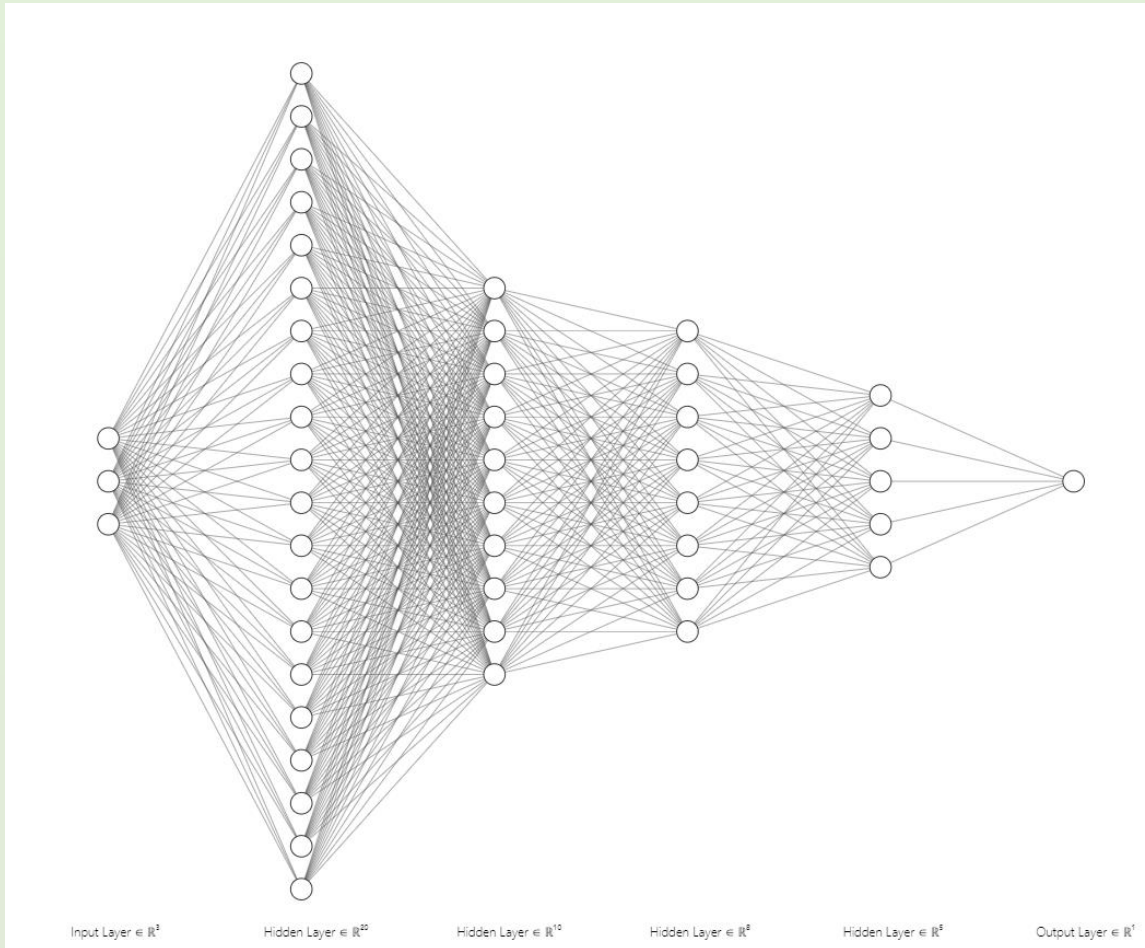
- Combine helper functions in order

```

1 def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):
2     """
3     Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.
4
5     Arguments:
6     X -- data, numpy array of shape (num_px * num_px * 3, number of examples)
7     Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of examples)
8     layers_dims -- list containing the input size and each layer size, of length (number of layers + 1).
9     learning_rate -- learning rate of the gradient descent update rule
10    num_iterations -- number of iterations of the optimization loop
11    print_cost -- if True, it prints the cost every 100 steps
12
13    Returns:
14    parameters -- parameters learnt by the model. They can then be used to predict.
15    """
16    np.random.seed(1)
17    costs = [] # keep track of cost
18
19    parameters = initialize_parameters_deep(layers_dims)
20
21    # Loop (gradient descent)
22    for i in range(0, num_iterations):
23
24        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
25        AL, caches = L_model_forward(X, parameters)
26
27        # Compute cost.
28        cost = compute_cost(AL, Y)
29
30        # Backward propagation.
31        grads = L_model_backward(AL, Y, caches)
32
33        # Update parameters.
34        parameters = update_parameters(parameters, grads, learning_rate)
35
36        # Print the cost every 1000 iterations
37        if print_cost and i % 1000 == 0 or i == num_iterations - 1:
38            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
39        if i % 1000 == 0 or i == num_iterations:
40            costs.append(cost)
41    return parameters, costs

```

Deep Learning Progress

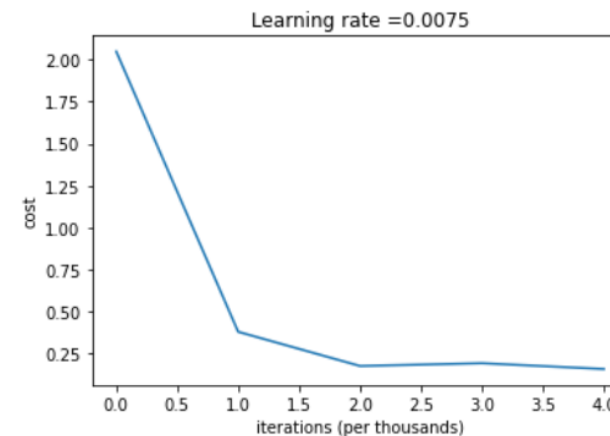


```
def plot_costs(costs, learning_rate=0.0075):
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per thousands)')
    plt.title("Learning rate = " + str(learning_rate))
    plt.show()
```

```
1 layers_dims = [3, 20, 10, 8, 5, 1] # L-layer model
2 learning_rate = 0.0075
```

```
1 parameters, costs = L_layer_model(X, Y, layers_dims, num_iterations = 5000, print_cost = True)
2 plot_costs(costs, learning_rate)
3
```

Cost after iteration 0: 2.0469973401647983
 Cost after iteration 1000: 0.38014323388868226
 Cost after iteration 2000: 0.17677566301781922
 Cost after iteration 3000: 0.1933672322208049
 Cost after iteration 4000: 0.15895567008556574
 Cost after iteration 4999: 0.12442577754613114



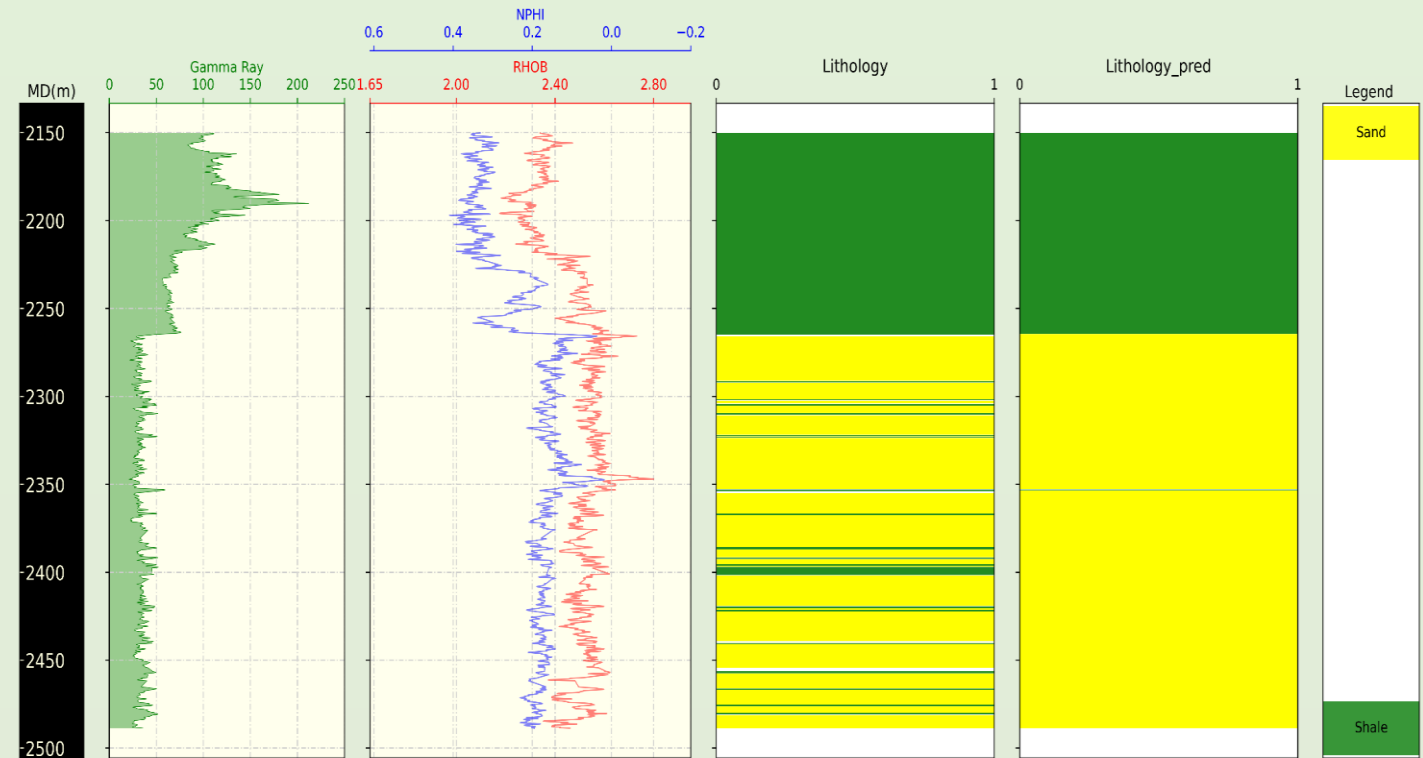
predict

Well: 17/04-01

```
1 def predict(X, y, parameters):
2     """
3     This function is used to predict the results of a L-layer neural network.
4
5     Arguments:
6     X -- data set of examples you would like to label
7     parameters -- parameters of the trained model
8
9     Returns:
10    p -- predictions for the given dataset X
11    """
12
13    m = X.shape[1]
14    n = len(parameters) // 2 # number of layers in the neural network
15    p = np.zeros((1,m))
16
17    # Forward propagation
18    probas, caches = L_model_forward(X, parameters)
19
20
21    # convert probas to 0/1 predictions
22    for i in range(0, probas.shape[1]):
23        if probas[0,i] > 0.5:
24            p[0,i] = 1
25        else:
26            p[0,i] = 0
27
28    print("Accuracy: " + str(np.sum((p == y)/m)))
29
30    return p
```

```
1 pred_train = predict(X, Y, parameters)
```

Accuracy: 0.9151888974556668



Final Thoughts

- In this work, the procedure and implementation to establish and run a neural network model considered as a first importance.
- This is just the simplest form of NN models. Some important hyper-parameters such as L1, L2 regularization can be added.
- Building a NN model from scratch in Python/Numpy can help us to understand the mechanics and fundamentals of this approach.

Reference

- www.deeplearning.ai
- <https://zenodo.org/record/4351156#.YLfBH6hKjUr>