

## Module 1: Practical Aspects of Deep Learning

The below pointers summarize what we can expect from this module:

- Recalling that different types of initializations lead to different results
- Recognizing the importance of initialization in complex neural networks
- Recognizing the difference between train/validation/test sets
- Diagnosing the bias and variance issues in our model
- Learning when and how to use regularization methods such as dropout or L2 regularization
- Understanding experimental issues in deep learning, such as Vanishing or Exploding gradients and how to deal with them
- Using gradient checking to verify the correctness of our backpropagation implementation

This module is fairly comprehensive, and is thus further divided into three parts:

- Part I: Setting up your Machine Learning Application
- Part II: Regularizing your Neural Network
- Part III: Setting up your Optimization Problem

Let's walk through each part in detail.

### Part I: Setting up your Machine Learning Application

#### Train / Dev / Test sets

While training a deep neural network, we are required to make a lot of decisions regarding the following hyperparameters:

1. Number of hidden layers in the network
2. Number of hidden units for each hidden layer
3. Learning rate
4. Activation function for different layers, etc.

There is no specified or pre-defined way of choosing these hyperparameters. The below is what we generally follow:

1. Start with an **idea**, i.e. start with a certain number of hidden layers, certain learning rate, etc.
2. Try the idea by **coding** it
3. **Experiment** how well the idea has worked



#### 4. Refine the idea and iterate this process

Now how do we identify whether the idea is working? This is where the train / dev / test sets come into play. Suppose we have an entire dataset:

Data



We can divide this dataset into three different sets like:

Data



Training  
Set

Dev Set   Test Set

1. **Training Set:** We train the model on the training data.
2. **Dev Set:** After training the model, we check how well it performs on the dev set.
3. **Test Set:** When we have a final model (i.e., the model that has performed well on both training as well as dev set), we evaluate it on the test set in order to get an unbiased estimate of how well our algorithm is doing.

There is still one question left after this – **what should be the length of these training, dev and test sets?** It's actually a pretty critical aspect of any machine learning project, and will end up playing a big part in deciding how well the model performs. Let's look at some traditional guidelines that experts follow to decide the length of each set:

- In the previous era, when we had small datasets, the distribution of different sets was:



Training  
Set  
(60%)

Dev Set  
(20%)

Test Set  
(20%)

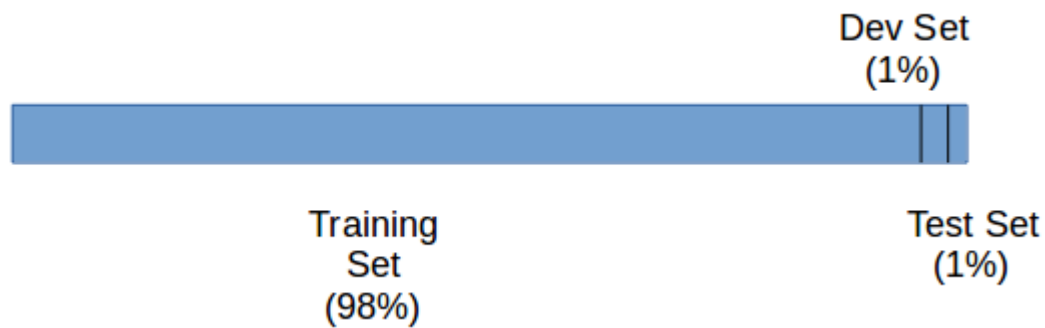
or just:



Training  
Set  
(70%)

Test Set  
(30%)

- As the availability of data has increased in recent years, we can use a huge slice of it for training the model:



This is certainly one way of deciding the length of these different sets. This works fine most of the time, but indulge me and consider the following scenario:

*Suppose we have scraped multiple images of cats from different sites, and also clicked a few images using our own camera. The distribution of both these types of images will be different, right? Now, we split the data in such a way that the training set contains all the scraped images, while the dev and test sets have all the camera images. In this case, the distribution of the training set will be different from the dev and test sets and hence, there's a good chance we might not get good results.*

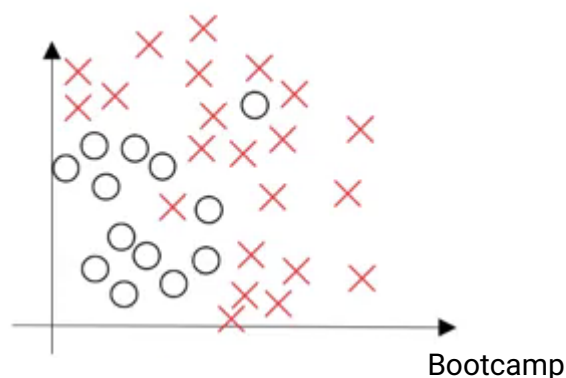
In cases like these (different distributions), we can follow the following guidelines:

1. Divide the training, dev and test sets in such a way that their distribution is similar
2. Skip the test set and validate the model using the dev set only

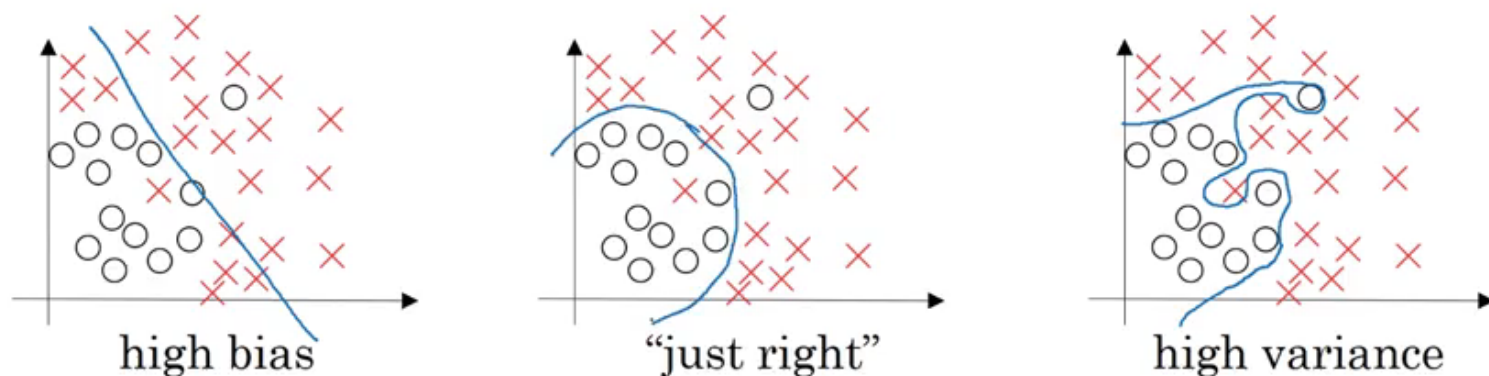
We can also use these sets to look at the bias and variance of the model, These help us decide how well the model is fitting and performing.

## Bias / Variance

Consider a dataset which gives us the below plot:



What will happen if we fit a straight line to classify the points into different classes? The model will under-fit and have a high bias. On the other hand, if we fit the data perfectly, i.e., all the points are classified into their respective class, we will have high variance (and overfitting). The right model fit is usually found between these two extremes:



We want our model to be just right, which means having low bias and low variance. We can decide if the model should have high bias or high variance by checking the train set and dev set error. Generally, we can define it as:

train set error (%)	1	15	15	0.5
dev set error (%)	11	16	30	1
result	High variance	High Bias	High Bias and High variance	Low bias and Low Variance

- If the dev set error is much more than the train set error, the model is overfitting and has a high variance
- When both train and dev set errors are high, the model is underfitting and has a high bias
- If the train set error is high and the dev set error is even worse, the model has both high bias and high variance
- And when both the train and dev set errors are small, the model fits the data reasonably and has low bias and low variance

## Basic Recipe for Machine Learning

I have a very simple method of dealing with certain problems I face in machine learning. Ask a set of questions and then figure out the answers one-by-one. It has proven to be extremely helpful for me in my journey and more often than not has led to improvements in the model's performance. These questions are listed below:

### Question 1: Does the model have high bias?

**Solution:** We can figure out whether the model has high bias by looking at the training set error. High training error results in high bias. In such cases, we can **try bigger networks, train models for a longer period of time, or try different neural network architectures.**

## Question 2: Does the model have high variance?

**Solution :** If the dev set error is high, we can say that the model has high variance. To reduce the variance, we can get more data, use regularization, or try different neural network architectures.

One of the most popular techniques to reduce variance is called regularization. Let's look at this concept and how it applies to neural networks in part II.

## Part II: Regularizing your Neural Network

We can reduce the variance by increasing the amount of data. But is that really a feasible option every time? Perhaps there is no other data available, and if there is, it might be too expensive for your project to source. This is quite a common problem. And that's why the concept of regularization plays an important role in preventing overfitting.

### Regularization

Let's take the example of logistic regression. We try to minimize the loss function:

$$\min_{w,b} J(w, b)$$
$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)})$$

Now, if we add regularization to this cost function, it will look like:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2$$
$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$



This is called **L2 regularization**.  $\lambda$  is the regularization parameter which we can tune while training the model. Now, let's see how to use regularization for a neural network. The cost function for a neural network can be written as:

$$J(w^{[0]}, b^{[0]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{n} \sum_{i=1}^n \ell(\hat{y}^{(i)}, y^{(i)})$$

We can add a regularization term to this cost function (just like we did in our logistic regression equation):

$$J(w^{[0]}, b^{[0]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{n} \sum_{i=1}^n \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|^2$$

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2$$

Finally, let's see how regularization works for a gradient descent algorithm:

- Update equation without regularization is given by:

$$dw^{[l]} = \text{from backpropagation}$$

$$w^{[l]} = w^{[l]} - \alpha dw^{[l]}$$

- Regularized form of these update equations will be:

$$dw^{[l]} = (\text{from backpropagation}) + (\lambda/m) w^{[l]}$$

$$w^{[l]} = w^{[l]} - \alpha [dw^{[l]} + (\lambda/m) w^{[l]}]$$

As you can surmise from the above equations, the reduction in weights will be more in case of regularization (since we are adding a higher quantity from the weights). This is the reason L2 regularization is also known as weight decay.

You must be wondering at this point **how in the world does regularization prevent overfitting in the model?** Let's try to understand it in the next section.

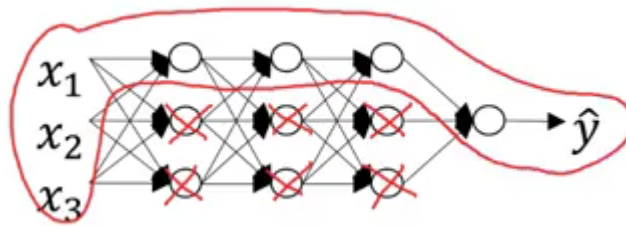
## How does regularization reduce overfitting?



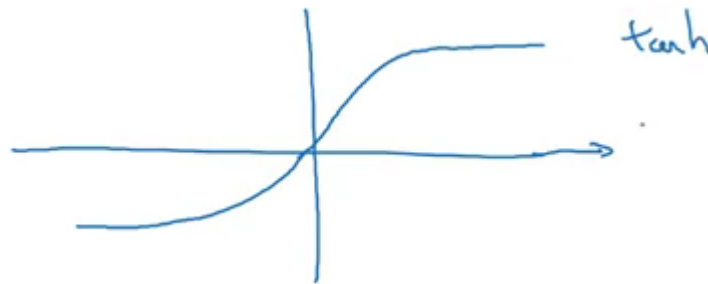
The primary reason overfitting happens is because the model learns even the tiniest details present in the data. So after learning all the possible patterns it can find, the model tends to perform extremely well on the training set but fails to produce good results on the dev and test sets. It falls apart when faced with previously unseen data.

One way to prevent overfitting is to reduce the complexity of the model. This is exactly what regularization does! If we set the regularization parameter ( $\lambda$ ) to a large value, the decay in the weights during gradient descent update will be more. Hence, the weights of most of the hidden units will be close to zero.

Since the weights are negligible, the model will not learn much from these units. This will end up making the network simpler and thus reduce overfitting:



Let's understand this concept through one more example. Suppose we use the *tanh* activation function:

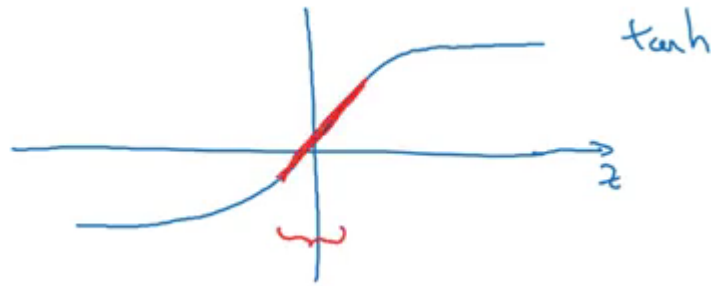


Now, if we set  $\lambda$  to a large value, the weight of the units  $w[l]$  will be less. To calculate the  $z[l]$  value, we will use the following formula:

$$z[l] = w[l] a^{[l-1]} + b[l]$$

Hence, the  $z$ -value will be less. If we use the *tanh* activation function, these low values of  $z[l]$  will lie near the origin:

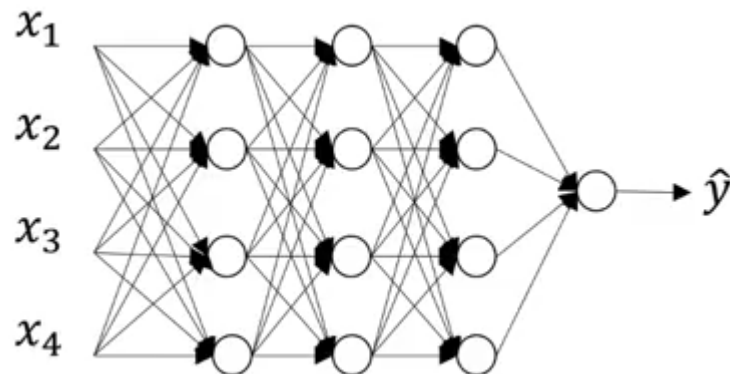




Here we are using only the linear region of the *tanh* function. This will make every layer in the network roughly linear, i.e., we will get linear boundaries that separate the data, thus preventing overfitting.

## Dropout Regularization

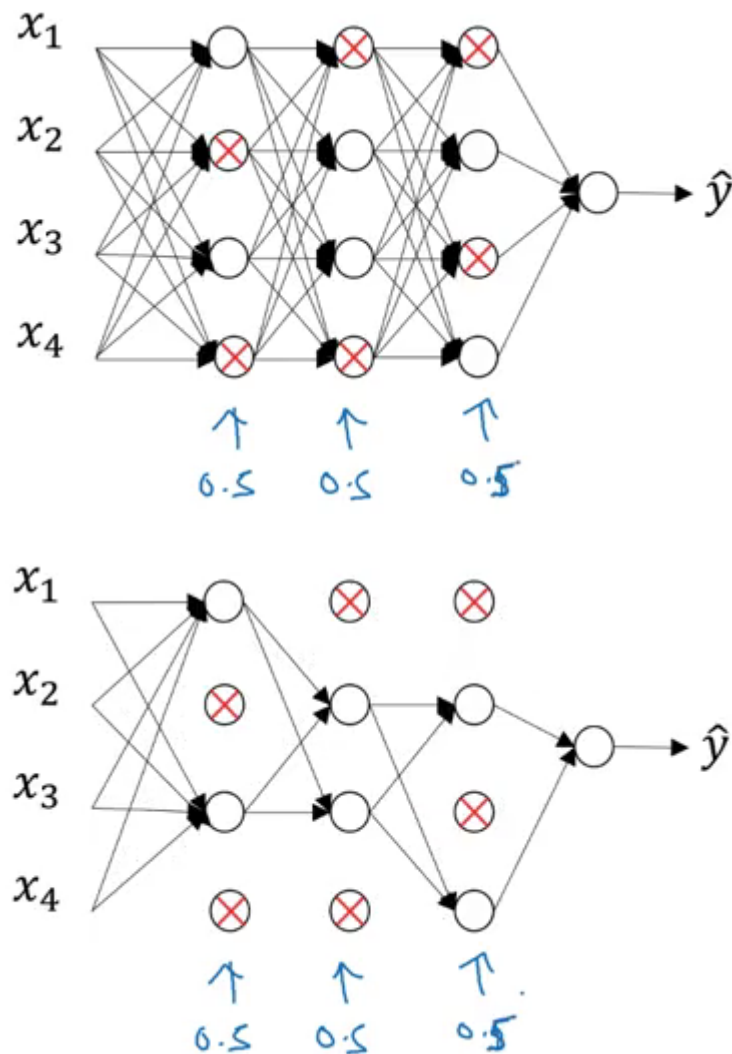
There is one more technique we can use to perform regularization. Consider you are building a neural network as shown below:



This neural network is overfitting on the training data. Suppose we add a dropout of 0.5 to all these images. The model will randomly remove 50% of the units from each layer and we finally end up with a much simpler network:







This has proven to be a very effective regularization technique. How can we implement it ourselves? Let's check it out!

We will be working on this very example where we have three hidden layers. For now, we will consider the third layer,  $l=3$ . The dropout vector  $d$  for the third hidden layer can be written as:

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
```

Here, *keep\_prob* is the probability of keeping a unit. Now, we will calculate the activations for the selected units:

```
a3 = np.multiply(a3, d3)
```

This  $a^3$  value will be reduced by a factor of  $(1 - \text{keep\_probs})$ . So to get the expected value of  $a^3$ , we divide the value:

$$a^3 / (1 - \text{keep\_probs})$$

Let's understand the concept of dropout using an example:

Number of units in the layer = 50

keep\_prob = 0.8

So, 20% of the total units (i.e. 10) will be randomly shut off.

Different sets of hidden layers are dropped randomly in each training iteration. Note that dropout is only done at the time of training the model (not during the test phase). The reason for doing this is because:

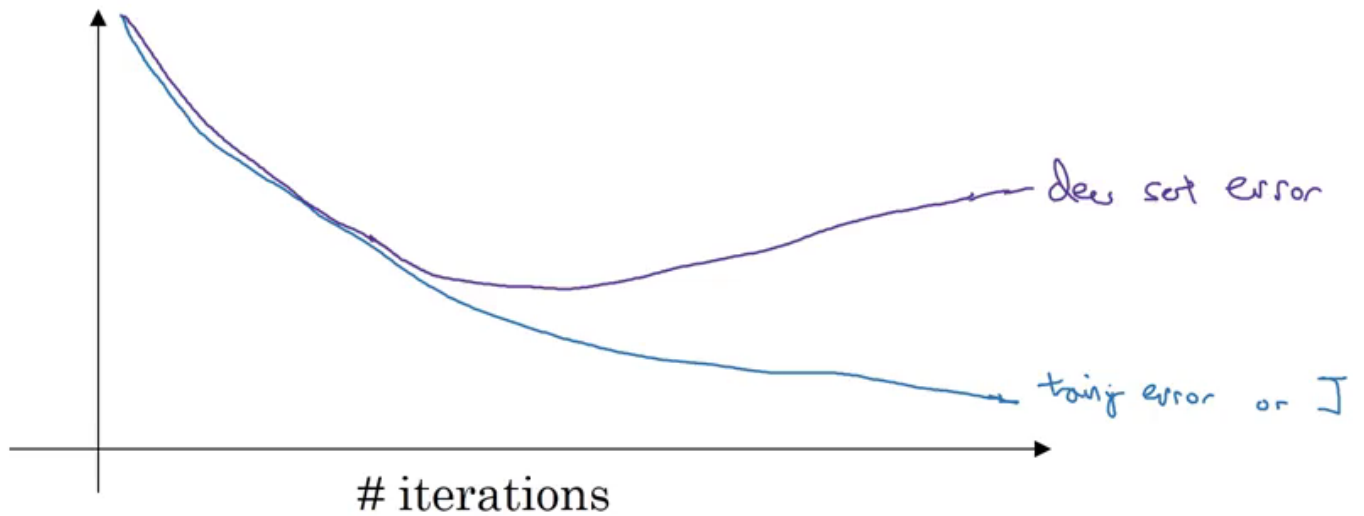
- We don't want our outputs to be random
- Dropout adds noise to the predictions

## Other Regularization Methods

Apart from L2 regularization and dropout, there are a few other techniques that can be used to reduce overfitting.

1. **Data Augmentation:** Suppose we are building an image classification model and are lacking the requisite data due to various reasons. In such cases, we can use data augmentation, i.e., applying some changes such as flipping the image, taking random crops of the image, randomly rotating images, etc. These can potentially help us get more training data and hence reduce overfitting.
2. **Early Stopping:** To understand this, consider the below example:





Here, the training error is continuously decreasing with respect to time. On the other hand, the dev set error is decreasing initially before increasing after a few iterations. We can stop the training at the point where the dev set error starts to increase. This, in a nutshell, is called early stopping.

And that is a wrap as far as regularization techniques are concerned!

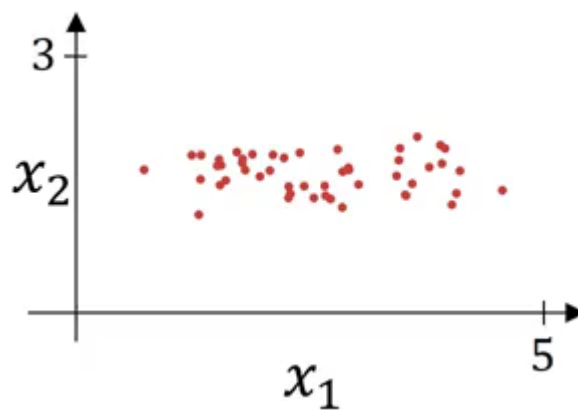
### Part III: Setting up your Optimization Problem

In this module, we will discuss the different techniques that can be used to speed up the training process.

#### Normalizing Inputs

Suppose we have 2 input features and their scatter plot looks like this:





This is how we can represent the input as a vector:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

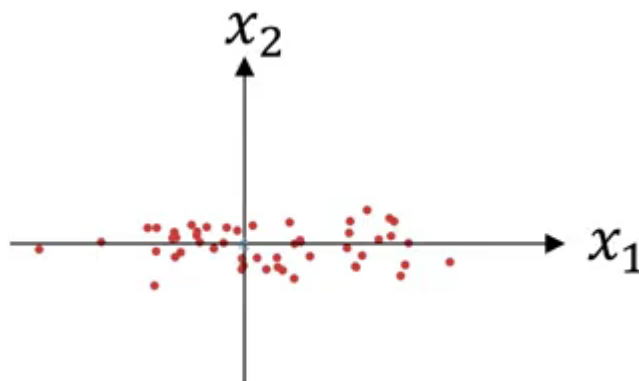
We'll follow the below steps to normalize the input:

1. Subtract the mean from the input features:

$$\mu = \frac{1}{n} \sum_{i=1}^n x^{(i)}$$

$$x := x - \mu$$

This will change the scatter plot from what is shown above to (notice that the variables have much higher variance in this graph):



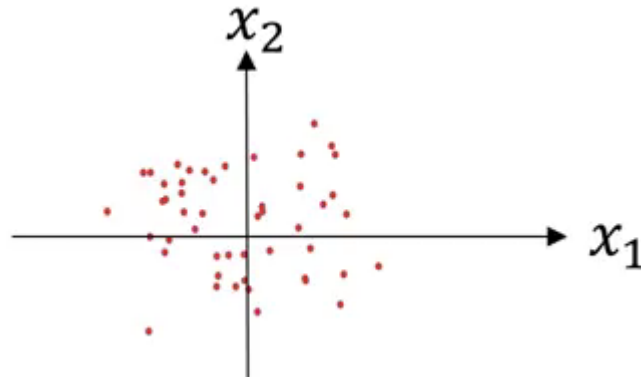
2. Next, we normalize the variance:



$$C^2 = \frac{1}{n} \sum_{i=1}^n x^{(i)} * x^2$$

$$\times 1 = \sigma^2$$

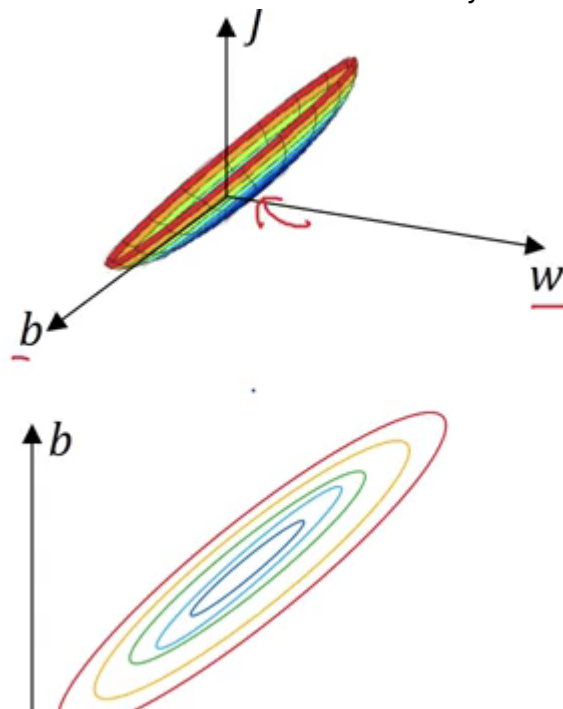
We divide the features with the variance. This will make the input look like:



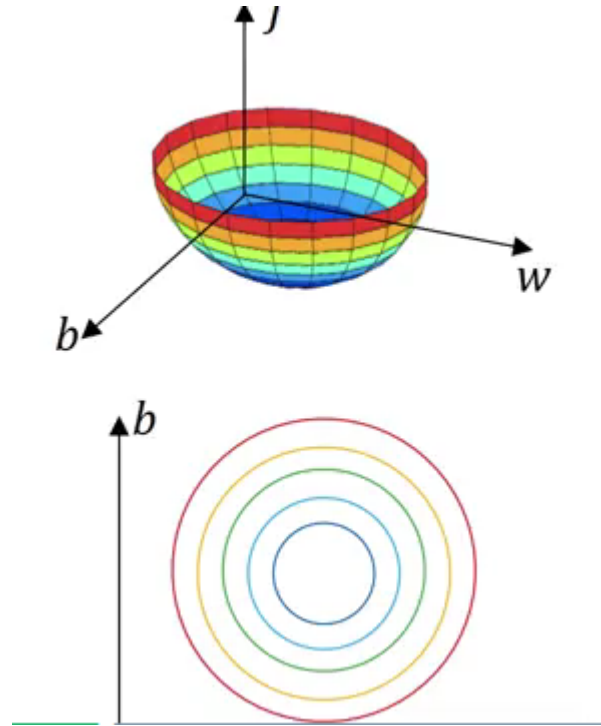
A key point to note is that we use the same mean and variance to normalize the test set as well. We should do this because we want the same transformation to happen on both the train and test data.

But **why does normalizing the data make the algorithm faster?**

In the case of unnormalized data, the scale of features will vary, and hence there will be a variation in the parameters learnt for each feature. This will make the cost function asymmetric:



Whereas, in the case of normalized data, the scale will be the same and the cost function will also be symmetric:



Normalizing the inputs makes the cost function symmetric. This makes it easier for the gradient descent algorithm to find the global minima more quickly. And this, in turn, makes the algorithm run much faster.

## Vanishing / Exploding gradients

While training deep neural networks, sometimes the derivatives (slopes) can become either very big or very small. It can make the training phase quite difficult. This is the problem of vanishing / exploding gradients. Suppose we are using a neural network with 'l' layers with two input features and we initialized the large weights:

$$W^{[2]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

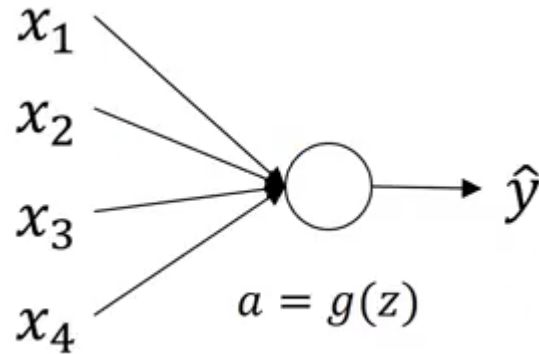
The final output at the  $l$ th layer will be (consider we are using a linear activation function):

$$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x$$

For deeper networks, L will be large, making the gradients very large and the learning process slow. Similarly, using small weights will make the gradients very small, and as a result, learning will be slow. We must deal with this problem in order to reduce the training time. So **how should the weights be initialized?**

## Weight Initialization for Deep Networks

One potential solution to this problem can be random initialization. Consider a single neuron as shown below:



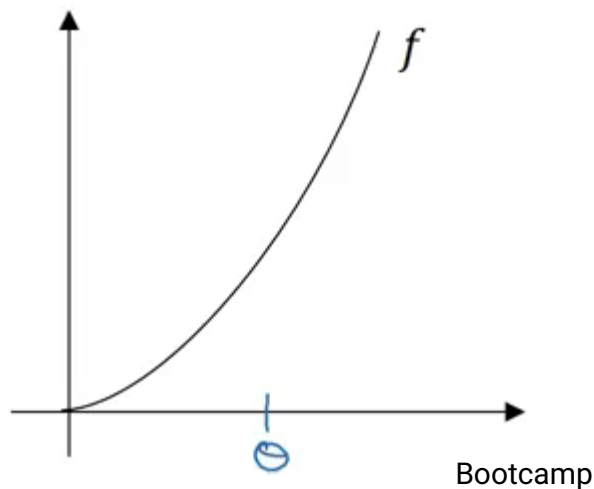
For this example, we can initialize the weights as:

$$W^{[1]} = np.random.randn(\text{shape}) * np.sqrt\left(\frac{2}{n^{[1-1]}}\right)$$

The primary reason behind initializing the weights randomly is to break symmetry. We want to make sure that different hidden units learn different patterns. There is one more technique that can help ensure that our implementations are correct and will run quickly.

## Gradient Checking

Gradient checking is used to find bugs (if any) in the implementation of backpropagation. Consider the following graph:



The derivative of the function w.r.to  $\Theta$  can be best expressed as:

$$\frac{f(\Theta + \epsilon) - f(\Theta - \epsilon)}{2\epsilon}$$

Where  $\epsilon$  is the small step which we take towards the left and right of  $\Theta$ . Make sure that the derivative calculated above is nearly equal to the actual derivative of the function. Below are the steps we follow for gradient checking:

- Take  $W[1], b[1], \dots, w[L], b[L]$  and reshape it into a big vector  $\Theta$ :

$$J(w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}) = J(\Theta)$$

- We also calculate  $dW[1], db[1], \dots, dw[L], db[L]$  and reshape it into a big vector  $d\Theta$
- Finally, we check whether  $d\Theta$  is the gradient of  $J(\Theta)$

For each  $i$ , we calculate:

$$d\Theta_{\text{approx}}[i] = \frac{J(\Theta_1, \Theta_2, \dots, \Theta_i + \epsilon, \dots) - J(\Theta_1, \Theta_2, \dots, \Theta_i - \epsilon, \dots)}{2\epsilon}$$

$$d\Theta_{\text{approx}}[i] \approx d\Theta[i] = \frac{\partial J}{\partial \Theta_i}$$

We use Euclidean distance ( $\epsilon$ ) to measure whether both these terms are equal:

$$\frac{\|d\Theta_{\text{approx}} - d\Theta\|_2}{\|d\Theta_{\text{approx}}\|_2 + \|d\Theta\|_2}$$

We want this value to be as small as possible. So, if  $\epsilon$  is  $10^{-7}$ , we say it is a great approximation, If  $\epsilon$  is  $10^{-5}$ , it is acceptable, and if  $\epsilon$  is in the range  $10^{-3}$ , we have to change the approximations and recalculate the weights.

And that's a wrap for module 1!

## Module 2: Optimization Algorithms



The objectives behind this module are:

- To learn different optimization methods such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam
- Using random mini batches to accelerate the convergence and improve the optimization
- To know the benefits of learning rate decay and apply it to your optimization

## Mini-Batch Gradient Descent

We saw in course 1 how vectorization can help us to effectively work with 'm' training examples. We can get rid of explicit for loops and make the training phase faster. So, we take the training examples as:

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & \dots & x^{(m)} \end{bmatrix}$$

$(n_x, m)$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & \dots & y^{(m)} \end{bmatrix}$$

$(1, m)$

Where X is the vectorized input and Y is their corresponding outputs. But what will happen if we have a large training set, say  $m = 5,000,000$ ? If we process through all of these training examples in every training iteration, the gradient descent update will take a lot of time. Instead, we can use a mini batch of the training examples and update the weights based on them.

Suppose we make a mini-batch containing 1000 examples each. This means we have 5000 batches and the training set will look like this:

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(1000)} & | & x^{(1001)} & \dots & x^{(2000)} & | & \dots & | & \dots & x^{(m)} \end{bmatrix}$$

$(n_x, m)$

$X^{\{1\}}$                        $X^{\{2\}}$                        $X^{\{5,000\}}$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(1000)} & | & y^{(1001)} & \dots & y^{(2000)} & | & \dots & | & \dots & y^{(m)} \end{bmatrix}$$

$(1, m)$

$Y^{\{1\}}$                        $Y^{\{2\}}$                        $Y^{\{5,000\}}$

Here,  $X\{t\}$ ,  $Y\{t\}$  represents the  $t$ th mini-batch input and output. Now, let's look at how to implement a mini-batch gradient descent:

```
for t = 1:No_of_batches                                # this is called an epoch
    A[L], Z[L] = forward_prop(X{t}, Y{t})
    cost = compute_cost(A[L], Y{t})
    grads = backward_prop(A[L], caches)
    update_parameters(grads)
```

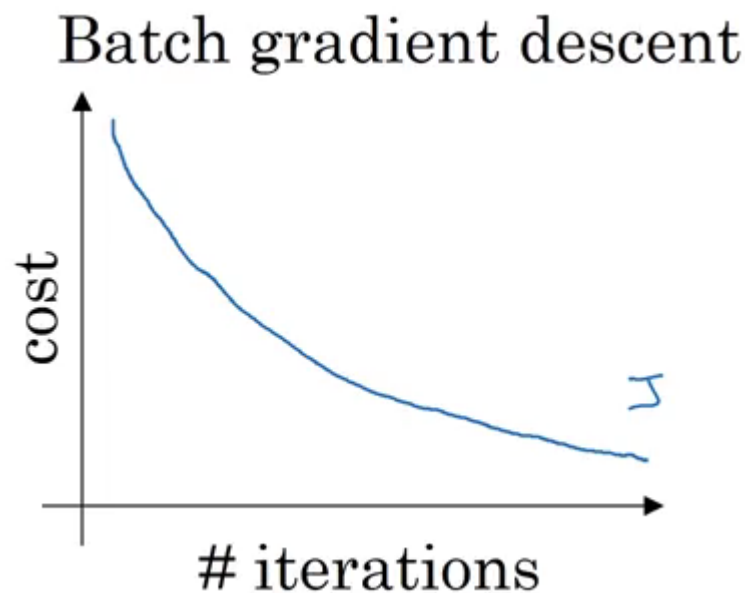
This is equivalent to 1 epoch (1 epoch = single pass through the training set). Note that the cost function for mini batch is given as:

$$J^{\text{batch}} = \frac{1}{1000} \sum_{i=1}^k \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_l \|W^{(l)}\|_F^2$$

Where 1000 is the number of mini batches we saw in our above example. Let's dive deeper and understand mini-batch gradient descent in detail.

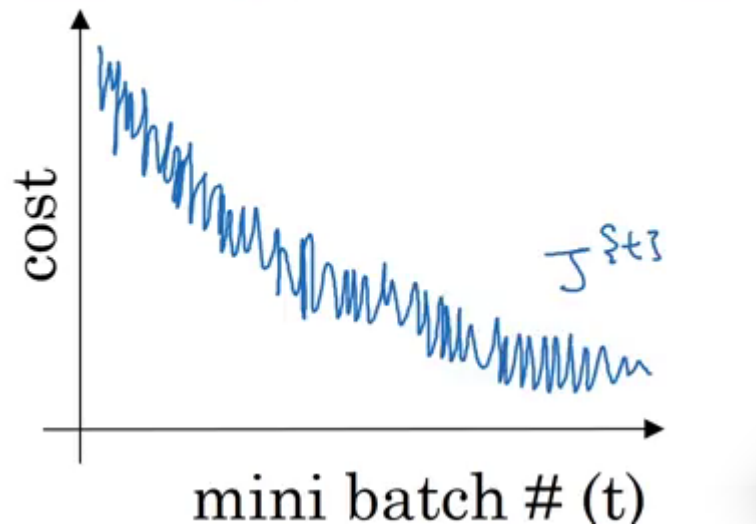
## Understanding Mini-Batch Gradient Descent

In batch gradient descent, our cost function should decrease on every single iteration:



In the case of mini-batch gradient descent, we only use a specified set of training examples. As a result, the cost function can decrease for some iterations:

# Mini-batch gradient descent



How can we choose a mini-batch size? Let's see various cases:

1. If the mini-batch size =  $m$ :

It is a batch gradient descent where all the training examples are used in each iteration. It takes too much time per iteration.

2. If the mini-batch size = 1:

It is called stochastic gradient descent, where each training example is its own mini-batch. Since in every iteration we are taking just a single example, it can become extremely noisy and takes much more time to reach the global minima.

3. If the mini-batch size is between 1 to  $m$ :

It is mini-batch gradient descent. The size of the mini-batch should not be too large or too small.

Below are a few general guidelines to keep in mind while deciding the mini-batch size:

1. If the training set is small, we can choose a mini-batch size of  $m < 2000$
2. For a larger training set, typical mini-batch sizes are: 64, 128, 256, 512
3. Make sure that the mini-batch size fits your CPU/GPU memory

## Exponentially weighted averages

Below is a sample of hypothetical temperature data collected for an entire year:

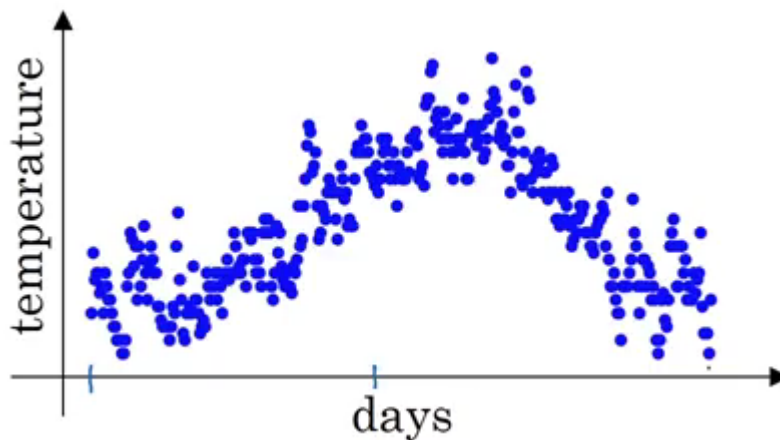


```

01 = 40 F
02 = 49 F
03 = 45 F
.
.
0180 = 60 F
0181 = 56 F
.
.

```

The below plot neatly summarizes this temperature data for us:



Exponentially weighted average, or exponentially weighted moving average, computes the trends. We will first initialize a term as 0:

$$V_0 = 0$$

Now, all the further terms will be calculated as the weighted sum of  $V_0$  and the temperature of that day:

$$V_1 = 0.9 * V_0 + 0.1 * \theta_1$$

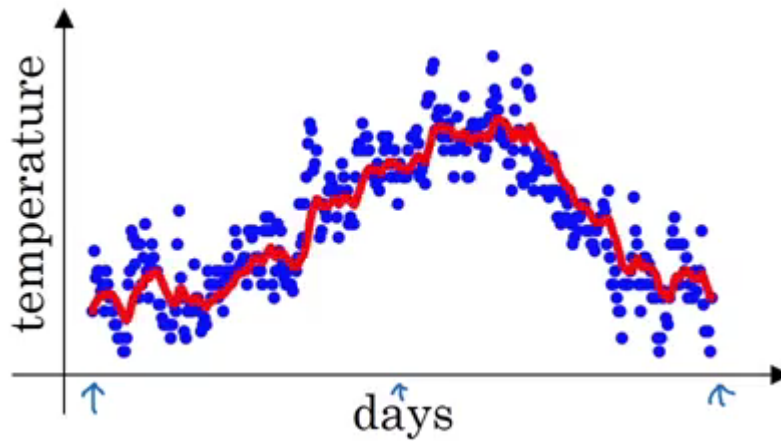
$$V_2 = 0.9 * V_1 + 0.1 * \theta_2$$

And so on. A more generalized form of exponentially weighted average can be written as:

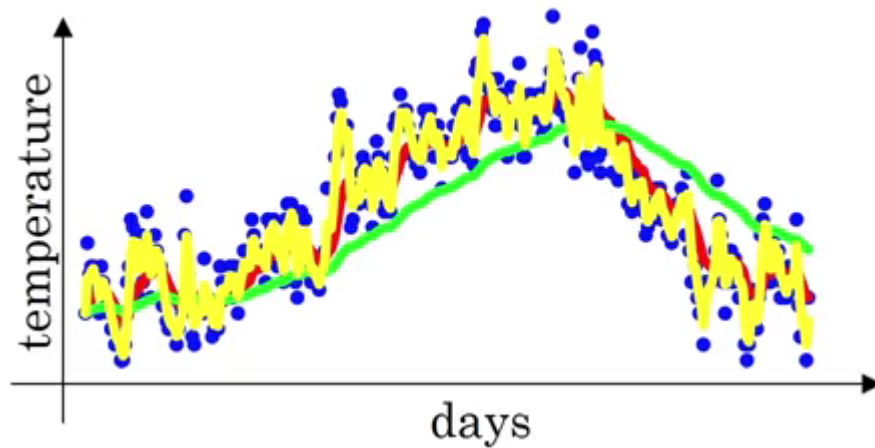
$$V_t = \beta * V_{(t-1)} + (1 - \beta) * \theta_t$$

Using this equation for trend, the data will be generalised as:





The  $\beta$  value in this example is 0.9, which means  $V_t$  is an approximation of average over  $1/(1-\beta)$  days, i.e.,  $1/(1-0.9) = 10$  days temperature. Increasing the value of  $\beta$  will result in approximating over more days, i.e., taking the average temperature of more days. If the  $\beta$  value is small, i.e., we use only 1 day's data for approximation, the predictions become much more noisy:



Here, the green line is the approximation when  $\beta = 0.98$  (using 50 days) and the yellow line is when  $\beta = 0.5$  (using 2 days). It can be seen that using small  $\beta$  results in noisy predictions.

The equation of exponentially weighted averages is given by:

$$V_t = \beta * V_{(t-1)} + (1 - \beta) * \theta_t$$

Let's look at how we can implement this:



```

Initialize  $V_0 = 0$ 
Repeat
{
get next  $\theta_t$ 
 $V_0 = \beta * V_0 + (1 - \beta) * \theta_t$ 
}

```

This step takes a lot less memory as we are overwriting the previous values. Hence, it is a computational, as well as memory efficient, process.

## Bias Correction in Exponentially Weighted Averages

We initialize  $V_0 = 0$ , so while calculating the  $V_1$  value it will only be equal to  $(1 - \beta) * \theta_1$ . It will not generalize well to the actual values. We need to use bias correction to overcome this challenge.

Instead of using the previous equation, i.e.,

$$V_t = \beta * V_{(t-1)} + (1 - \beta) * \theta_t$$

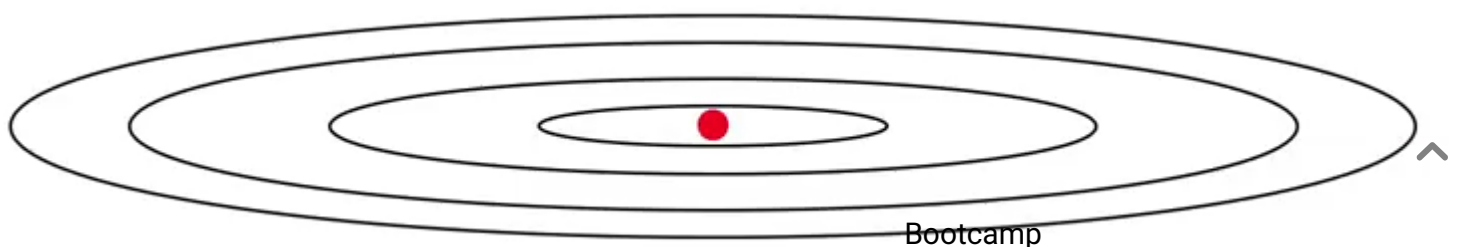
we include a bias correction term:

$$V_t = [\beta * V_{(t-1)} + (1 - \beta) * \theta_t] / (1 - \beta_t)$$

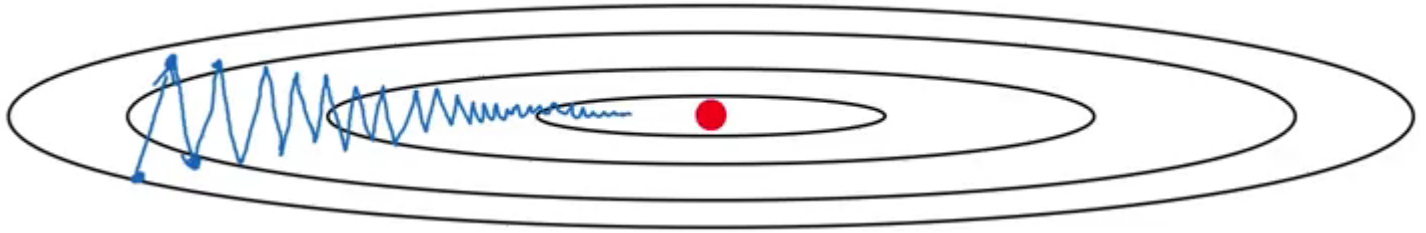
When  $t$  is small,  $\beta_t$  will be large, resulting in a smaller value of  $(1 - \beta_t)$ . This will make the  $V_t$  value larger ensuring that our predictions are accurate.

## Gradient Descent with Momentum

The underlying idea of gradient descent with momentum is to calculate the exponential weighted average of gradients and use them to update weights. Suppose we have a cost function whose contours look like this:



The red dot is the global minima, and we want to reach that point. Using gradient descent, the updates will look like:



One more way could be to use a larger learning rate. But that could result in large upgrade steps, and we might not reach global minima. Additionally, too small a learning rate makes the gradient descent slower. **We want a slower learning in the vertical direction and a faster learning in the horizontal direction which will help us to reach the global minima much faster.**

Let's see how we can achieve it using momentum:

On iteration  $t$ :

*Compute  $dW$ ,  $dB$  on current mini-batch using momentum*

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

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

*Update weights*

$$W = W - \alpha * V_{dW}$$

$$b = b - \alpha * V_{db}$$

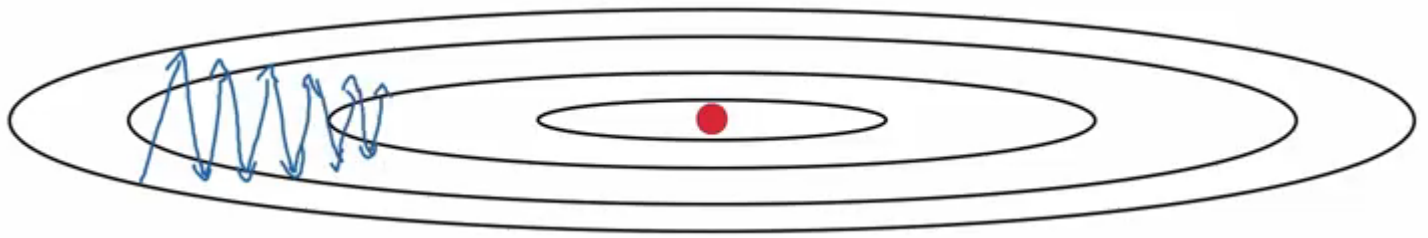
Here, we have two hyperparameters, i.e.,  $\alpha$  and  $\beta$ . The role of  $dW$  and  $db$  in the above equation is to provide momentum,  $V_{dW}$  and  $V_{db}$  provides velocity, and  $\beta$  acts as friction and prevents speeding over the limit. Consider a ball rolling down –  $V_{dW}$  and  $V_{db}$  provide velocity to that ball and make it move faster. We do not want our ball to speed up so much that it misses the global minima, and hence  $\beta$  acts as friction.

Let me introduce you to a few more optimization algorithms.

## RMSprop

Consider the example of a simple gradient descent:





Suppose we have two parameters  $w$  and  $b$  as shown below:



Look at the contour shown above and the parameters graph. We want to slow down the learning in  $b$  direction, i.e., the vertical direction, and speed up the learning in  $w$  direction, i.e., the horizontal direction. The steps followed in RMSprop can be summarised as:

On iteration  $t$ :

*Compute  $dW$ ,  $dB$  on current mini-batch*

$$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * dW^2$$

$$V_{db} = \beta_2 * S_{db} + (1 - \beta_2) * db^2$$

*Update weights*

$$W = W - \alpha * (dW / S_{dW})$$

$$b = b - \alpha * (db / S_{db})$$

The slope in the vertical direction ( $db$  in our case) is steeper, resulting in a large value of  $S_{db}$ . As we want slow learning in the vertical direction, dividing  $db$  with  $S_{db}$  in update step will result in a smaller change in  $b$ . Hence, learning in the vertical direction will be less. Similarly, a small value of  $S_{dW}$  will result in faster learning in the horizontal direction, thus making the algorithm faster.

## Adam optimization algorithm

Adam is essentially a combination of momentum and RMSprop. Let's see how we can implement it:





$$V_{dW} = 0, S_{dW} = 0, V_{db} = 0, S_{db} = 0$$

On iteration  $t$ :

*Compute  $dW$ ,  $dB$  on current mini-batch using momentum and RMSprop*

$$V_{dW} = \beta_1 * V_{dW} + (1 - \beta_1) * dW$$

$$V_{db} = \beta_1 * V_{db} + (1 - \beta_1) * db$$

$$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * dW^2$$

$$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * db^2$$

*Apply bias correction*

$$V_{dW}^{\text{corrected}} = V_{dW} / (1 - \beta_1^t)$$

$$V_{db}^{\text{corrected}} = V_{db} / (1 - \beta_1^t)$$

$$S_{dW}^{\text{corrected}} = S_{dW} / (1 - \beta_2^t)$$

$$S_{db}^{\text{corrected}} = S_{db} / (1 - \beta_2^t)$$

*Update weights*

$$W = W - \alpha * (V_{dW}^{\text{corrected}} / S_{dW}^{\text{corrected}} + \epsilon)$$

$$b = b - \alpha * (V_{db}^{\text{corrected}} / S_{db}^{\text{corrected}} + \epsilon)$$

There are a range of hyperparameters used in Adam and some of the common ones are:

- **Learning rate  $\alpha$** : needs to be tuned
- **Momentum term  $\beta_1$** : common choice is 0.9
- **RMSprop term  $\beta_2$** : common choice is 0.999
- **$\epsilon$** :  $10^{-8}$

Adam helps to train a neural network model much more quickly than the techniques we have seen earlier.

## Learning Rate Decay

If we slowly reduce the learning rate over time, we might speed up the learning process. This process is called learning rate decay.



Initially, when the learning rate is not very small, training will be faster. If we slowly reduce the learning rate, there is a higher chance of coming close to the global minima.

Learning rate decay can be given as:

$$\alpha = [1 / (1 + \text{decay\_rate} * \text{epoch\_number})] * \alpha_0$$

Let's understand it with an example. Consider:

- $\alpha_0 = 0.2$
- $\text{decay\_rate} = 1$

epoch_number	$\alpha$
1	$[1/(1+1)]*0.2 = 0.1$
2	$[1/(1+2)]*0.2 = 0.067$
3	0.05
4	0.04

This is how, after each epoch, there is a decay in the learning rate which helps us reach the global minima much more quickly. There are a few more learning rate decay methods:

1. **Exponential decay:**  $\alpha = (0.95)^{\text{epoch\_number}} * \alpha_0$
2.  $\alpha = k / \text{epochnumber}^{1/2} * \alpha_0$
3.  $\alpha = k / t^{1/2} * \alpha_0$

Here,  $t$  is the mini-batch number.

This was all about optimization algorithms and module 2! Take a deep breath, we are about to enter the final module of this article.

## Module 3: Hyperparameter tuning, Batch Normalization and Programming Frameworks

The primary objectives of module 3 are:

- To master the process of hyperparameter tuning
- To familiarize yourself with the concept of Batch Normalization

Bootcamp



Much like the first module, this is further divided into three sections:

- Part I: Hyperparameter tuning
- Part II: Batch Normalization
- Part III: Multi-class classification

## Part I: Hyperparameter tuning

### Tuning process

Hyperparameters. We see this term popularly being bandied about in data science competitions and hackathons. But how important is it in the overall scheme of things?

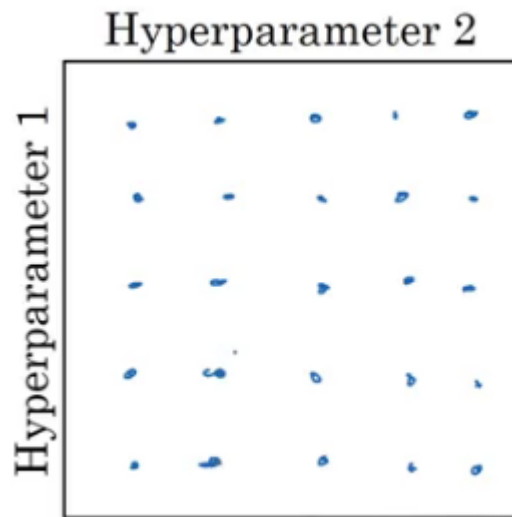
Tuning these hyperparameters effectively can lead to a massive improvement in your position on the leaderboard. Following are a few common hyperparameters we frequently work with in a deep neural network:

- Learning rate –  $\alpha$
- Momentum –  $\beta$
- Adam's hyperparameter –  $\beta_1, \beta_2, \epsilon$
- Number of hidden layers
- Number of hidden units for different layers
- Learning rate decay
- Mini-batch size

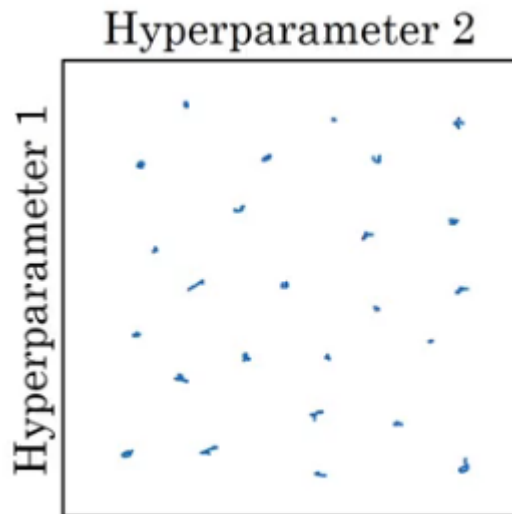
Learning rate usually proves to be the most important among the above. This is followed by the number of hidden units, momentum, mini-batch size, the number of hidden layers, and then the learning rate decay.

Now, suppose we have two hyperparameters. We sample the points in a grid and then systematically explore these values. Consider a five-by-five grid:





We check all 25 values and pick whichever hyperparameter works best. Instead of using these grids, we can try random values as well. Why? Because we do not know which hyperparameter value might turn out to be important, and in a grid we only define particular values.

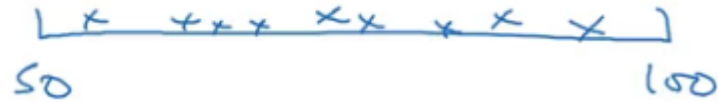


The major takeaway from this sub-section is to use random sampling and adequate search.

### Using an Appropriate Scale to Pick Hyperparameters

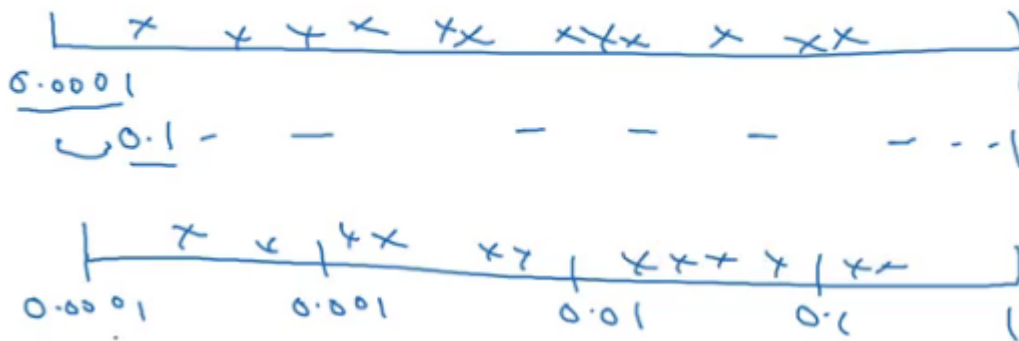
To understand this, consider the number of hidden units hyperparameter. The range we are interested in is from 50 to 100. We can use a grid which contains values between 50 and 100 and use that to find the best value: ^

$$n^{test} = 50, \dots, 100$$



Now consider the learning rate with a range between 0.0001 and 1. If we draw a number line with these extreme values and sample the values uniformly at random, around 90% of the values will fall between 0.1 to 1. In other words, we are using 90% resources to search between 0.1 to 1, and only 10% to search between 0.0001 to 0.1. This does not look correct! Instead, we can use a log scale to choose the values:

$$\alpha = 0.0001, \dots, 1$$



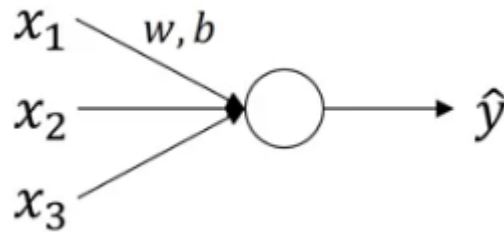
Next, we will learn a technique which makes our neural network much more robust to the choice of hyperparameters and also makes the training phase even more faster.

## Part II: Batch Normalization

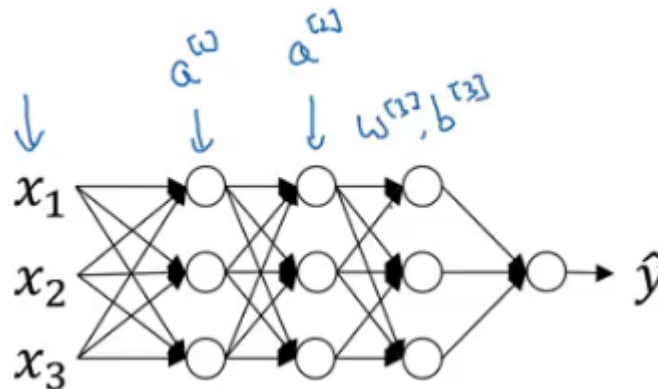
### Normalizing Activations in a Network

Let's recall how a logistic regression looks like:





We have seen how normalizing the input in this case can speed up the learning process. In case of deep neural networks, we have a lot of hidden layers and this results in a lot of activations:



Wouldn't it be great if we can normalize the mean and variance of these activations ( $a[2]$ ) in order to make the training of  $w[3]$ ,  $b[3]$  more effective? This is how batch normalization works. We normalize the activations of the hidden layer(s) so that the weights of the next layer can be updated faster. Technically, we normalize the values of  $z[2]$  and then use an activation function of the normalized values. Here is how we can implement batch normalization:

Given some intermediate values in NN  $z(1), \dots, z(m)$ :

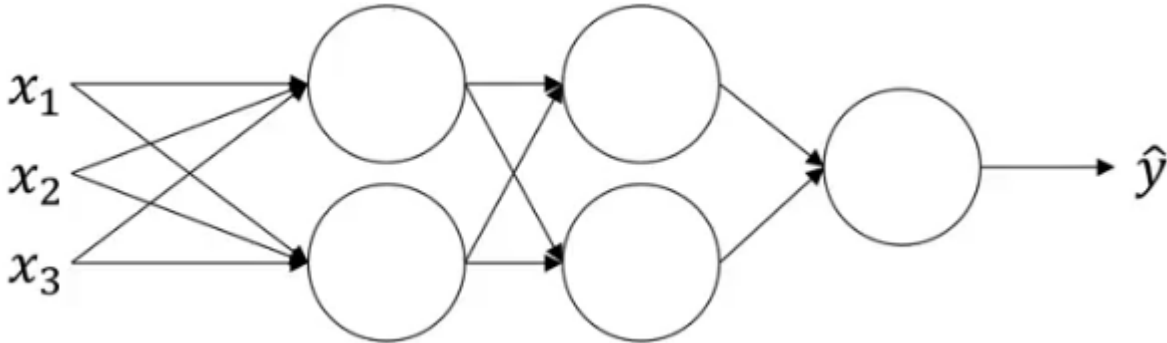
$$\begin{aligned}\mu &= \frac{1}{m} \sum_i z^{(i)} \\ \sigma^2 &= \frac{1}{m} \sum_i (z_i - \mu)^2 \\ z_{\text{norm}}^{(i)} &= \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{z}^{(i)} &= \gamma z_{\text{norm}}^{(i)} + \beta\end{aligned}$$

Here,  $\gamma$  and  $\beta$  are learnable parameters.

Bootcamp

## Fitting Batch Norm into a Neural Network

Consider the neural network shown below:



Each unit of the neural network computes two things. It first computes  $Z$ , and then applies the activation function on it to compute  $A$ . If we apply batch norm at each layer, the computation will look like:

$$x \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \xrightarrow{W^{[3]}, b^{[3]}} z^{[3]} \xrightarrow{\beta^{[3]}, \gamma^{[3]}} a^{[3]} = \hat{y}$$

After calculating the  $Z$ -value, we apply batch norm and then the activation function on that. Parameters in this case are:

$$\text{Parameters: } W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}, \\ \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$$

Finally, let's see how we can apply gradient descent using batch norm:

For  $t=1, \dots$ , number of batches:

Compute forward propagation on  $X\{t\}$

In each hidden layer, use batch normalization

Use backpropagation to compute  $dW^{[l]}$ ,  $db^{[l]}$ ,  $d\beta^{[l]}$  and  $d\gamma^{[l]}$

Update the parameters:

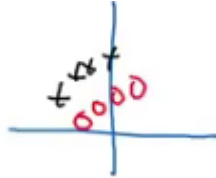
$$W^{[l]} = W^{[l]} - \alpha * dW^{[l]}$$

$$\beta^{[l]} = \beta^{[l]} - \alpha * d\beta^{[l]}$$

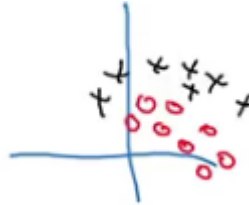
Note that this also works with momentum, RMSprop and Adam.

## How does Batch Norm work?

In the case of logistic regression, we now know how normalizing the inputs helps to speed up the learning. Batch norm works in much the same way. Let's take one more use case to understand it better. Consider the training set for a binary classification problem:



But when we try to generalize it to a dataset having different distribution, say:



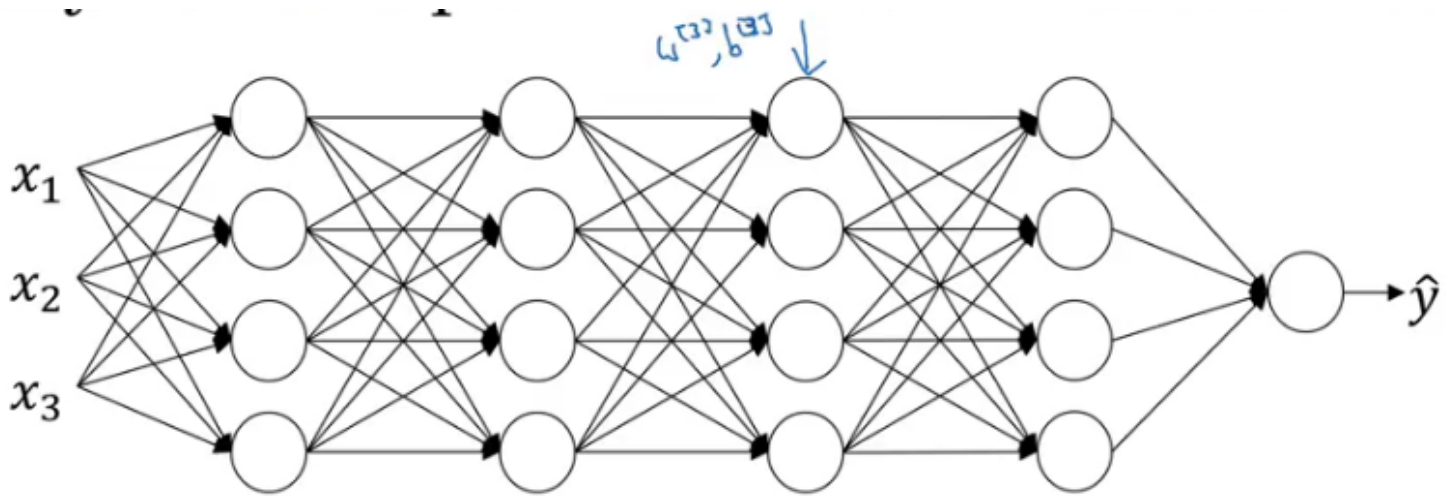
The decision boundary in both the cases might be same:



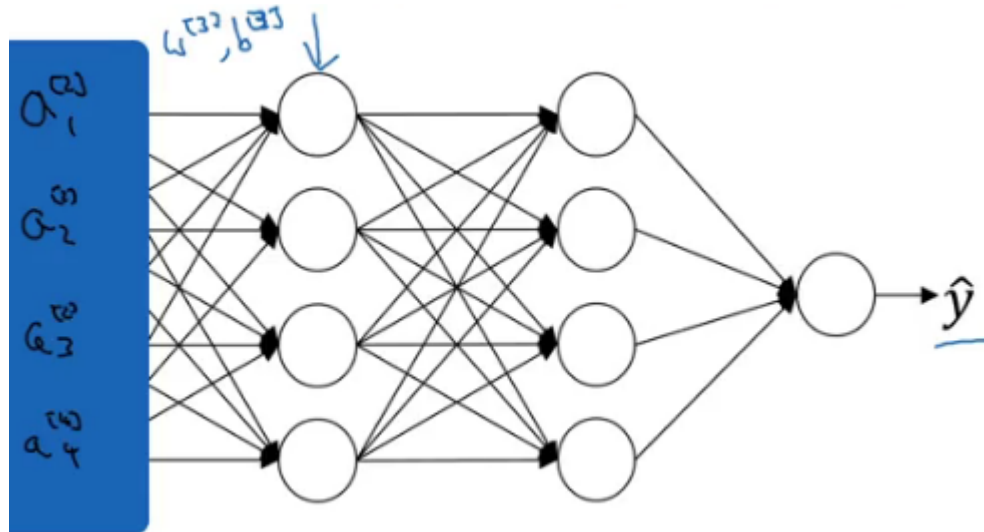
But the model would not be able to discover this green decision boundary. So, as the distribution of the input changes, we might have to train the model again. Consider a deep neural network:







And let's only consider the learning of layer 3. It will have activations from layer two as its input:



The aim of the third hidden layer is to take these activations and map them with the output. These activations change every time as the parameters of the previous layers change. Hence, we see a lot of shift in the activation values. Batch norm reduces the amount that the distribution of these hidden unit values shift around.

Additionally, it turns out that batch norm has a regularization effect as well:

- Each mini-batch is normalized using the mean / variance computed just on that mini batch
- This adds some noise to the values of  $z[l]$  within that mini batch, which is similar to the effect of dropout
- Hence this has a slight effect of regularization as well

One thing to note is that while making predictions, there is a slight difference in the way we use batch normalization.

## Batch Norm at Test Time

We need to process the examples one at a time when making predictions on the test data. In the training period, the steps of batch norm can be written as:


$$\begin{aligned}\mu &= \frac{1}{m} \sum_i z^{(i)} \\ \sigma^2 &= \frac{1}{m} \sum_i (z^{(i)} - \mu)^2 \\ z_{\text{norm}}^{(i)} &= \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{z}^{(i)} &= \gamma z_{\text{norm}}^{(i)} + \beta\end{aligned}$$

We first calculate the mean and variance of that mini-batch, and use that to normalize the z-value. We will be using the entire mini-batch to calculate the mean and standard deviation. We process each image separately, so taking the mean and standard deviation of a single image does not make sense.

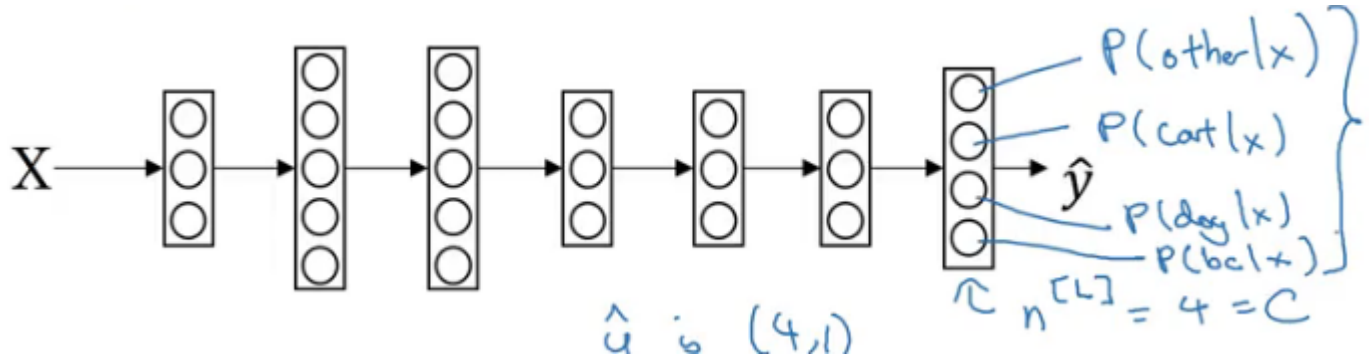
We use exponentially weighted average to calculate the mean and variance across different mini-batches. Finally, we use these values to scale the test data.

## Part III: Multi-Class Classification

### Softmax Regression

Binary classification means dealing with two classes. But when we have more than two classes in a problem, that is called multi-class classification. Suppose we have to recognize cats, dogs and tigers in a set of images. How many types of classes are there? 4 – cat, dog, tiger and none of them. If you said three there then think again! 

For solving such problems, we use softmax regression. At the output layer, instead of having a single unit, we have units equal to the total number of classes (4 in our case). Each unit tells us the probability of the image falling in different classes. Since it tells the probability, the sum of values from each unit is always equal to 1.



This is how a neural network for a multi-class classification looks like. So, for layer  $L$ , the output will be:

$$Z[L] = W[L] \cdot a[L-1] + b[L]$$

The activation function will be:

$$t = e^{z^{[L]}}$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^n t_i}$$

Let's understand this with an example. Consider the output from the last hidden layer:

$$Z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

We then calculate  $t$  using the formula given above:

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

Finally, we calculate the activations:

$a[L] =$	0.842
	0.042
	0.002
	0.114