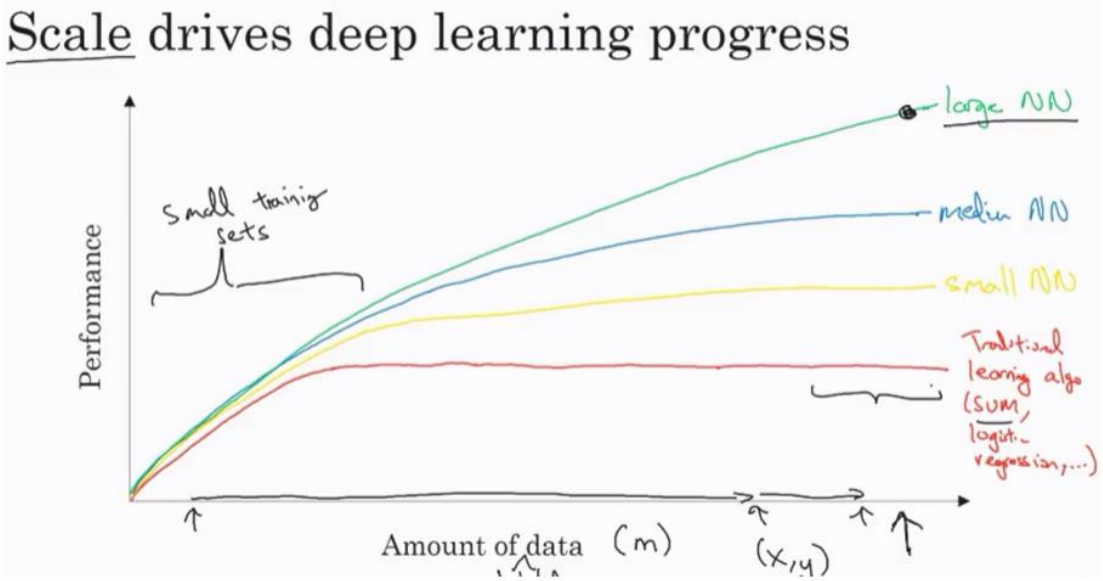
Supervised learning with neural networks

- Different types of neural networks for supervised learning which includes:
    - CNN or convolutional neural networks (Useful in computer vision)
    - RNN or Recurrent neural networks (Useful in Speech recognition or NLP)
    - Standard NN (Useful for Structured data)
    - Hybrid/custom NN or a Collection of NNs types
- Structured data is like the databases and tables.
- Unstructured data is like images, video, audio, and text.
- Structured data gives more money because companies relies on prediction on its big data.

Why is deep learning taking off?

- Deep learning is taking off for 3 reasons:

    i. Data:
        - Using this image we can conclude:



        - For small data NN can perform as Linear regression or SVM (Support vector machine)
        - For big data a small NN is better that SVM

- For big data a big NN is better that a medium NN is better that small NN.
- Hopefully we have a lot of data because the world is using the computer a little bit more
    - Mobiles
    - IOT (Internet of things)

ii. Computation:
- GPUs.
- Powerful CPUs.
- Distributed computing.
- ASICs

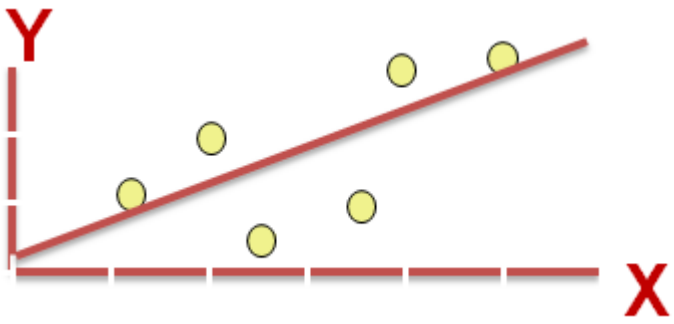
iii. Algorithm:

a. Creative algorithms has appeared that changed the way NN works.
    - For example using RELU function is so much better than using SIGMOID function in training a NN because it helps with the vanishing gradient problem.

Neural Networks Basics

Binary classification

- Mainly he is talking about how to do a logistic regression to make a binary classifier.



- **notations:**
  - `M is the number of training vectors`
  - `Nx is the size of the input vector`
  - `Ny is the size of the output vector`
  - `X(1) is the first input vector`
  - `Y(1) is the first output vector`
  - `X = [x(1) x(2).. x(M)]`
  - `Y = (y(1) y(2).. y(M))`

Logistic regression

- Algorithm is used for classification algorithm of 2 classes.
- Equations:
  - Simple equation: `y = wx + b`
  - If x is a vector: `y = w(transpose)x + b`
  - If we need y to be in between 0 and 1 (probability): `y = sigmoid(w(transpose)x + b)`

Logistic regression cost function

Loss function:

- **First loss function**: would be the square root error: `L(y',y) = 1/2 (y' - y)^2`
    - **problem** , **non convex**, means it contains **local optimum points**.
- **second loss function**: `L(y',y) = - (y*log(y') + (1-y)*log(1-y'))`
    - if `y = 1 ==> L(y',1) = -log(y') ==>` we want `y'` to be the largest `==> y'` biggest value is 1
    - if `y = 0 ==> L(y',0) = -log(1-y') ==>` we want `1-y'` to be the largest `==> y'` to be smaller as possible because it can only has 1 value.

Cost function:

```
J(w,b) = (1/m) * Sum(L(y'[i],y[i]))
```

- The loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set.

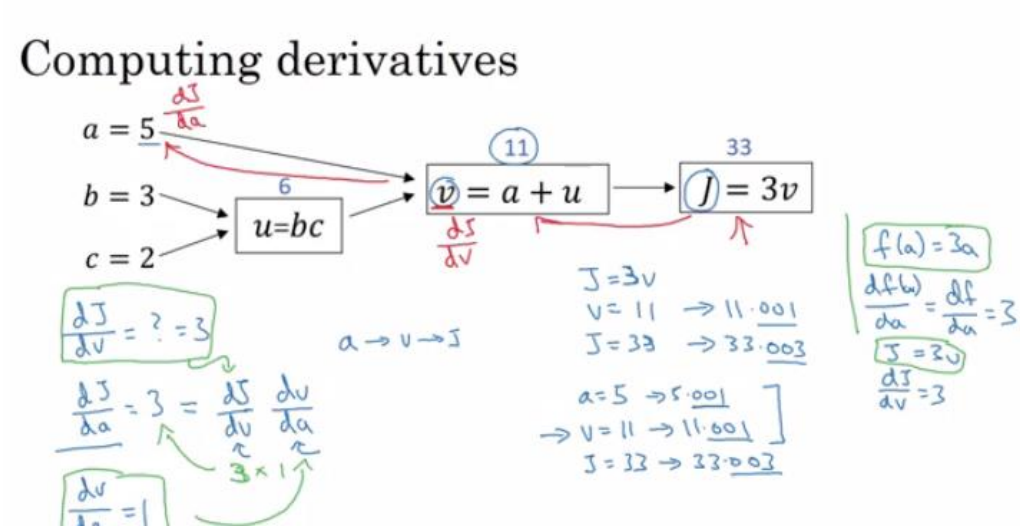
Gradient Descent

- `w` and `b` - initialize them to a random.
- The gradient decent algorithm repeats: `w = w - alpha * dw` where alpha is the learning rate and `dw` is the derivative of `w` (Change to `w`) The derivative is also the slope of `w`

- Equations:

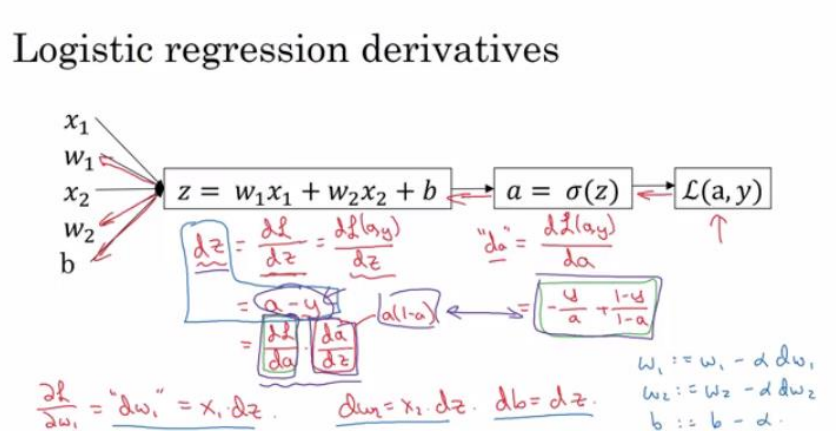
    - `w = w - alpha * d(J(w,b) / dw)` (how much the function slopes in the w direction)
    - `b = b - alpha * d(J(w,b) / db)` (how much the function slopes in the d direction)

Computation graph



- We compute the derivatives on a graph from right to left and it will be a lot more easier.

Logistic Regression Gradient Descent- Computation graph



variables:

- X1 Feature
- X2 Feature
- W1 Weight of the first feature.
- W2 Weight of the second feature.
- B Logistic Regression parameter.
- M Number of training examples
- Y(i) Expected output of i

```
X1  \
W1   \
X2   ===>  z(i) = X1W1 + X2W2 + B ===> a(i) = Sigmoid(z(i)) ===> l(a(i),Y(i)) = - (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))
Y2   /
B   /
```

$$Z = X * W + B$$
$$sigmoid: \hat{Y} = a = \frac{1}{1+e^{-z}} \rightarrow e^{-z} = \frac{1-a}{a}$$
$$L = -Ylog(a) + (1-Y)\log(1-a)$$
$$(1)\frac{dL}{da} = -\frac{Y}{a} + \frac{1-Y}{1-a}$$
$$(2)\ \frac{da}{dZ} = (e^{-z})\frac{1}{(1+e^{-z})^2} = \frac{1-a}{a} * a^2 = (1-a) * a$$
$$(3)\frac{dZ}{dW} = X, \frac{dZ}{db} = 1$$

Final all:

$$dz(sigmoid) = \frac{dL}{dz} = \frac{dL}{da}\frac{da}{dz} == \left(-\frac{Y}{a} + \frac{1-Y}{1-a}\right)(1-a) * a = a - Y$$

$$dz\left(\tanh = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}\right) = 1 - a^2$$

$$dz(RELU = \max(0, Z)) = \begin{cases} 0, z < 0 \\ 1, z > 0 \end{cases}$$
$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dw} = dz * X$$
$$\frac{dL}{db} = \frac{dL}{dz}\frac{dz}{db} = dz$$

logistic regression pseudo code-no vectorizing:

```
     J = 0; dw1 = 0; dw2 =0; db = 0;                 # Devs.
     w1 = 0; w2 = 0;
b=0;                                                   # Weights
     for i = 1 to m
              # Forward pass
              z(i) = W1*x1(i) + W2*x2(i) + b
              a(i) = Sigmoid(z(i))
              J += (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))

              # Backward pass
              dz(i) = a(i) - Y(i)
              dw1 += dz(i) * x1(i)
              dw2 += dz(i) * x2(i)
              db  += dz(i)
     J /= m
     dw1/= m
     dw2/= m
     db/= m
```

```
    # Gradient descent
    w1 = w1 - alpa * dw1
    w2 = w2 - alpa * dw2
    b = b - alpa * db
```

Vectorization is so important on deep learning to reduce loops. In the last code we can make the whole loop in one step using vectorization!

Vectorization

- we need vectorization to get rid of some of our for loops.
- NumPy library (dot) function is using vectorization by default.
- The vectorization can be done on CPU or GPU thought the SIMD operation. But its faster on GPU.
- Whenever possible avoid for loops.
- Most of the NumPy library methods are vectorized version.

Vectorizing Logistic Regression

- We will implement Logistic Regression using one for loop then without any for loop.

  input : `X [Nx, m]`
  output `Y [Ny, m]`

```
Z = np.dot(W.T,X) + b    # (1, m)
A = 1 / 1 + np.exp(-Z)   # (1, m)
```

Notes on Python and NumPy

- In NumPy, `obj.sum(axis = 0)` sums the columns while `obj.sum(axis = 1)` sums the rows.
- In NumPy, `obj.reshape(1,4)` changes the shape of the matrix by broadcasting the values.

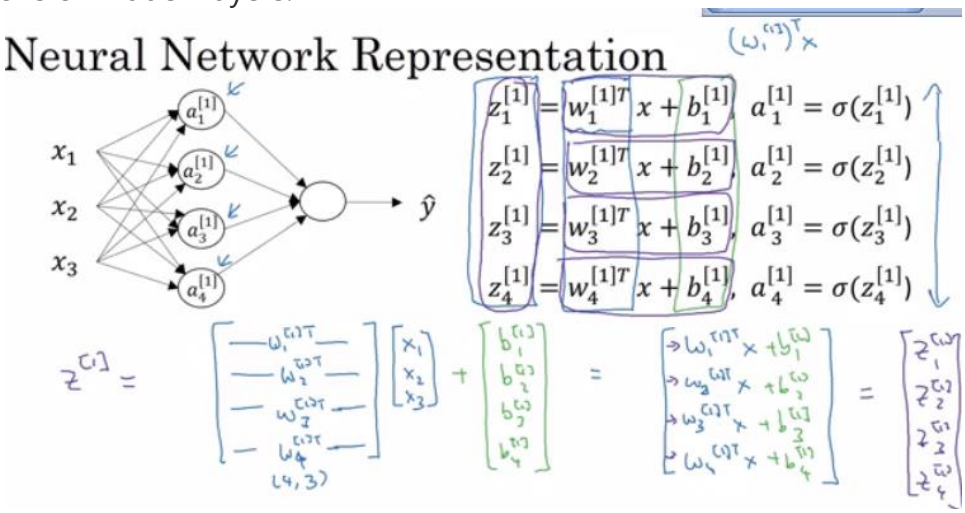
- Reshape is cheap in calculations so put it everywhere you're not sure about the calculations.

- Broadcasting works when you do a matrix operation with matrices that doesn't match for the operation, in this case NumPy automatically makes the shapes ready for the operation by broadcasting the values.

- In general principle of broadcasting. If you have an (m,n) matrix and you add(+) or subtract(-) or multiply(*) or divide(/) with a (1,n) matrix, then this will copy it m times into an (m,n) matrix. The same with if you use those operations with a (m , 1) matrix, then this will copy it n times into (m, n) matrix. And then apply the addition, subtraction, and multiplication of division element wise.
- Some tricks to eliminate all the strange bugs in the code:
    - If you didn't specify the shape of a vector, it will take a shape of `(m,)` and the transpose operation won't work. You have to reshape it to `(m, 1)`
    - Try to not use the rank one matrix in ANN
- Jupyter / IPython notebooks are so useful library in python that makes it easy to integrate code and document at the same time. It runs in the browser and doesn't need an IDE to run.

Neural Networks Overview

Computing a Neural Network's Output

- Equations of Hidden layers:
    - 

- Here are some informations about the last image:
    - `noOfHiddenNeurons = 4`
    - `Nx = 3`
    - Shapes of the variables:

- `W1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,nx)`
- `b1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,1)`
- `z1` is the result of the equation `z1 = W1*X + b`, it has a shape of `(noOfHiddenNeurons,1)`
- `a1` is the result of the equation `a1 = sigmoid(z1)`, it has a shape of `(noOfHiddenNeurons,1)`
- `W2` is the matrix of the second hidden layer, it has a shape of `(1,noOfHiddenNeurons)`
- `b2` is the matrix of the second hidden layer, it has a shape of `(1,1)`
- `z2` is the result of the equation `z2 = W2*a1 + b`, it has a shape of `(1,1)`
- `a2` is the result of the equation `a2 = sigmoid(z2)`, it has a shape of `(1,1)`

Pseudo code for forward propagation and backward propogation for the 2 layers NN:

Lets say we have x on shape `(Nx,m)`.

```
    Z1 = W1X + b1     # shape of Z1 (noOfHiddenNeurons,m)
    A1 = sigmoid(Z1)  # shape of A1 (noOfHiddenNeurons,m)
    Z2 = W2A1 + b2    # shape of Z2 is (1,m)
    A2 = sigmoid(Z2)  # shape of A2 is (1,m)

Compute predictions (y'[i], i = 0,...m)
       Get derivatives: dW1, db1, dW2, db2
       Update: W1 = W1 - LearningRate * dW1
                    b1 = b1 - LearningRate * db1
                    W2 = W2 - LearningRate * dW2
                    b2 = b2 - LearningRate * db2

dZ2 = A2 - Y      # derivative of cost function we used * derivative of the
sigmoid function
dW2 = (dZ2 * A1.T) / m
db2 = Sum(dZ2) / m
dZ1 = (W2.T * dZ2) * g'1(Z1)  # element wise product (*)
dW1 = (dZ1 * A0.T) / m   # A0 = X
db1 = Sum(dZ1) / m
```

Activation functions

- Sigmoid can lead us to gradient decent problem where the updates are so low.
- Tanh activation function range is [-1,1] (Shifted version of sigmoid function)
- It turns out that the tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.
- Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the gradient decent problem.
- **basic rule for choosing activation functions**, if your classification is between 0 and 1, use the output activation as sigmoid and the others as RELU.

Why do you need non-linear activation functions?

- Linear activation function will output linear activations
    - Whatever hidden layers you add, the activation will be always linear like logistic regression (So its useless in a lot of complex problems)

Random Initialization

- If we initialize all the weights with zeros in NN it won't work (initializing bias with zero is OK):
    - all hidden units will be completely identical (symmetric) - compute exactly the same function
    - on each gradient descent iteration all the hidden units will always update the same
- To solve this we initialize the W's with a small random numbers:

```
W1 = np.random.randn((2,2)) * 0.01    # 0.01 to make it small enough
b1 = np.zeros((2,1))                  # its ok to have b as zero, it won't
get us to the symmetry breaking problem
```

- small values - sigmoid (or tanh), for example, if the weight is too large you are in the saturated part of the functions, the derivatives are small thus slowing down learning. RELU this is less of an issue.

- Constant 0.01 is alright for 1 hidden layer networks, but if the NN is deep this number can be changed but it will always be a small number.

Deep Neural Networks

Deep L-layer neural network

Notations:

- Shallow NN is a NN with one or two layers.
- Deep NN is a NN with three or more layers.
- We will use the notation `L` to denote the number of layers in a NN.
- `n[l]` is the number of neurons in a specific layer `l`.
- `n[0]` denotes the number of neurons input layer. `n[L]` denotes the number of neurons in output layer.
- `g[l]` is the activation function.
- `a[l] = g[l](z[l])`
- `w[l]` weights is used for `z[l]`
- `x = a[0]`, `a[l] = y'`

Forward Propagation in a Deep Network

Forward propagation general rule for `m` inputs:

- `Z[l] = W[l]A[l-1] + B[l]`
- `A[l] = g[l](A[l])`

matrix dimensions right

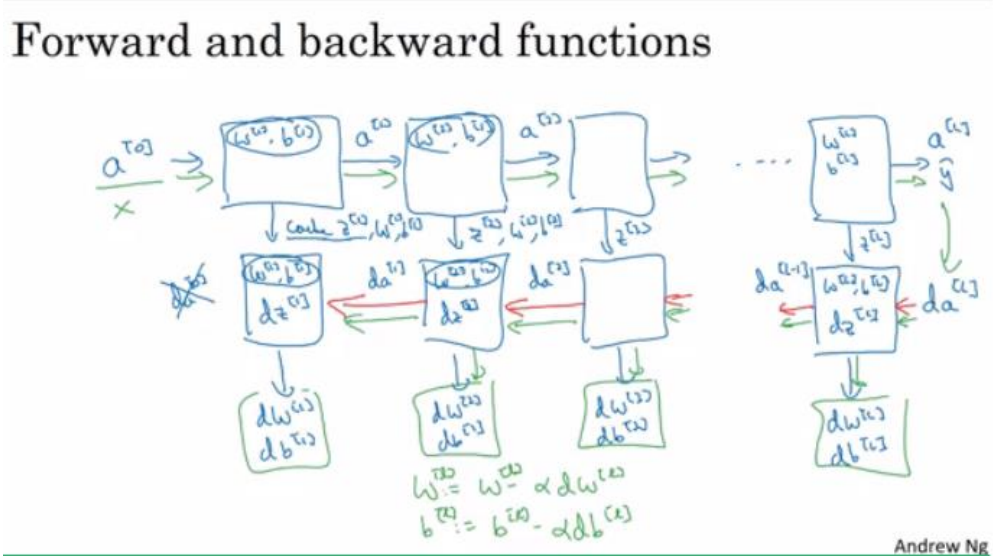
- `W,dW = (n[l],n[l-1])` .
- `b,db =(n[l],1)`
- `Z[l], A[l], dZ[l], dA[l] =(n[l],m)`

Building blocks of deep neural networks

- Forward and back propagation for a layer l:

```
                 ----------
                |  Forward |
A[l-1] ==>      |   Using: | ==>A[l]                    Equations: Z[l] = W[l]A[l-1] + b[l]    then A[l] = g[l](Z[l])
                |    W[l]  |
                |    b[l]  |
                 ----------
                     |
                     |  cache Z[l]
                     |  cache A[l-1]
                 ----------
                | Backward |
dA[l-1]<==      |   Using: |   <== dA[l]                Equations: dZ[l-1] = (W[l].T * dZ[l]) * g'[l](Z[l-1])
                | W[l],b[l]|
                |    Z[l]  |
                 ----------
                     |
                     | dW[l]                           Equations: dW[l] =  (dZ[l] * A[l-1].T) / m
                     | db[l]                           Equations: db[l] =  sum(dZ[l]) / m  # Sum over rows
```

- Deep NN blocks:



Forward and Backward Propagation

- Pseudo code for forward propagation for layer l:

```
Input  A[l-1]
Z[l] = W[l]A[l-1] + b[l]
A[l] = g[l](Z[l])
Output A[l], cache(Z[l])
```

- Pseudo code for back propagation for layer l:

```
Input da[l], Caches
dZ[l] = dA[l] * g'[l](Z[l])
dW[l] = (dZ[l]A[l-1].T) / m
db[l] = sum(dZ[l])/m                # Dont forget axis=1, keepdims=True
```

- `dA[l-1] = w[l].T * dZ[l]  # The multiplication here are a dot product.`
- `Output dA[l-1], dW[l], db[l]`

- If we have used our loss function then:

- `dA[L] = (-(y/a) + ((1-y)/(1-a)))`

Parameters vs Hyperparameters

- Main parameters of the NN is `W` and `b`
- Hyper parameters (parameters that control the algorithm) are like:
  - Learning rate.
  - Number of iteration.
  - Number of hidden layers `L`.
  - Number of hidden units `n`.
  - Choice of activation functions.
- You have to try values yourself of hyper parameters.
- In the earlier days of DL and ML learning rate was often called a parameter, but it really is (and now everybody call it) a hyperparameter.
- On the next course we will see how to optimize hyperparameters.