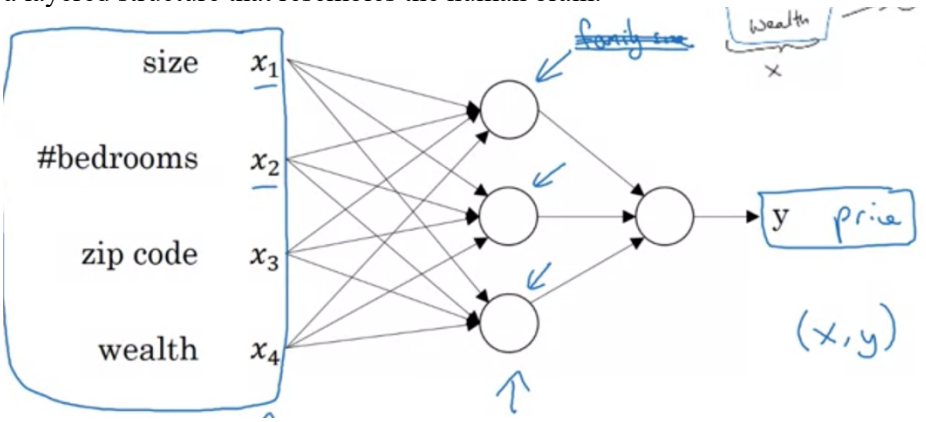
# Neural Networks and Deep Learning

- Introduction to Deep Learning

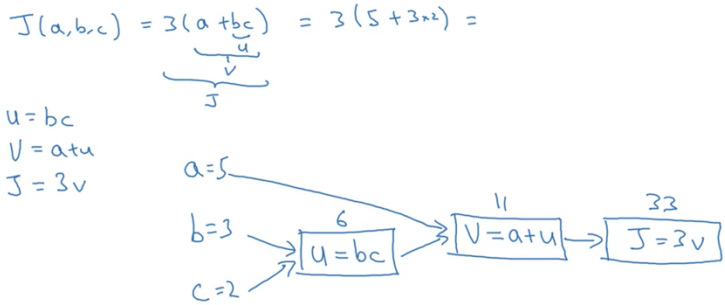
Deep learning is a type of machine learning, which is a subset of artificial intelligence.
Artificial Intelligence is the concept of creating smart intelligent machines. Machine Learning is a subset of artificial intelligence that helps you build AI-driven applications. Deep Learning is a subset of machine learning that uses vast volumes of data and complex algorithms to train a model.

**What is Neural Network?**
It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain.



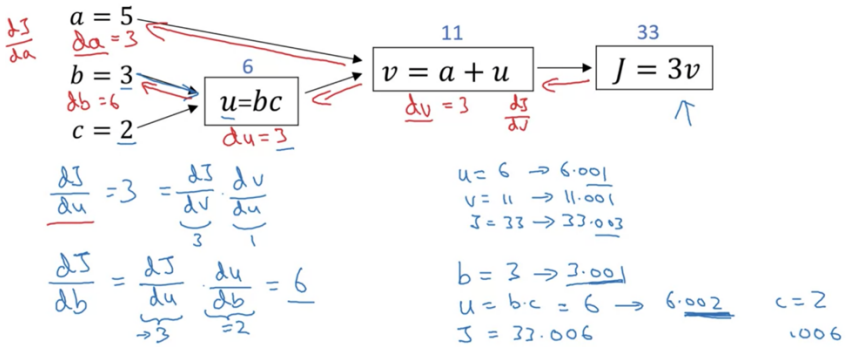

Here, we make sure the nodes in between will take in all x parameters and they will decide what will the nodes mean, when we compute the loss, we take the average of all nodes computed loss

## Computation Graph

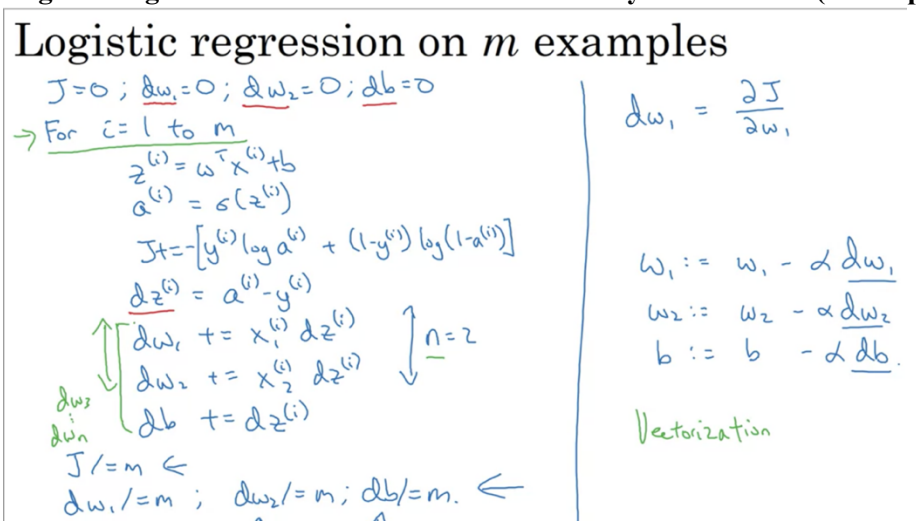



For derivative, it's backwards
Du/db = when b changes with value a, how much times of m changes in u

## Computing derivatives




**Logistic Regression with Gradient Descent for only one iteration (for loop)**




**What is Vectorization?**

Vectorization is a way in machine learning that we use in-built library / function(eg. numpy) for implementation to make it run faster than for loop

by using a vectorized implementation in an optimization algorithm we can make the process of computation much faster compared to Unvectorized Implementation

## Broadcasting example

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} 100 = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}_{(m,n)\ (2,3)} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix}_{(1,n)\rightsquigarrow(m,n)\ (2,3)} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$$

**Outer product vs Inner/dot product**

Outer product produces matrix

Given two vectors of size $m \times 1$ and $n \times 1$ respectively

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

their outer product, denoted $\mathbf{u} \otimes \mathbf{v}$, is defined as the $m \times n$ matrix $\mathbf{A}$ obtained by multiplying each element of $\mathbf{u}$ by each element of $\mathbf{v}$:[1]

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{A} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \dots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \dots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \dots & u_m v_n \end{bmatrix}$$

```
for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i] * x2[j] -> np.outer
```

Inner product produces scaler

```
for i in range(len(x1)):
    dot += x1[i] * x2[i]-> np.dot
```

Element wise product:

```
for i in range(len(x1)):
    mul[i] = x1[i] * x2[i] -> np.multiply
```

- Shallow Neural Network

**Neural Network Representation**

Example of 2-layer NN

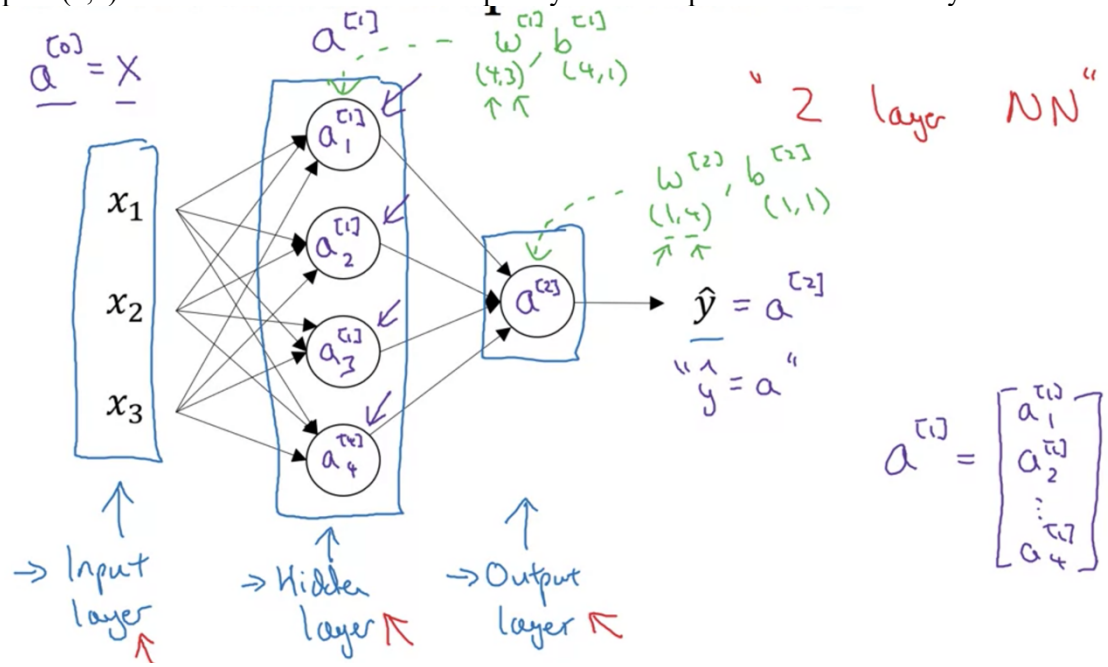
Input layer is layer 0, hidden layer is layer 1 etc.

a represent some activation function

different layer will have different parameters w and b

eg, w[1].shape = (4,3) because we have 4 nodes in hidden layers and 3 input features

w[2].shape = (1,4) because we have 1 node in output layer and 4 input nodes in hidden layer




It is similar as Logistic Regression, with multiple times repeat

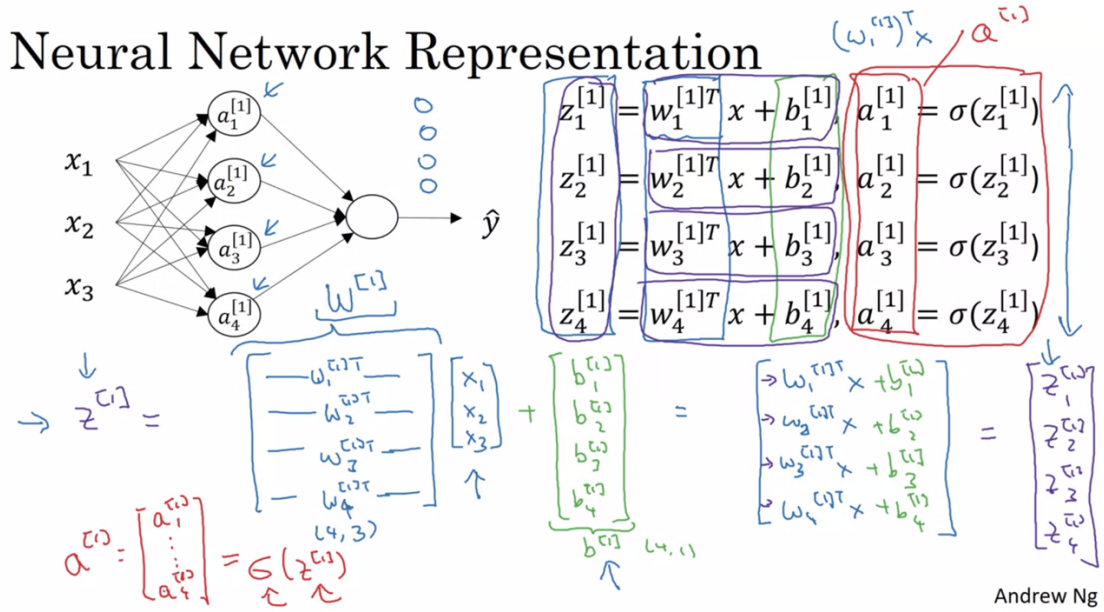
The superscript [] means the nth layer , such as a_[1] means activate function for layer 1

The subscript means nth node/unit, such as $a^{[1]}_2$ means second node for first layer

The superscript () means nth training example, such as $a^{[1](2)}$ means second training example's activated result in layer 1

With more details described…

1. Compute z = W.T * X+b for each node for layer1, then compute a = sigmoid(z) for layer1
2. Since hidden layer typically has multiple nodes/units, to make it more efficient, we need to vectorize.
    a. Which means, stack each parameter vertically. Ie, stacking w for each node for layer 1 into one matrix etc.




If we continue to the following layers with m training examples…
For i to m:

Given input x:

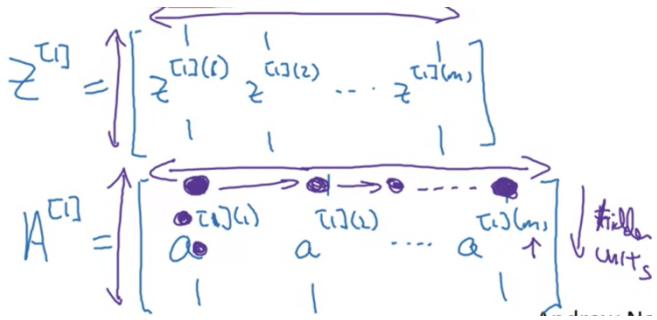
$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$ (4,1) (4,3) (3,1) (4,1)

$a^{[1]} = \sigma(z^{[1]})$ (4,1) (4,1)

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ (1,1) (1,4) (4,1) (1,1)

$a^{[2]} = \sigma(z^{[2]})$ (1,1) (1,1)

But we want to get rid of the loop, ie, vectorize it
Similar as how we vectorize w and b, we now need to stacking z and a
$Z^{[1]}$ now means the first layer results containing all training examples, but $z^{[1](2)}$ means the first layer second training example
Similarly, $a^{[1](2)}$ means activation result for first layer second example , $A^{[1]}$ means activation result for first layer containing all training examples,



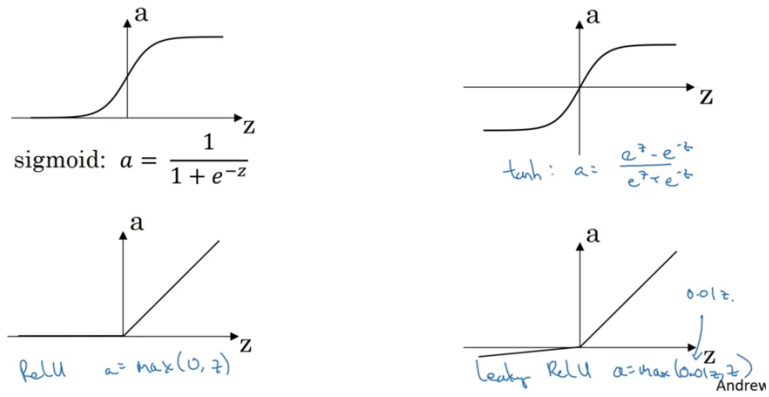
For both Z and A, column means nth training example, row means each hidden unit in the layer, for this example, in layer 1, because we have $Z^{[1]}$
To understand this, we have training example as superscript, that generally means col, but we have unit as subscript, that generally means rows.

**Activation Function**

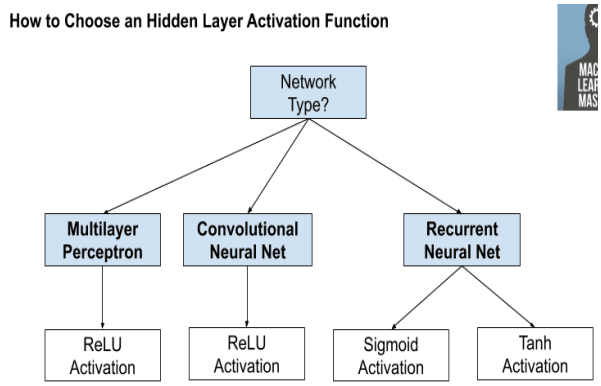
***For hidden layer***

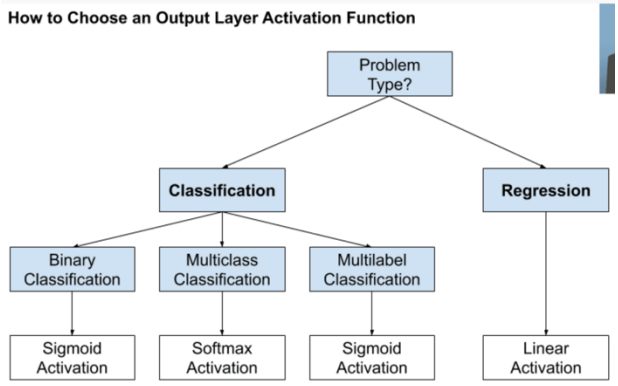
Sigmoid function is just an example of activation function used in logistic regression, we have much more choices in NN
One reason we use sigmoid function in logistic regression because it limits the results to [0,1] , because a = 1/(1+e^-z)

# Pros and cons of activation functions




Sigmoid: only when you need output to be binary
Tanh: works better for sigmoid, output is [-1,1], The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.the tanh activation is not always better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data, making learning complex for the next layer.
The cons for both sigmoid and Tanh, it will converge very slowly once input z is really large because the slope will be really small
Relu: solve the large z problem
General rules of thumb:



**Derivative of activation functions**
Let a = activation function
For sigmoid function: a' = 1/(1-a)
For tanh function : a' = $1-a^2$
For ReLu function : a' = 0 if z<0
a' = 1 if z >0 otherwise undefined

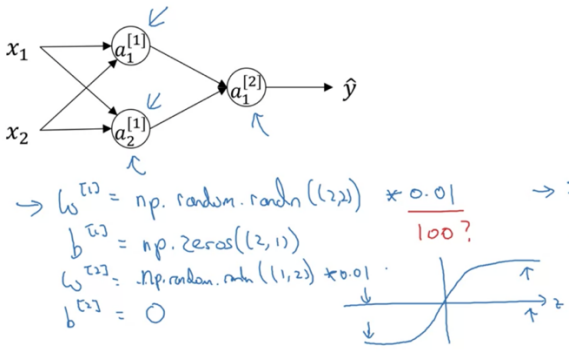
**Derivative of Neural Network with Vectorization**

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

This is an example of two layers NN, and this is calculated by back propagation

**Why we use non linear activation function?**
The purpose of the activation function is to introduce nonlinearity to the network
In turn, this allows you to model a response variable that varies non-linearity with its explanatory variables (cannot be represented by a straight line)
Another way to think about it: without a non-linear activation function in the network, no matter how many layers we have, all we did is just like a single-layer perceptron, because summing these layers would give you just another linear function
A linear activation function can be used for linear regression

**Random Initialization**
We want to initialize w randomly instead of 0 as we did in logistic regression, because 0 will make the nodes in the hidden layer symmetric




0.01 because we want to make sure our initialized w to be small, large w can result in large z, so our slope of activation function can be large->more flat, so that the gradient descent will converge slowly

**General Methodology/ step to build a NN :**

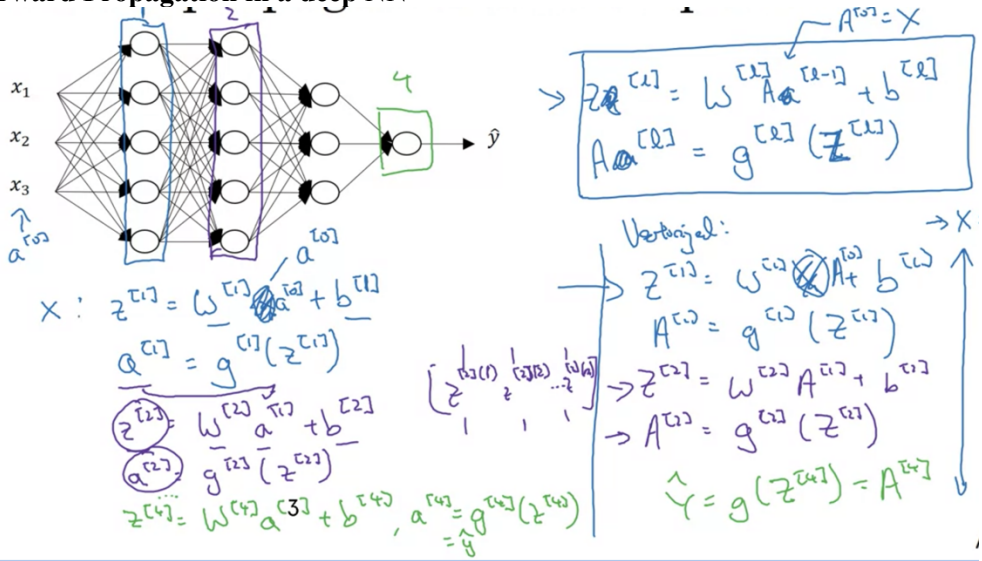
1. Define the neural network structure (# of input, # of hidden units, etc)
2. Initialize the model's parameter
3. Loop:
    a. Implement forward propagation ->compute activation for each layer and estimate y_hat
    b. Compute loss
    c. Implement backward propagation to get the gradients
    d. Update parameters

In practice, we should build helper functions to compute 1-3 then merge them into one function called nn_model()

- Deep Neural Network

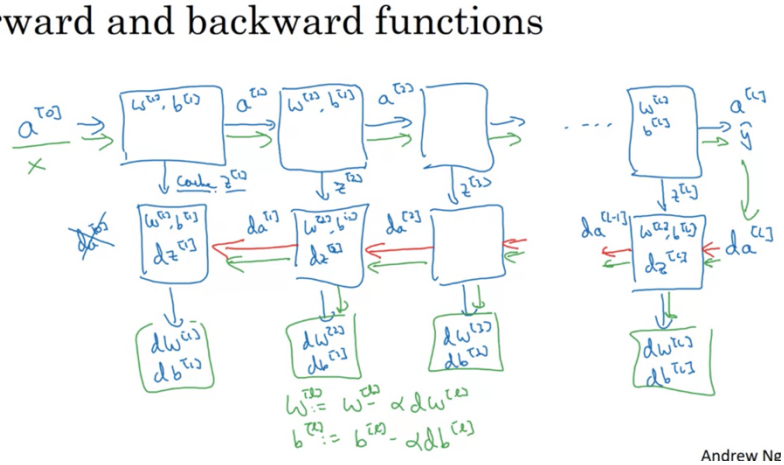
**What is deep NN?**
The difference between deep NN and shallow NN mainly is how many hidden layer does the network has
**Some notations:**
L = # of layers (including hidden layer and output layer)
$N^{[L]}$ = # of units in layer L
$a^{[L]} = g^{[L]}(Z^{[L]})$ = activation in layer L
mostly the same as shallow NN

**Forward Propagation in a deep NN**



The final result we want is the Z and A to use in gradient descent and Y_hat for prediction
There is a for loop for looping through the layers, and there is no good way to get rid of it
**How to get the matrix dimensions right?**
$W^{[L]}$ .shape = $(n^{[L]}, n^{[L-1]})$
$Z^{[L]}$.shape = $(n^{[L]},m) = a^{[L]}$.shape
$b^{[L]}$.shape = $(n^{[L]},m)$
*their derivatives have the same dimensions as themselves
*n denotes the number of units in layer L, m denotes the number of training examples

**General Steps of building NN:**



**Implementation**

1. Initialize parameters
2. For loop – Gradient Descent
    a. Forward Propagation

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

until Z_L

    b. Compute cost

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

an example of Logistic regression

    c. Compute grads – Back Propagation

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

    d. Update parameters

$\theta = \theta - \alpha\frac{\partial J}{\partial \theta}$ where $\alpha$ is the learning rate and $\theta$ represents a parameter.

3. Using forward propagation to predict

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

**Hyperparameters and Parameters**

Parameters: W,b

Hyperparameters: learning rate, #iterations, #hidden layers, #hidden units, choice of activation function – things you need to tell your learning algorithm, that can optimize your parameters

# Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

- Week1. Practical aspects of DL

**Setting up your ML application**

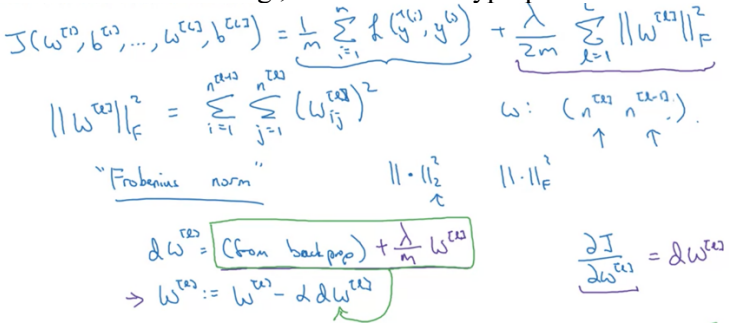
In traditional ML, when we split train/dev/test, we normally follow the percentages: 60/20/20, that's good enough for normal sized data, such as 1k,10k, but when we talk about deep learning, sometimes the data becomes too large, such as 1m, then we don't need so much data in dev/test
For example, if we have 1M data, our dev size should be big enough to compare two algorithms, such as 1k, or 10k, our test size should be big enough to test if our learning algorithm works, such as 1k, 10k
Try to make sure Dev/test sets come from the same distribution
If you want to build a reliable machine learning model, you need to split your dataset into the training, validation, and test sets. If you don't, **your results will be biased, and you'll end up with a false impression of better model accuracy**.

- **Bias and Variance**

Train error ≈dev error, but they are both high -> high bias -> underfitting ->model too simple
Train error < dev error, and train error is small -> high variance -> overfitting -> model too complex
Train error < dev error, but they are both high -> high variance and bias

- **Apply L2 Regularization in Neural Network**

To deal with overfitting , lambda is the hyperparameter we need to tune




Why does Regularization works in NN?
When our lambda big enough, we are forcing w to be small for some layers, so we think of this as we make our NN close to LR

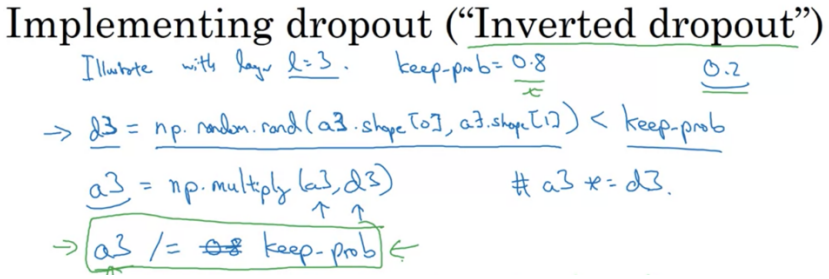
**What is L2-regularization actually doing?**:

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

**Dropout- Regularization**

It means we dropped some nodes in each layer randomly with specified probabilities, the dropped nodes are randomly chosen for each iteration in Gradient Descent . to make our NN model simpler.
For layer with a lot nodes, we can set the keep-prob low, because there is a high chance of overfitting. For layer with just a little nodes, we can set the keep-prob high, for example, for the output layer with only one node, we don't need to worry about the overfitting, so that we can set the keep-prob = 1

Implementation:




Keep-prob = 0.8 means each node has 80% to be kept, 20% to be dropped
D3 is the matrix with the same dimension as the activation matrix with random number, so if the number < 0.8, d3 = 1, otherwise =0
Multiply d3 with a3 makes a3 drop 20% nodes

**Other methods for overfitting:**

Data augmentation, early stopping(but may affect negatively on minimizing cost function)

**What is weight decay?**

A regularization technique (such as L2 regularization) that results in gradient descent shrinking the weights on every iteration

- **Setting up your Optimization Problem**

**Normalizing inputs**

$\frac{X-\mu}{\sigma}$ use the same mean and standard deviation for test and train sets

Why do we want to do normalization?

When features distributions different a lot, Normalization can help gradient descent converge sooner

**Weight Initialization**

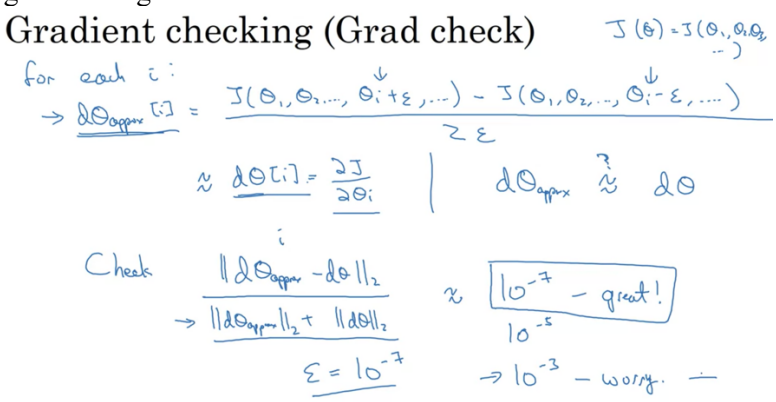
Why do we want w to be close to 1? -> because when w is a little larger or smaller than x , with a DNN, y will explode or vanish

Xaviers initialization : $w^l = np.random.randn(n\_h,n\_x) * \sqrt{\frac{1}{dim\ of\ the\ previous\ layer\ (layers\_dim\ [l-1]}}$

He initialization : $w^l = np.random.randn(n\_h,n\_x) * \sqrt{\frac{2}{dim\ of\ the\ previous\ layer\ (layers\_dim\ [l-1]}}$ *(for Relu)*

**Numerical Approximation of Gradients**

Sometimes backward Propagation is easily to get wrong, with grad check we can check if we got all the gradients right




θ = [w1,b1,…wl,bl]

## implementation

### Exercise 3 - gradient_check

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

**Instructions**:

- First compute "gradapprox" using the formula above (1) and a small value of $\varepsilon$. Here are the Steps to follow:
  1. $\theta^{+} = \theta + \varepsilon$
  2. $\theta^{-} = \theta - \varepsilon$
  3. $J^{+} = J(\theta^{+})$
  4. $J^{-} = J(\theta^{-})$
  5. $gradapprox = \frac{J^{+} - J^{-}}{2\varepsilon}$
- Then compute the gradient using backward propagation, and store the result in a variable "grad"
- Finally, compute the relative difference between "gradapprox" and the "grad" using the following formula:

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

  You will need 3 Steps to compute this formula:
  - 1'. compute the numerator using np.linalg.norm(...)
  - 2'. compute the denominator. You will need to call np.linalg.norm(...) twice.
  - 3'. divide them.
- If this difference is small (say less than $10^{-7}$), you can be quite confident that you have computed your gradient correctly. Otherwise, there may be a mistake in the gradient computation.

## What to remember :

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

- Week2: Optimized Algorithms

**Mini-Batch Gradient Descent**

Batch GD: We take the average of the gradients of **all the training examples** and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch

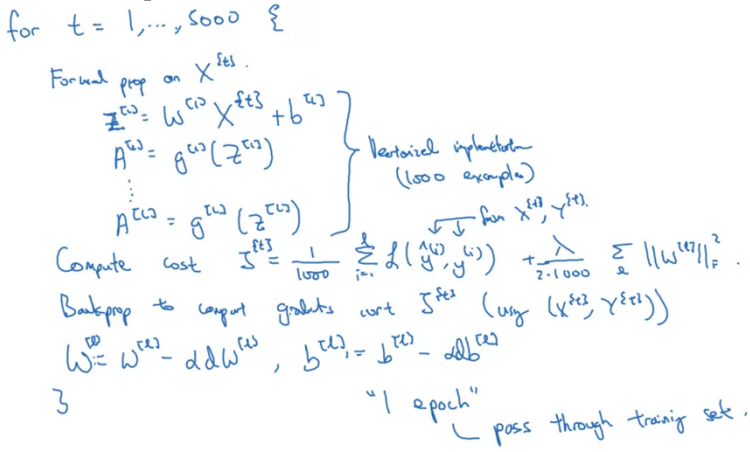
Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.

However, it is not working on large dataset

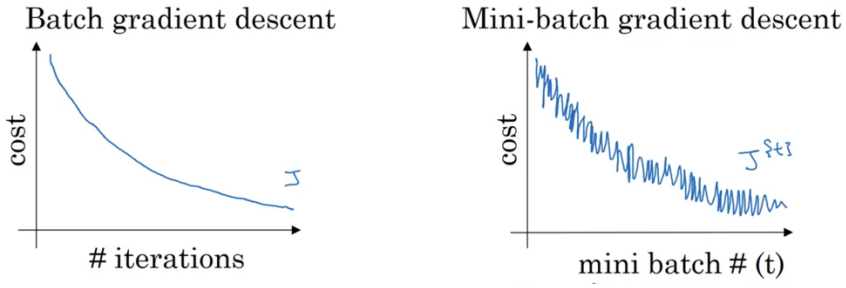
Mini-Batch Gradient Descent: Basically, we divided our training examples into t batches, and run GD

1. Pick a mini bacth
2. Feed it to NN
3. Calculate the mean gradient of the mini batch t ($X^{\{t\}}$)
4. Use the mean gradient to update weights
5. Repeat 1-4 for all the mini batches

*Details in step2:*




***Understanding mini-batch GD***




Hyperparameter: mini-batch size

If mini-batch size = m: batch gradient descent -> $(X^{\{1\}},Y^{\{1\}}) = (X,Y)$ -> cost too large for each iteration ->should use on small training set

If mini-batch size = 1: stochastic gradient descent, train on every sample -> may not converge, too many iterations, lose speed up from vectorization

Desired mini-batch size = between 1 and m, power of 2

Make sure mini batch fit in CPU / GPU memory

**Exponentially Weighted Moving Averages**

EWMA applies weights to the values of a time series. More weight is applied to more recent data points, making them more relevant for future forecasts. That's the most significant improvement over simple moving averages.

$$y_0 = x_0$$

$$y_t = \alpha x_t + (1 - \alpha) y_{t-1}$$

Image 3 — EWMA weight calculation when adjust=False (image by author)

- Increasing α will shift the line slightly to the right.
- Decreasing α will create more oscillation within the line.

**More Optimization algorithms working better than GD**

**GD with Momentum**

The problem with gradient descent is that the weight update at a moment (t) is governed by the learning rate and gradient at that moment only. It doesn't take into account the past steps taken while traversing the cost space

By adding a momentum term in the gradient descent-> updated parameters with weighted moving average, gradients accumulated from past iterations will push the cost further to move around a saddle point even when the current gradient is negligible or zero.

Implementation details

On iteration $t$:

Compute $dW, db$ on the current mini-batch

$v_{dW} = \beta v_{dW} + (1 - \beta) dW$

$v_{db} = \beta v_{db} + (1 - \beta) db$

$W = W - \alpha v_{dW}, \; b = b - \alpha v_{db}$

Hyperparameters: $\alpha, \beta$ $\beta = 0.9$

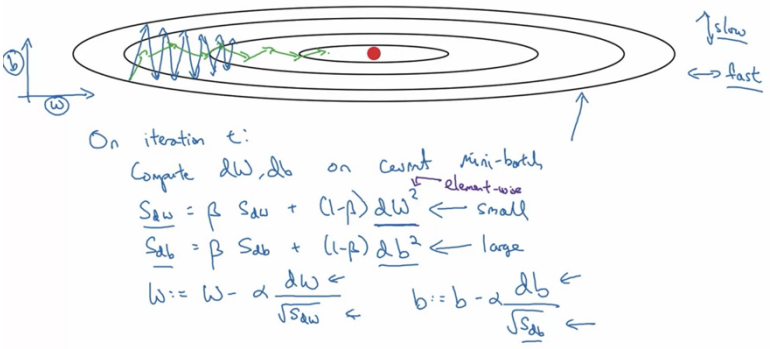
What will happen if you change value of β?

- Increasing β will make the line smoother. The larger the momentum $\beta$ is, the smoother the update, because it takes the past gradients into account more. But if $\beta$ is too big, it could also smooth out the updates too much.

- Decreasing β will create more oscillation within the line.

**RMSprop**

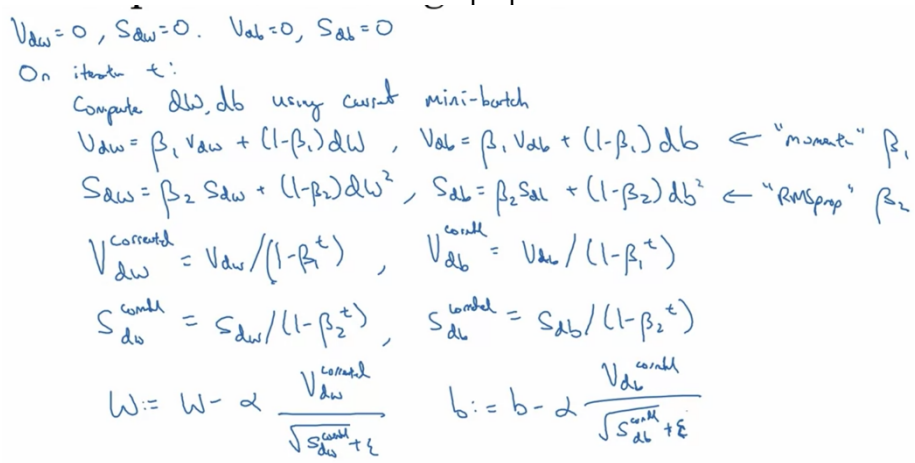
Simply put, RMSprop uses an **adaptive learning rate** instead of treating the learning rate as a hyperparameter. This means that the learning rate changes over time. using a moving average of squared gradients to normalize the gradient. This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding and increasing the step for small gradients to avoid vanishing.




The intuition is that we want our learning to move slower at b direction and faster at w direction. If we have a small dw, so sdw will be small so that w will be large, similarly, if db is large, b will be small.

**Adam optimization algorithm**

A combination of momentum and RMSprop




→ bias correction

Hyperparamters:
Alpha: learning rate, need to tune
Beta1 : 0.9
Beta2: 0.999
Epsilon: 10e-8

**Learning Rate Decay**

During the first part of training, your model can get away with taking large steps, but over time, using a fixed value for the learning rate alpha can cause your model to get stuck in a wide oscillation that never quite converges. But if you were to slowly reduce your learning rate alpha over time, you could then take smaller, slower steps that bring you closer to the minimum. This is the idea behind learning rate decay.

Learning rate decay can be achieved by using either adaptive methods or pre-defined learning rate schedules.

$$\alpha = \frac{1}{1+decayRate \times epochNumber}\alpha_0$$

Parameters to tune: alpha0, decay rate

When you're training for a few epoch this doesn't cause a lot of troubles, but when the number of epochs is large the optimization algorithm will stop updating. One common fix to this issue is to decay the learning rate every few steps. This is called fixed interval scheduling.

$$\alpha = \frac{1}{1 + decayRate \times \lfloor \frac{epochNum}{timeInterval} \rfloor}\alpha_0$$

**Saddle point problems (plateau)**

When you train a supervised machine learning model, your goal is to minimize the loss function - or error function - that you specify for your model.
Generally speaking, this goes pretty easily in the first iterations of your training process. Loss falls fast, and the model improves substantially. But then, you get stuck.

You encounter what is known as a loss plateau - suddenly, it seems to have become impossible to improve the model, with loss values balancing around some constant value.
It may be the case that you have reached the global loss minimum. In that case, you're precisely where you want to be.
But what if you're not? What if your model is stuck in what is known as a saddle point, or a local minimum?

- Week3: Hyperparameter Tuning, Batch Normalization and Programming Frameworks

**Coarse to Fine – ways to find the best combination of hyperparameters**

"Coarse to Fine" usually refers to the hyperparameter optimization of a neural network during which you would like to try out different combinations of the hyperparameters and evaluate the performance of the network.

However, due to the large number of parameters AND the big range of their values, it is almost impossible to check all the available combinations. For that reason, you usually discretize the available value range of each parameter into a "coarse" grid of values (i.e. val = 5,6,7,8,9) to estimate the effect of increasing or decreasing the value of that parameter. After selecting the value that seems most promising/meaningful (i.e. val = 6), you perform a "finer" search around it (i.e. val = 5.8, 5.9, 6.0, 6.1, 6.2) to optimize even further.

## Batch Normalization

An algorithm method which makes the training of DNN faster and stable by normalizing the activation vectors (Z) from hidden layers using mean and variance.

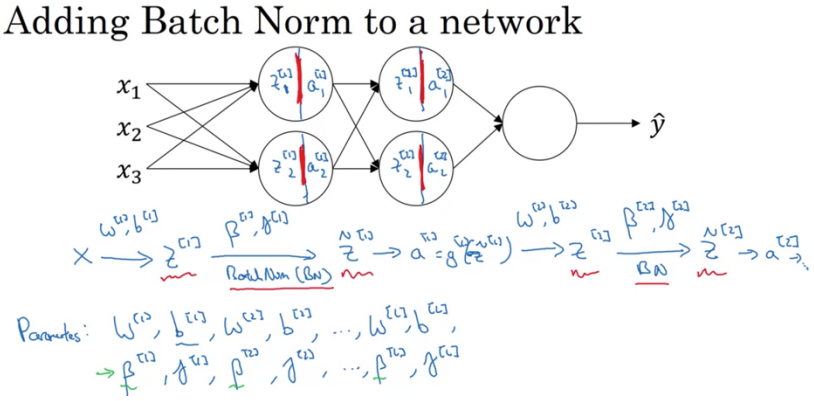
At each hidden layer, Batch Normalization transforms the signal as follow :

$$(1)\ \mu = \frac{1}{n}\sum_i Z^{(i)} \qquad (2)\ \sigma^2 = \frac{1}{n}\sum_i (Z^{(i)} - \mu)^2$$

$$(3)\ Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 - \epsilon}} \qquad (4)\ \check{Z} = \gamma * Z^{(i)}_{norm} + \beta$$

where we can tune γand β to control the value of Z. this is called linear transformation.
The output of (2) is under standard normal distribution, but for some activation function, we don't want our input to be st. normal, so we can use (4) to choose the optimum distribution

- γ allows us to adjust standard deviation
- β allows us to adjust bias, shifting the curve on the right or left side




when working with mini batches, working on $X^{\{i\}}$ for each normalization
βand γ has the same dimension as b

**Implementation with GD**

For t =1…,num_mini_batcges:

Compute Forward Prop on $X^{\{t\}}$

- In each hidden layer, use Batch normalization to replace $Z^{[l]}$ with $Z^{\sim[l]}$

Use back prop to compute $dw^{[l]}$, $db^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

We can ignore db because it will be eliminated during the normalization

Update parameters : w, β， γ

**Why BN work?**

The main difference with BN and normalization is that normalization works with your input X but BN works on hidden layers

It helps us to shift input distribution while learning

It helps each layer to learn independently

Each mini-batch is scaled computed on just that mini-batch, this adds some noise to the values Zl within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations, this has a slight regularization effect(unintended). With a larger batch size, you reduce the regularization effect

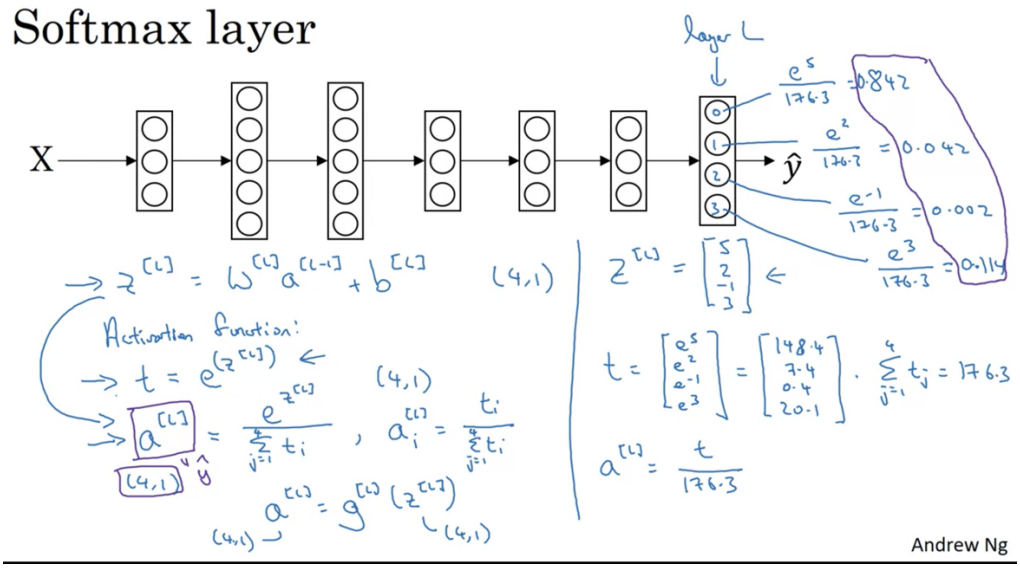
BN handles training set differently with **test set**. Unlike the training phase, **we may not have a full batch to feed into the model during the evaluation phase**.

How:

This trick is to define (***μ***_pop , **σ**_pop), which are respectively the estimated mean and standard deviation of the **targeted population**. Those parameters are calculated as the mean of all the (***μ***_batch, **σ**_batch) determined during training.

## Multi-class Classification – Softmax Regression

In traditional logistic regression, we are only able to predict binary classification, but now we want to predict multi classes.
C = # of classes (0,…C-1)
So that our output layer will have C units, each unit = P(class i|X), so sum of all units =1
Y_hat.shape = (C,1)



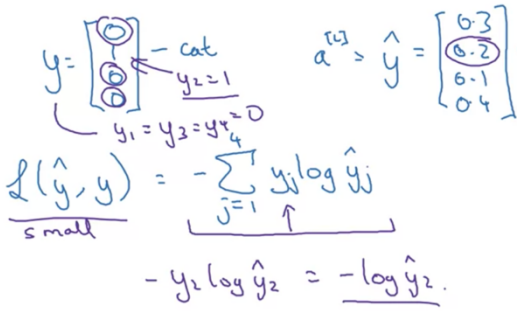

t is just a temporary variable contains the exponential function of Z

**Loss function**

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j.$$

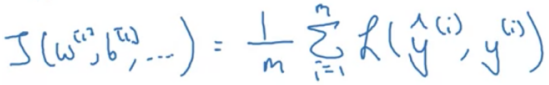
q is the number of classes (only used in this pic)

why this makes sense?



so, if we want to minimize loss function, we need y_hat to be large, that makes the loss function reasonable

Similarly, the cost function would be:



**Gradient Descent:**
$dZ^{[L]} = y\_hat - y$, $dZ^{[L]}.shape = (C,1)$