**Logistic Regression:**

Given an input feature x what is y\_hat (output) where x is vector of size nx.

Parameters: w🡪vector of size nx ,m , b🡪number

z = wTx+b y\_hat = sigmoid(z)

Cost function:

When there are m training examples then we need output for all and we compute cost accordingly.

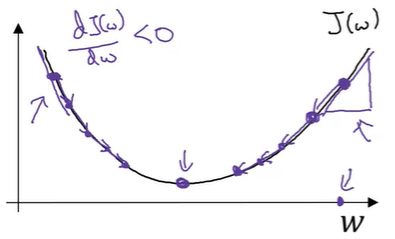
L(y\_hat, y) = - (y\*log(y\_hat) + (1-y)\*( log(1-y\_hat) ) )

cf🡪J(w,b) = -sum(L( y\_hat(i), y(i) ))/m [from i=1 to m]

Loss function measures how well your algorithm outputs y\_hat(i) on training examples compares to y(i)

So, cost function measures how well w and b are doing on your training set.

Minimize J(w,b)



To minimize J(w,b) w should as shown.

To compute this we use Gradient Descent select an random value for w now

Repeat {

w = w - alpha\*dw } where dw🡪 dJ(w)/dw = 0 (partial derivative should be zero ) similarly for ‘b’

Z = WTX + b = np.dot(W.T, X) +b

A = sigmoid(Z)

dZ = A – Y

dW = XdZT/m = np.dot(X, dZ.T)/m

db = np.sum(dZ)/m

W = W – alpha\*dW

b = b – alpha\*db

**Steps:**

* **Pre-processing:**

1. Load the dataset, divide train and test data.
2. F latten the data ( column vector )
3. One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255.

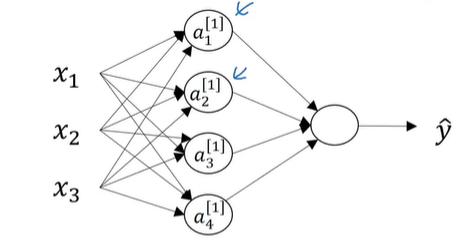
* **Building parts of algorithm:**

1. Define model structure(such as number of parameters).
2. Initialize model parameters.
3. Loop:
4. Calculate current loss (Forward propagation)
5. Calculate current gradient (Backward propagation)
6. Update parameters (gradient descent).
7. Then implement predict() using updated values and set threshold to convert predictions to 1 or 0 ( here if x>0.5 then 1 else 0 ).
8. Make model using building blocks

**Neural Network with one hidden layer:**

**Layers in neural network**🡪

* Input layer
* Hidden layer
* Output layer



**Parameters:**

W[1], W[2], b[1], b[2]

Cost function: J(w,b) = -sum(L( y\_hat(i), y(i) ))/m [from i=1 to m]

dZ[2] = A[2] – Y

dW[2] = dZ[2]A[1]T

db[2] = np.sum(dZ[2], axis = 1, keepdims =True)/m

dZ[1] = W[2]TdZ[2] \* g[1]’(Z[1])

dW[1] = dZ[1]XT/m

db[1] = np.sum(dZ[1], axis = 1, keepdims =True)/m

Z1 = np.dot(W1, X) + b1

A1 = np.tanh(Z1)

Z2 = np.dot(W2, A1) +b2

A2 = sigmoid(Z2)

logprobs = np.multiply(Y,np.log(A2))+(1-Y)\*np.log(1-A2)

cost = -np.sum(logprobs)/m

cost = float(np.squeeze(cost))

dZ2 = A2-Y

dW2 = np.dot(dZ2,A1.T)/m

db2 = np.sum(dZ2, axis = 1, keepdims = True)/m

dZ1 = np.dot(W2.T,dZ2)\*(1-np.power(A1,2))

dW1 = np.dot(dZ1, X.T)/m

db1 = np.sum(dZ1, axis = 1, keepdims = True)/m

**Random Initialization:**

Initializing the weights to zero would result in each neuron computing the same calculations to avoid this we initialise the weights randomly using W = np.random.rand( nx, m)\*0.01. The 0.01 here is used to make the weight small as on large values activation functions output would be close to zero. But ‘b’ can be initialized with zeros

**Steps:**

* Define the neural network structure ( # of input units, # of hidden units, etc).
* Initialize the model's parameters
* Loop:

1. Implement forward propagation
2. Compute loss
3. Implement backward propagation to get the gradients
4. Update parameters (gradient descent)