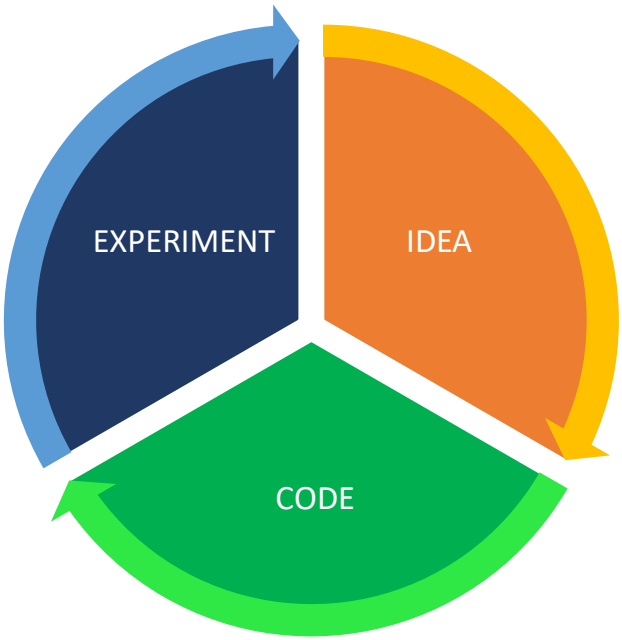
DEEP LEARNING SPECIALIZATION

COURSERA-DEEPLEARNING.AI-ANDREW NG

These are notes taken from Deep Learning Specialization by deeplearning.ai in Coursera. All the screenshots shown here are taken from the course videos. I would advise you to first watch the videos of the course in order to completely understand the notes.

COURSE 2

Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization



Hyperparameters

- **Learning rate α** → Decides the rate at which gradients the parameters change.

- **No of iterations** → Decides no of times gradient descent Is done to adjust the parameters.
- **No of hidden layers l** → Decides the number of variety of features.
- **No of hidden units $n^{[l]}$** → Decides the number of activation units.
- **Choice of activation function** → Decides the nature of features produced in the NN units.

Other than these common hyperparameters there are other parameters that we can tune by considering them as hyperparameters, which we would see as we go through the course.

All the hyperparameters are tuned to decide which ones are best to get the best results. It's very rare to have knowledge of the best value for all these parameters in advance.

So, what we do is we experiment with the code that we written using the particular parameter values. If we fail, we get an idea about what changes to make then we code again and we test again.

In this course we would learn about how to approach these Idea →Code →Experiment method to get the best model

About this Course

This course will teach you the "magic" of getting deep learning to work well. Rather than the deep learning process being a black box, you will understand what drives performance, and be able to more systematically get good results. You will also learn TensorFlow.

After 3 weeks, you will:

1. Understand industry best-practices for building deep learning applications.

2. Be able to effectively use the common neural network "tricks", including initialization, L2 and dropout regularization, Batch normalization, gradient checking.

3. Be able to implement and apply a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and check for their convergence.

4. Understand new best-practices for the deep learning era of how to set up train/dev/test sets and analyse bias/variance

5. Be able to implement a neural network in TensorFlow.

This is the **second course** of the Deep Learning Specialization.

WEEK 1 - Setting Up Your Machine Learning Application

Train, Dev, Test sets (For small data size)

Usually when we create a machine learning model, we split our dataset into 3 parts, Train set which we use to train our model, Cross validation set which we use to tune our parameters and experiment with different models and choose which is the best and Test set which is used to finally test our model. And before big data came in, we used to have data in medium sizes like in 10,000 or 1000 etc. So, we used to give more percentage of data for testing and cross validation. But when we work with **Big data**, we have data in range of millions or even billions. So, we usually divide our data into portions like **99.5%** train, **4%** cross validation and **1%** test. So, it basically depends on your size of data that determines the ratio at which you divide the dataset into train, cross validation and test data.

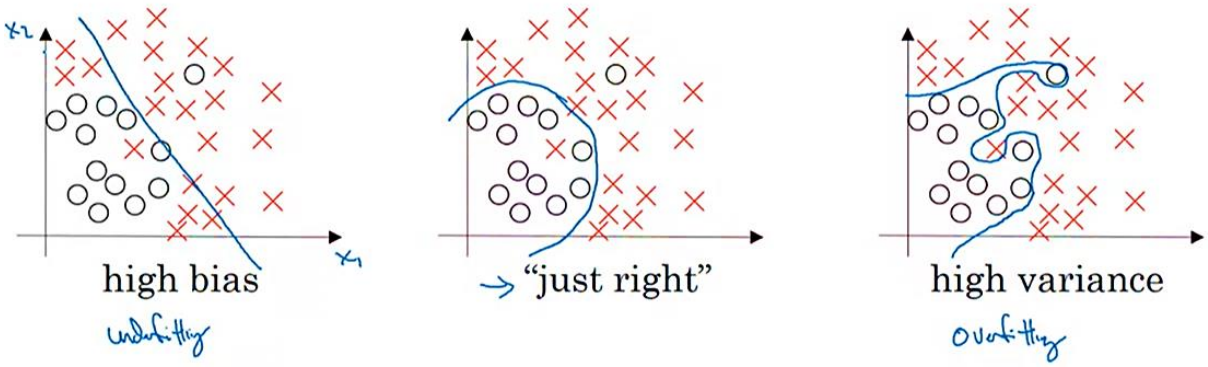
Train, Dev, Test sets (For Big Data)

Mismatched Train, Test and Dev set

When people divide train/test and dev set they might tend to choose different sources for different sets in order to collect maximum data for training set. This is not good since images

collected from different sources might have different resolutions and might not give true accuracy when we use data from different source for test set. So, its always good to divide data collected from similar sources into train, test and dev sets.

Bias and Variance



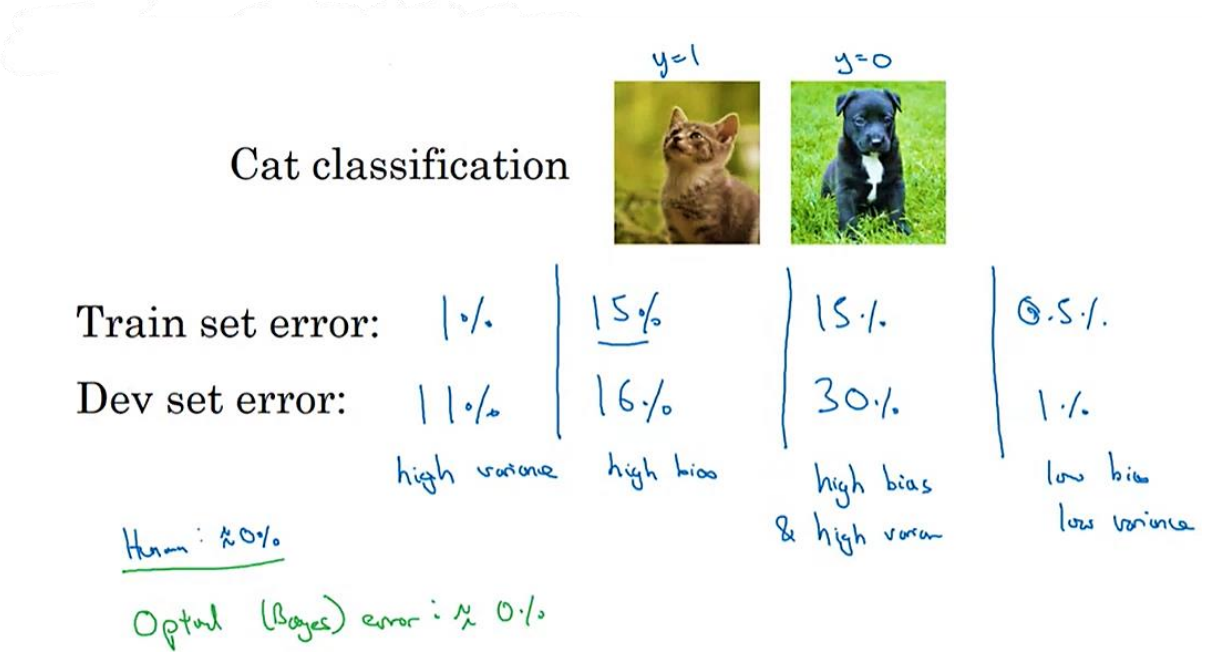
Assumptions

1. **The base error is 0**
2. **The data is from same distribution**

In the above data we can see that if we try to fit a logistic regression straight line onto the data it might not fit the data very well and we can say it has high bias as it might predict lot of the observations wrongly. The last figure shows fitting the data with that unusual curve which might fit the training data very well but not the test data. It might be able to predict all the values in the training set correctly but it might perform poorly on the test data. The middle figure shows the better fit to the data as it covers all the points very well and even though there are one or two mismatches it would perform better when we try it on test data.

- **High bias**→ Fit the training data poorly and dev data too(**underfit**)

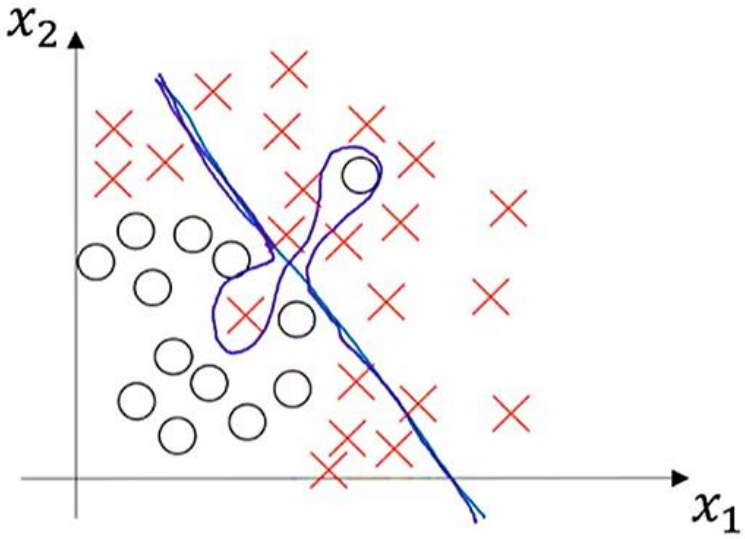
- **High variance**→ **Overfit** the training data and not good fit on dev data



In this example we classify the images into cat or not cat. We can see that we got 1% train set error and 11% Dev set error. Which shows high variance as it is fitting the training data perfectly but not the Dev data. The other case where there is a 15% error for the training data and 16% error for the dev data it means that your model is not fitting even the training data well. It shows that the model is underfitting and ha bias.

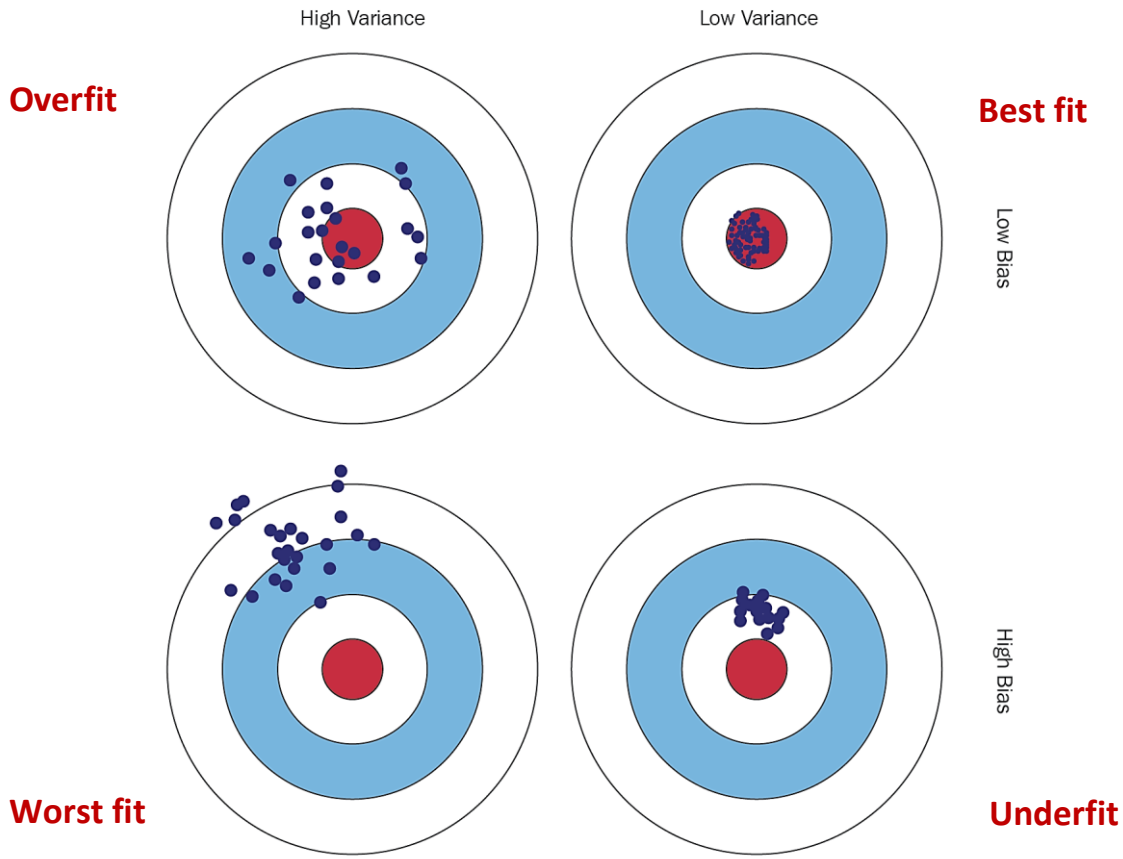
When it has 15%error on training data and 30% error on dev data it is both underfit and overfit. Let's see an example to better understand this.

High bias and high variance



We can see that the line is underfitting the data since it is linear almost in all portions but it must be a curve to fit it perfectly. At the same time, it takes a weird path in the middle and overfit the data in middle.

And finally, if your model has very low bias and variance then it is said to be good fit.



http://scott.fortmann-roe.com/docs/BiasVariance.html

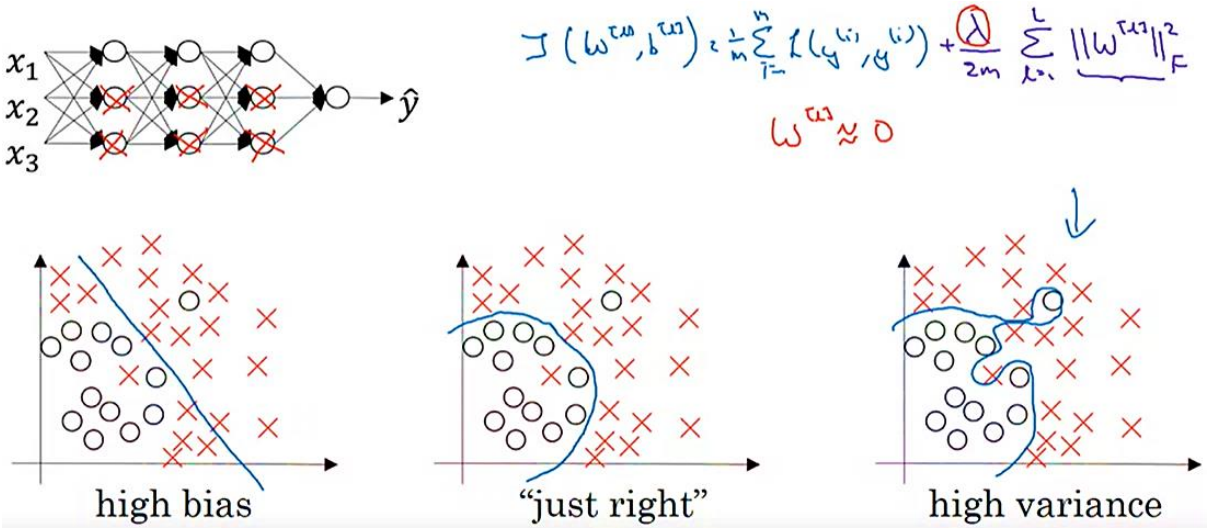
Ways to reduce high bias or variance

High Bias	High Variance
Bigger Neural Network	More Data
Train Longer	Regularization
Search for a better NN architecture	Search for a better NN architecture

If we have a model that is underfitting the training it with more layers of neural network would try to fit it better and might reduce bias. Also, we can try to train It bit longer to see its best fit. If we have high variance in our model, we can try to add more data to it so it won't overfit it, also we can try regularization to reduce overfitting. Using a bigger neural network won't overfit the data if we do regularization.

Bias variance trade-off is a term used to describe the increase in variance/bias as we decrease variance/bias.

Why Regularization reduces Overfitting?



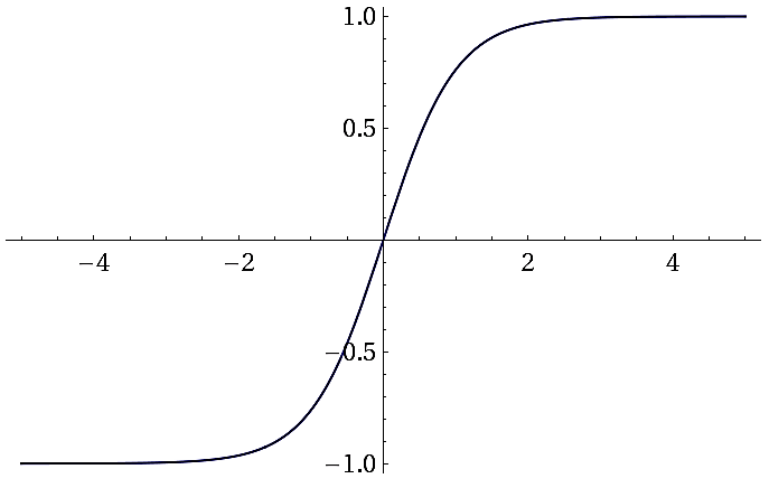
If we are including regularization term in our cost function our cost function looks like this:

$$\mathbf{J(W^{[l]}, b^{[l]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{m} \sum_{l=1}^{L} \|W^{[1]}\|_F^2}$$

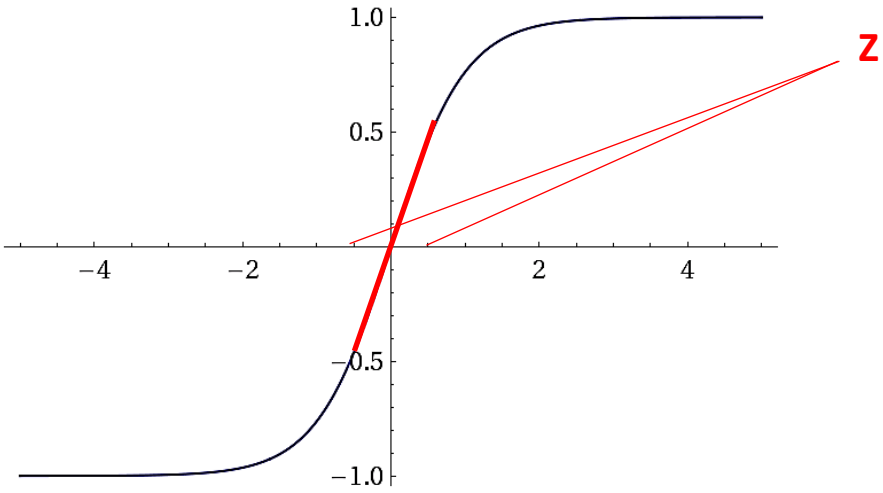
So, the parameter update would be:

$$\mathbf{W^{[l]} = W^{[l]} - \alpha(dW^{[l]} + \frac{\lambda}{m} W^{[l]})}$$

Due to the addition of the new regularization term if we increase $\boldsymbol{\lambda}$ the term $\mathbf{W^{[l]}}$ would be reduced. And some of the weights that contribute less to the model might get closer to zero. So effectively it might reduce the effect of few of the neural network nodes on the overall model and thereby reducing overfitting because most of the times overfitting might be caused when we use deeper neural networks and train our data. But we might need these deeper neural networks as it would give us additional features that we won't get from shallow networks.



If we are using the tanh function as activation function for any layers then we know as Z increases or decreases so much the functions gradient would become loser to 0. But if we are regularizing the cost function then we would decrease **W** thereby decreasing **Z** as **Z = WA +b.** Therefor for some neurons the activation function would behave as linear as the value of Z would be lying in a small region close to 0. Thereby reducing the formation of complex curves which overfit the data.



Regularization

Now let's see how to do regularization

$$\mathbf{J(W^{[l]}, b^{[l]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|W^{[l]}\|_F^2}$$

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

The norm $\|\mathbf{W^{[l]}}\|_\mathbf{F}^\mathbf{2}$ is called Forbenius norm.

So $\mathbf{dW^{[l]} = \frac{1}{m} dZ^l A^{[l-1]} + \frac{\lambda}{m} W^{[l]}}$

$\mathbf{W^{[l]} = W^{[l]} - \alpha(dW^{[l]} + \frac{\lambda}{m} W^{[l]})}$

This is also called weight decay.

Here we can see that how regularization is controlling the weights further and thereby reducing the chance of overfitting. It doesn't allow the weights to decrease or increase too much.

Explanation

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

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{l})^2 = np.sum(np.multiply(w^{[l]}, w^{[l]}))$$

= For i = 1 to L :

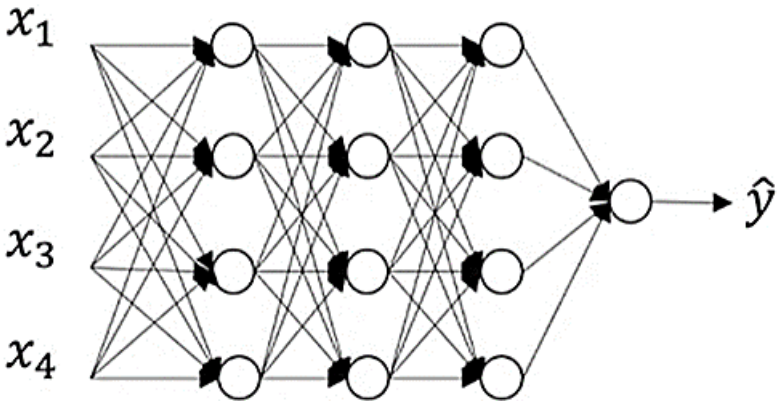
$$\text{Sum} \left(\begin{bmatrix} w_{11}^{[l]2} & w_{12}^{[l]2} & \cdots & w_{1n^{[l-1]}}^{[l]2} \\ w_{21}^{[l]2} & w_{22}^{[l]2} & \cdots & w_{2n^{[l-1]}}^{[l]2} \\ w_{31}^{[l]2} & w_{32}^{[l]2} & \cdots & w_{3n^{[l-1]}}^{[l]2} \\ \vdots & \vdots & & \vdots \\ w_{n^{[l]}1}^{[l]2} & w_{n^{[l]}2}^{[l]2} & \cdots & w_{n^{[l]}n^{[l-1]}}^{[l]2} \end{bmatrix} \right)_{n^{[l]} \times n^{[l-1]}}$$

$$dw^{[l]} = \frac{dL}{dw^{[l]}} = \frac{dL}{dz^{[l]}} \times \frac{dz^{[l]}}{dw^{[l]}} + \frac{d}{dw^{[l]}}\left(\frac{\lambda}{2m}\sum_{i=1}^{m} \|w^{[l]}\|^2\right)$$

$$= dz^{[l]} \times A^{[l-1]} + \frac{\lambda}{m}\|w^{[l]}\|$$

$$\left\{ \begin{aligned} dw^{[l]} &= \frac{dL}{dz^{[l]}} \times \frac{dz^{[l]}}{dw^{[l]}} \\ \frac{dz^{[l]}}{dw^{[l]}} &= \frac{d}{dw^{[l]}}\left(w^{[l]}A^{[l-1]} + b^{[l]}\right) \\ &= A^{[l-1]} \end{aligned} \right\}$$ For non regularized

Dropout regularization



In dropout regularization we randomly eliminate some neurons from different layers in the network in order to get simpler network.

How is it done?

Let's say we are applying dropout to layer l.

```
Keep_prob = x
```

```
db = np.random.rand(a[l].shape[0], a[l].shape[1])> Keep_prob
```

```
db would be matrix with random True,False values
```

```
a[l] = np.multiply(a[l],db)
```

```
a[l]/= keep_prob
```

How does it work?

Let's say `x = 0.5, and a[l] is of size 3,3`

```
Let a[l] =
```

```
[[1, 2, 5],
 [4, 5, 8],
 [8, 9, 5]]
```

```
db = np.random.rand(a[l].shape[0], a[l].shape[1])> Keep_prob
```

This would create matrix db as follows:

```
[[False False False]
 [ True  True  True]
 [ True  True False]]
```

```
a[l] = np.multiply(a[l],db)
```

After this $a^{[l]}$ =

```
[[0, 0, 0],
 [4, 5, 8],
 [8, 9, 0]]
```

We can see that half the values are turn to zero. Now as we know this would mean that when we multiply $a^{[l]}$ with W and find Z it would be a smaller value than previous $a^{[l]}$ so in order to counter this loss we divide $a^{[l]}$ by the value we use to set probability keep_prob

$a^{[l]}$ `/= keep_prob`

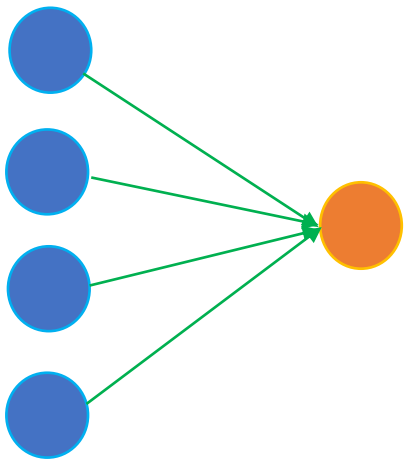
So $a^{[l]}$ would now look like:

```
[[ 0.,  0.,  0.],
 [ 8., 10., 16.],
 [16., 18.,  0.]]
```

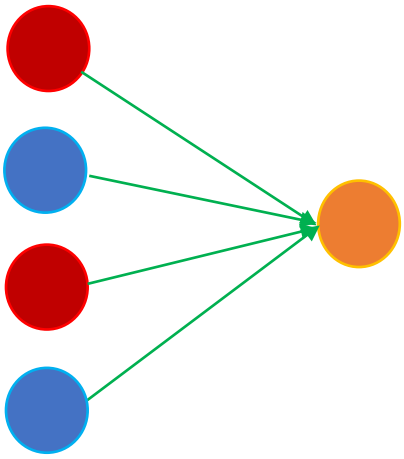
Which is scaled up to counter the loss

Why does it work?

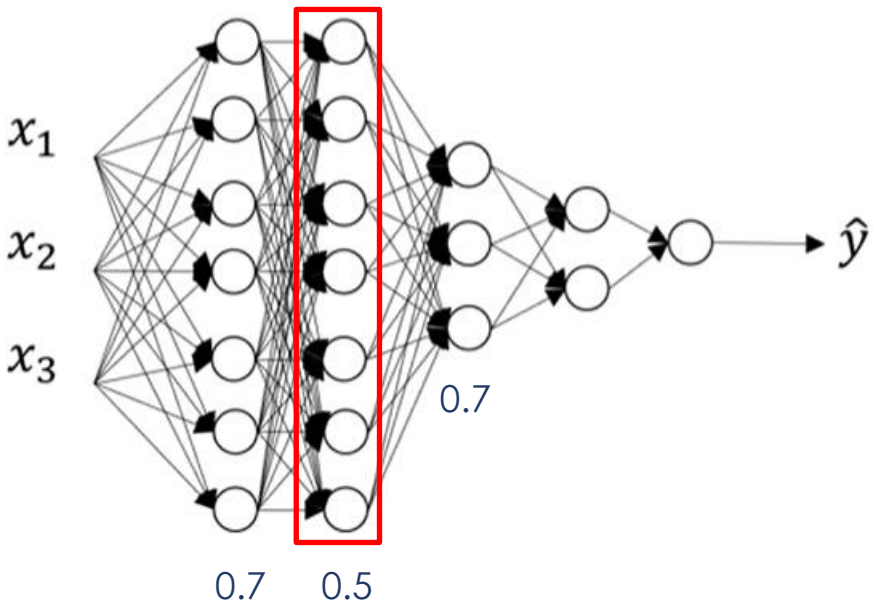
Consider a simple neural network



Let's say we apply dropout to eliminate few of these nodes



Now what happens is a we eliminate few neurons the weight assigned for each node would become smaller as it can't rely on any particular node more as any node can be eliminated. Therefor the overall norm of weights would be less. Therefor it reduces overfitting as it reduces weights distributed little bit.



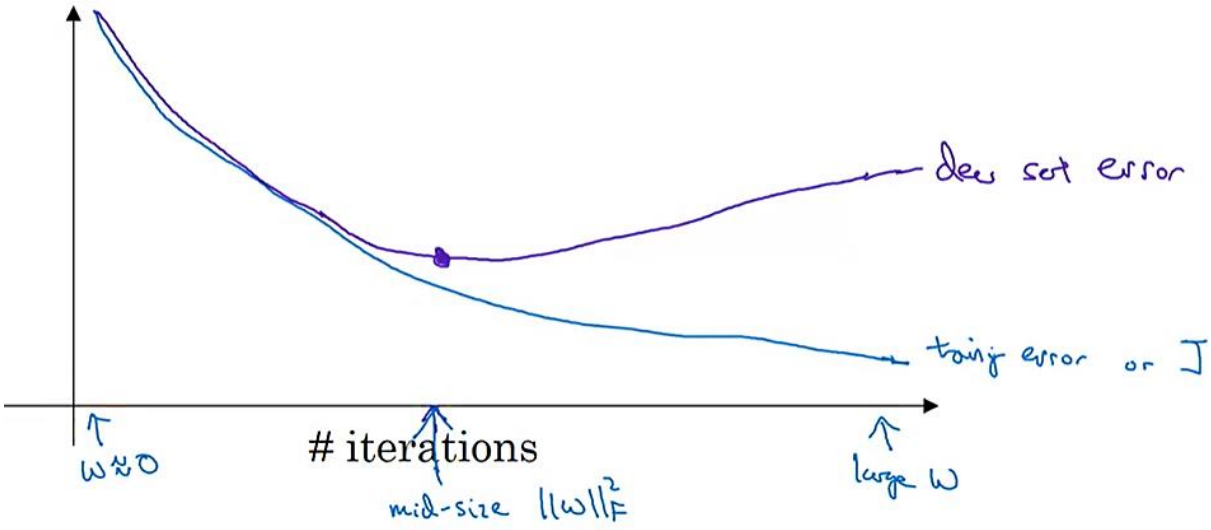
If we consider this neural network, we can see that 2nd layer is layer with most networks and is more prone to overfitting, therefore we apply less value of keep_prob for that layer and more for others.

Other techniques to reduce overfitting

Data Augmentation

One way to reduce overfitting is to collect more data. But most of the times it's a costly process. So, one thing we can do is to take our data and make few changes like the orientation or dimension or maybe distort if our data is set of images and use it as additional data.

Early stopping



In early stopping we will be plotting graphs of cost function for dev set and training set. And after few iterations we would observe the dev set error stop reducing and going up. We would stop the training at a point were both training set and dev set are at their best low point. This will effectively reduce the weights as the weights would only keep increasing as the number of iterations increase and by stopping it in middle, we would reduce it from increasing very high and would do similar job as L2 regularization.

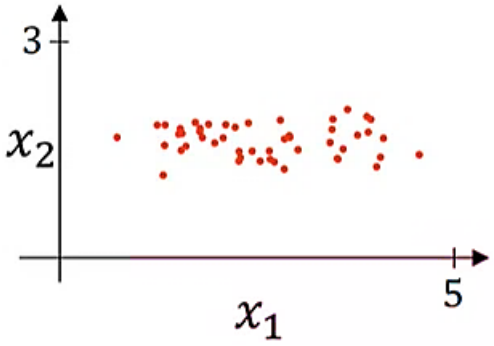
Normalizing inputs

What?

Normalizing is the process of making all the input features to a similar scale. That is to have similar value of mean and standard deviation for all the features.

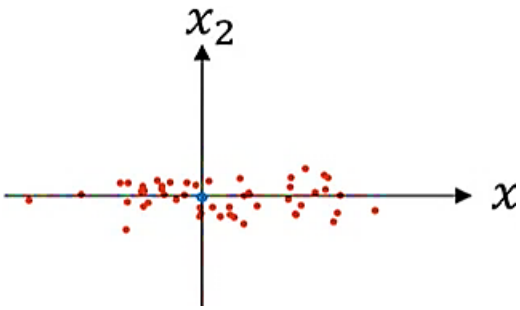
How?

Let's take an example of data with different scales for different features.



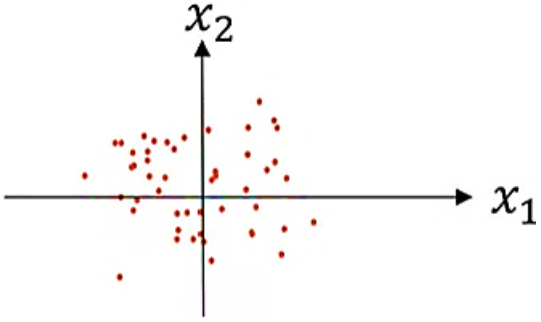
Here we can see that it doesn't have a common mean and the deviation for x1 is larger than x2.

1st we would make the mean of data to 0.



$$x := x - \mu$$

2nd we would make the standard deviation equal by dividing with std.



$$x := x|\sigma$$

Now we can see that our both features have same mean and variance.

Steps:

```
u = (1/m)*np.sum(x)
x = x-u
var = (1/m)*np.sum(np.power(x,2))
std = np.sqrt(var)
x = x/std
```

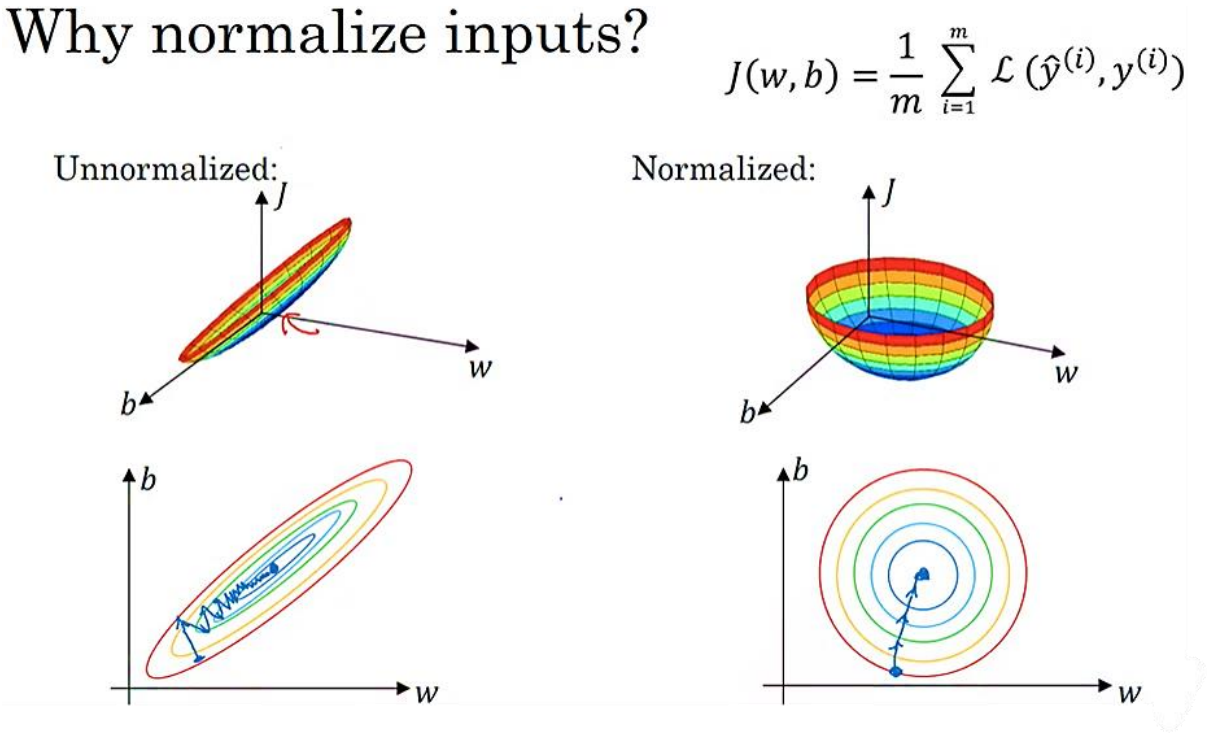
Why?

Usually when we collect data for a particular purpose it might have different features with different scales. For example, weight of a person might have scale from 20-100 and his salary might have scale from 100-1000000. So, both these features are having different might contribute more towards the activation functions final output than the other feature. So, our model might be predicting the results based on effectively fewer features than what's given.

Let's again consider the example above where we want to find out whether a person is going to buy an exercise machine which is

costly based on his weight and salary. So, as we know the prediction should depend on both the weight and the salary of the person. But since the salary has a higher scale of values our model might trend to give that feature more importance than the weight. So, we might end up getting wrong predictions for people with less weight and more money or vice versa.

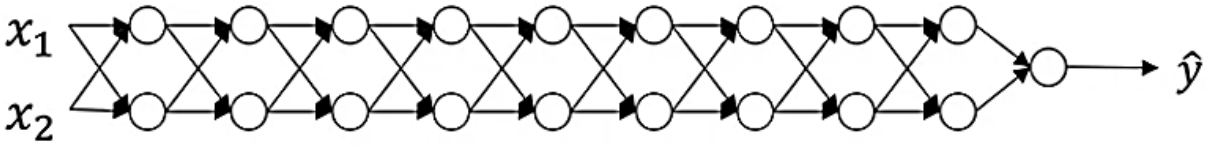
Also, another factor is that normalisation helps gradient descent easier.



We can see the when we normalize the cost function graph would look more symmetric and therefore gradient decent would be much faster as we can use bigger values of learning rate.

Vanishing/Exploding gradient descent

When we train deep neural networks like the one above, the gradient descent might get exponentially high and we call it exploding gradient descent. When the it gets exponentially small, we call it vanishing gradient descent.



Let the weight for each layer be $W^{[1]}, W^{[2]}, W^{[3]}, \ldots, W^{[L]}$. Lest take b = 0 here. And let the activation function $g(x) = z$ then we have:

$$\hat{Y} = W^{[L]} A^{[L-1]}$$

$$\downarrow$$

$$Z^{[L-1]} = W^{[L-1]} A^{[L-2]}$$

$$\downarrow$$

$$Z^{[L-2]} = W^{[L-2]} A^{[L-3]}$$

$$\vdots$$

$$\downarrow$$

$$\vdots$$

$$\hat{Y} = W^{[L]} W^{[L-1]} W^{[L-2]} \ldots\ldots W^{[2]} W^{[1]} X$$

Let $W^{[l]} = \begin{bmatrix} x & 0 \\ 0 & x \end{bmatrix}$, then $\hat{Y} = W^{[L]} \begin{bmatrix} x & 0 \\ 0 & x \end{bmatrix}^{L-1}$

$W^{[l]}$ has different dimension

1. IF $x > 1$, then $\hat{Y} \approx 1.5^{L-1} X \quad (x = 1.5)$

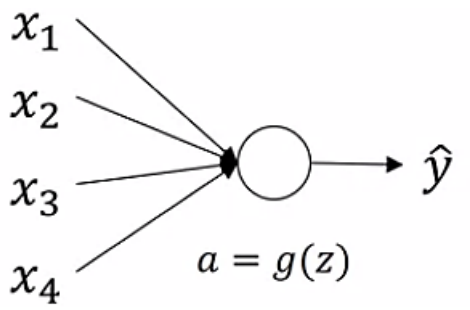
2. FOR $x < 1$, $x = 0.5$ $\hat{Y} \approx 0.5^{L-1} X$

As we can see if the values for weight is greater than 1 the output value increases exponentially or if weights are less than 1 then it decreases exponentially. Similarly, we can show that the gradients increase and decrease exponentially as we change the weights. This can make the learning very difficult as a very less value means the gradient descent would take very small steps and if its too large then gradient descent wouldn't converge to local minima. To reduce this, we have to initialize weights carefully.

Weight Initialization for Deep Learning Networks

Here we would see a way in which we can initialize weights so that we can reduce the vanishing/exploding problem.

Let's consider a single neuron neural network for this.



Here $\boldsymbol{z = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n}$

So, in order to reduce the problem, the weights should be smaller for larger number of features since z is the sum of all these terms, i.e. when n increases the weight values should decrease. One way to achieve this is to set the variance of the randomly initialized values of weights to be neither too large than 1 nor too smaller than one. So, we would set **var(w) = 1/n**. So that when n increases the values of weights would be smaller.

In general, we set:

$W^{[l]}$ = `np.random.randn(`$n^{[l]}, n^{[l-1]}$`) * np.sqrt(1/`$n^{[l-1]}$`)`

When $g(z)$ = ReLU we would use $2/n^{[l-1]}$ as variance.

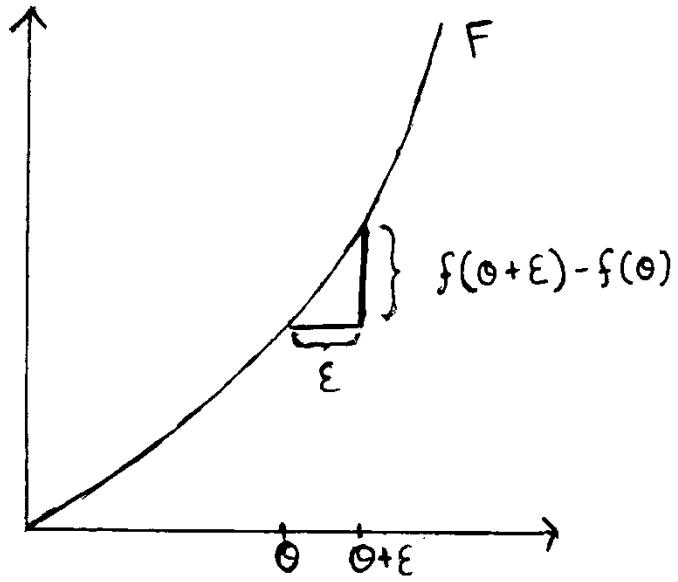
Methods for gradient checking

There are two methods to find gradient of a function:

$\frac{\boldsymbol{f(\theta+\varepsilon)-f(\theta)}}{\boldsymbol{\varepsilon}}$ **&** $\frac{\boldsymbol{f(\theta+\varepsilon)-f(\theta-\varepsilon)}}{\boldsymbol{2\varepsilon}}$

Let's see which one performs better in finding gradient by taking an example function $f(\theta) = \theta^3$ and take the positive side of the graph to find gradient.

1st finding gradient with one triangle.



$f(\theta) = \theta^3$

$g(\theta) = f'(\theta) = 3\theta^2$

$$f'(\theta) = \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$ // From the graph

Let $\theta = 1$ & $\varepsilon = 0.01$

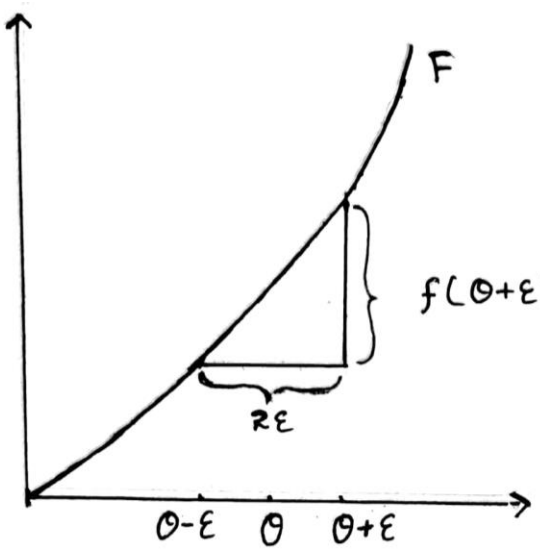
then $\theta + \varepsilon = 1.01$, $f(1) = 1$, $f(1.01) = 1.030301$

$$\therefore f'(\theta) = \frac{1.030301 - 1}{0.01} = 3.0301 \quad // \theta = 1$$

$$g(\theta) = 3 \times 1^2 = 3 \quad // \theta = 1$$

$$f'(\theta) - g(\theta) = 0.0301 \quad -①$$

Here we can see that the error in finding gradient is 0.0301 when we use triangle with $\boldsymbol{\varepsilon}$ width. Next, we would find gradient using triangle with width $2\boldsymbol{\varepsilon}$.



$f(\theta) = \theta^3$

$g(\theta) = 3\theta^2$

$g(\theta) = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ From graph

Let $\theta = 1$ & $\varepsilon = 0.01$

then $\theta + \varepsilon = 1.01$, $\theta - \varepsilon = 0.99$, $f(1.01) = 1.030301$,

$f(0.99) = 0.970299$

$\therefore f'(1) = \frac{1.030301 - 0.970299}{2 \times 0.01} = 3.0001$

$g(1) = 3$

$f'(1) - g(1) = 0.0001$ – ②

From the two pictures we can see that finding gradient using triangle with width 2ε is much more accurate than the other.

Gradient checking

How it works?

Take $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ.

Take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

Then $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots, W^{[L]}, b^{[L]})$ would be equal to $J(\theta_1, \theta_2, \theta_3, , , \theta_L) = J(\theta)$

Let $\varepsilon = 10^{-7}$

```
For i = 1
```

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \theta_3, \ldots \theta_i + \varepsilon, \ldots \theta_L) - J(\theta_1, \theta_2, \ldots \theta_i - \varepsilon, \ldots, \theta_L)}{2\varepsilon}$$

The value of $d\theta_{approx}$ should be ≈ to $d\theta$ to check this we do:

Check

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$$

For $\varepsilon = 10^{-7}$ if the difference value we got = 10^{-7} then gradients are all correct.

If it is less than or equal to about 10^{-3} then we need to check for errors in calculation of gradients.

Gradient checking implementation notes

- Don't use in training – only to debug

- If algorithm fails grad check, look at components to try to identify bug.

- Remember regularization.

- Doesn't work with dropout.

- Run at random initialization; perhaps again after some training.

WEEK 2 - Optimization Algorithms

Mini Batch Gradient Descent

Assumption: The dataset is large (BIG Data)

In mini batch gradient descent, we split our dataset of m=5,000,000 training samples into different batches containing let's say 1000 examples each. We use the term $X^{\{t\}}$ to represent one batch. So, with 5,000,000 training samples we would have 5000 batches each having 1000 training samples.

$X^{\{t\}}$ $(n_x, 1000)$, $Y^{\{t\}}$ $(1, 1000)$

Repeat

{

For $t = 1$ to 5000

{

Forward Propagation

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$\vdots$$

$$A^{[L]} = g^{[L]}(Z^{[L]})$$

$$J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \times 1000} \sum_{l=1}^{L} \| W^{[l]} \|_F^2$$

backward propagation

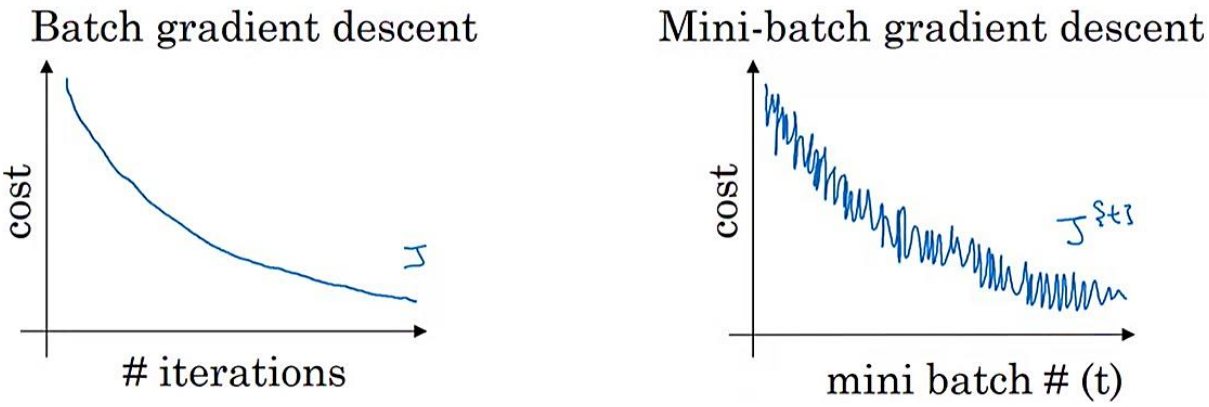
$$W^{[l]} := W^{[l]} - \alpha \, dW^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha \, db^{[l]}$$

}

}

We can see that by dividing the training samples into different batches we can now run gradient descent on a batch of training samples multiple times. Means after one epoch (one iteration) we have already trained the weights 5000 times using small batches of the training samples. Whereas in batch gradient descent after one epoch we have only changed the weights once.

Cost function curve



The oscillations in the curve in the case of mini-batch gradient descent is due to the different costs for different batches. But it would also trend down following a similar curve.

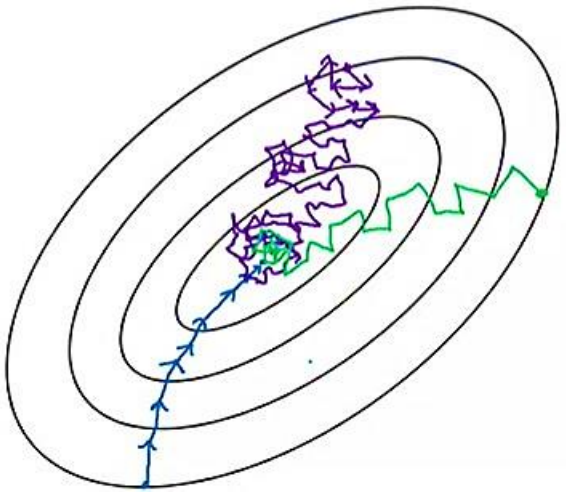
Types of batch gradient descent

If size of batch is equal to m then it is batch gradient descent. The whole training sample is taken as a batch.

If we take each batch as one training sample from the training set the it is called Stochastic gradient descent.

If we take batch size as anywhere between 1 and m then it is mini batch gradient descent.

These three gradient descent methods have difference in the way in which they converge towards the minima.



In batch gradient descent the cost reduces gradually and converge to the minima in time.

In stochastic gradient descent the cost oscillates few times while moving towards the minima and keep on circling the minima while never converging completely

In mini batch gradient descent, the cost oscillates while moving towards the minima and it would come really close to the minima but never completely converge. These oscillations can be reduced by changing the learning rate.

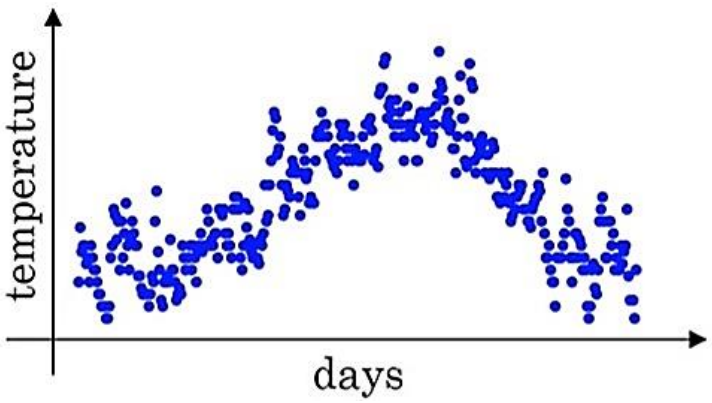
Choosing mini-batch size

- If size of training data is $\boldsymbol{m \leq 2000}$ then do batch gradient descent. If it Is greater than that then choose mini-batch.
- When we choose the size for batch choose the size as powers of 2. For example, like **64, 128, 512, 1024** etc.
- Also, when we choose size of the batch, we have to make sure that it fits in the CPU memory.

Exponentially weighted averages

Let's take an example f the temperatures in London for a year.

$\theta_1 = 40°F$
$\theta_2 = 49°F$
$\theta_3 = 45°F$
$\vdots$
$\theta_{180} = 60°F$
$\theta_{181} = 56°F$
$\vdots$



So, if we need to find the trend of this data by calculating the moving average in a particular window, we would do like this.

V_0 = temperature of day 1

$V_1 = \beta V_0 + (1-\beta)\theta_1$

$V_2 = \beta V_1 + (1-\beta)\theta_2$

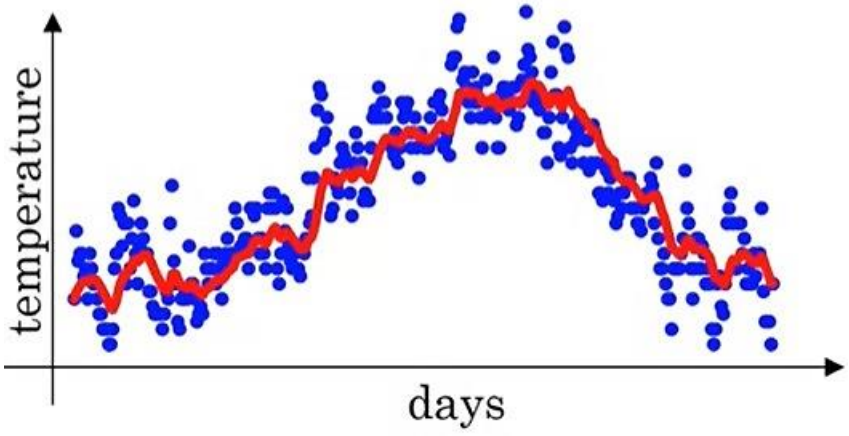
$\vdots$

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

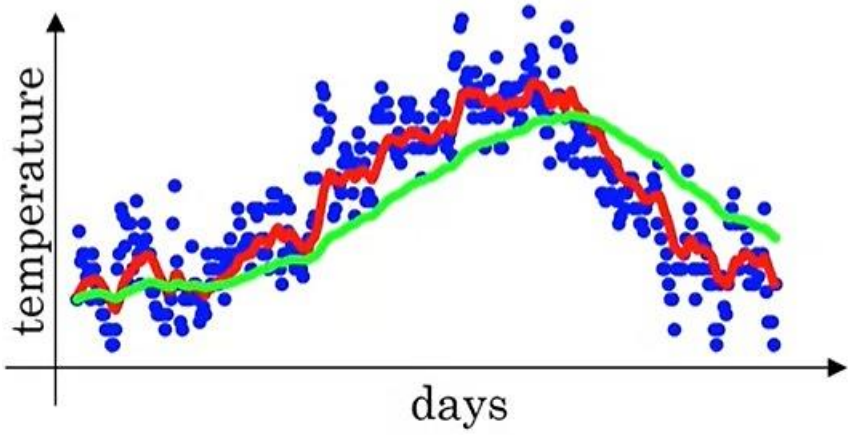
θ_t is the original temperature of the day t

Where v_t is average over $\frac{1}{1-\beta}$ days of temperature.

When β = 0.9 we are averaging over 10 days of temperature. The graph would look like this:

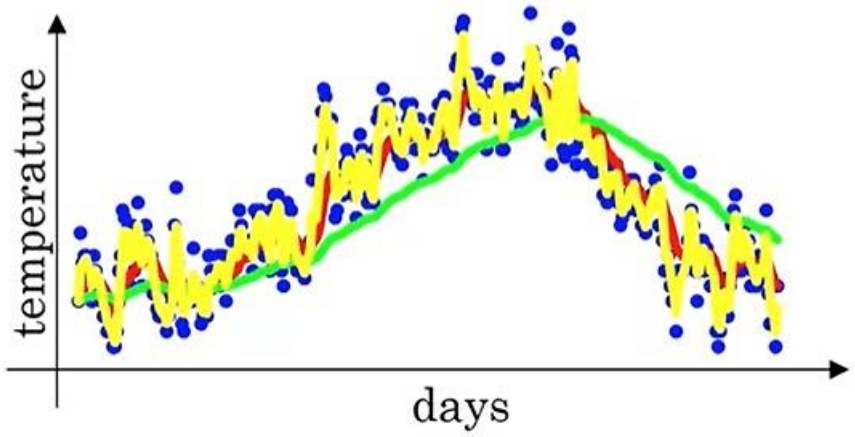


When β = 0.98 we are averaging over 50 days of temperature. The graph would look like this:



The change of the green line and its squeezed towards right is because we are giving more weight to the past day's temperature than the current days one. So, the average temperature would be little late to adapt to the new changes.

When β = 0.5 we are averaging over 2 days of temperature. The graph would look like this:



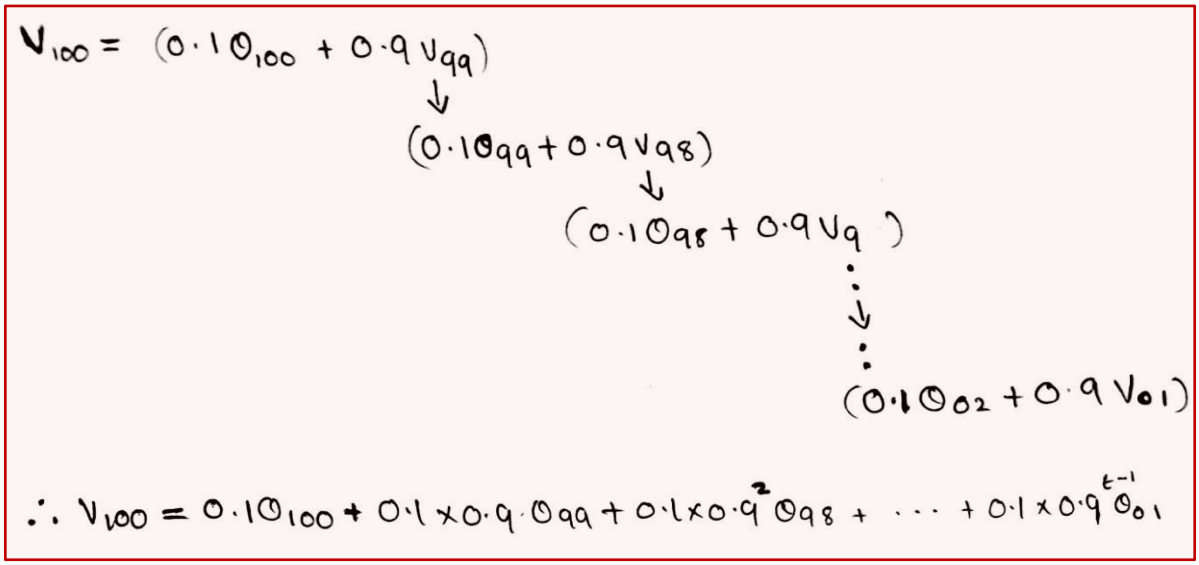
We can see that this is not a great way to find the trend as it can also include some outliers which wouldn't help in predicting the trend. So, we can say in this case the red line looks more better choice.

Explanation & implementation

Consider:

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$



The number of days in the window to find the moving average is calculated as 1/1- β. So, for β = 0.9 no of days = 10. The reason is because when calculating the value of v_{100} above in the example we have to calculate the weighted sum of previous days temperatures. So, when the weight reaches $0 \cdot 9^{10}$ it is a small number almost equal to 0.35 which is approximately 1/e. For a window of n days the value of β is calculated as (n-1)/n.

Implementation

$V_\theta := 0$

$V_\theta := \beta V + (1-\beta)\theta_1$

$V_\theta := \beta V + (1-\beta)\theta_2$

⋮

Generally:

$V_\theta := 0$

Repeat {

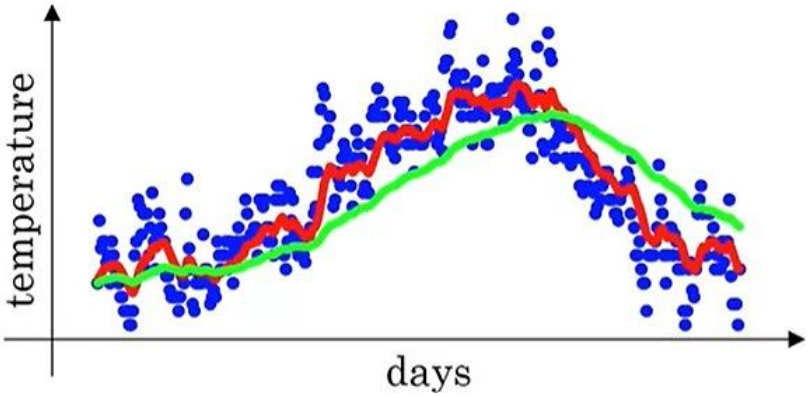
get next θ_t

$V_\theta := \beta V_\theta + (1-\beta)\theta_t$

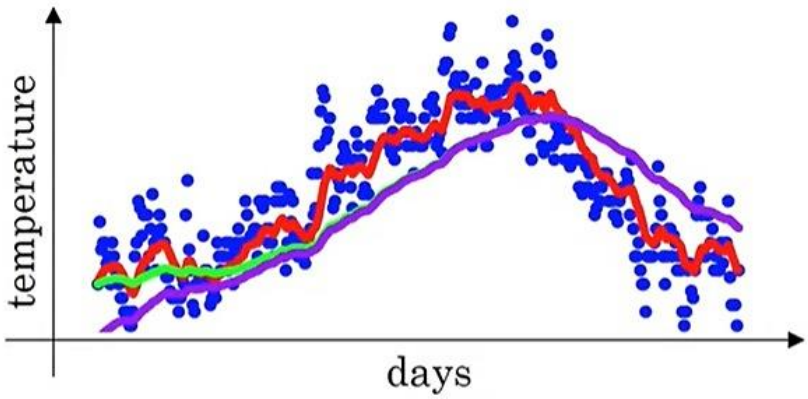
}

Bias Correction

We know that for β = **0.9** and **0.98** we get lines like these



But actually, without bias correction the line we would get for β = **0.98** is given below as violet coloured line:



We can see that the line starts well below the actual average value is supposed to be. The reason is:

$V_0 = 0$

$V_1 = 0.98 V_0 + 0.02\theta_1$

$= 0 + 0.02\theta_1$

So if θ_1 = lets say 40°c V_1 would be 0.8°c

Similarly

$V_2 = 0.98 \times 0.02\theta_1 + 0.02\theta_2$

$= 0.01960_1 + 0.02\theta_2$

which would again be a small value

This is the reason why we see that very low starting point for the line in the graph.

To solve this we find $\frac{V_t}{1-\beta^t}$ instead of V_t

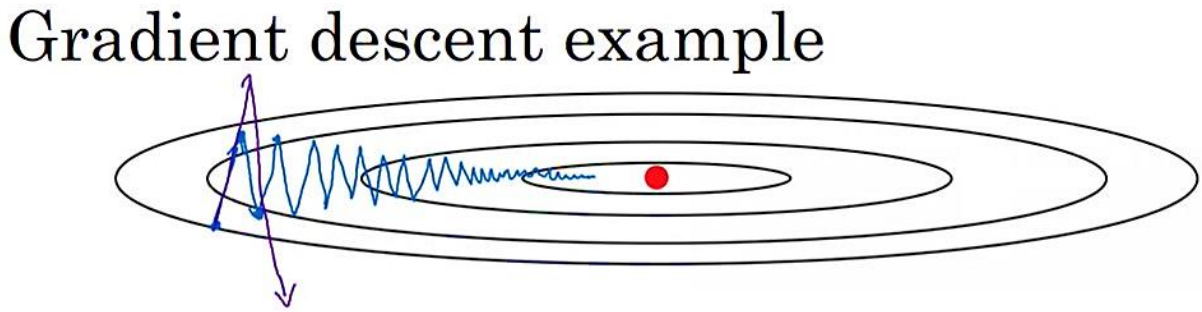
So for $t=2$, $1-\beta^2 = 1-(0.98)^2 = 0.0396$

$$\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

Here we can see that $0.0196 + 0.02 = 0.0396$ ∴ we are actually taking the weighted average of θ_1 and θ_2 hence removing the biase

Also we can see that as t increases β^t decreases ∴ reducing its effect on the values as t increases

Gradient Descent with Momentum



When we do mini-batch or batch gradient descent on our data, the cost function reduces to minima as shown above in blue line. We can see that it **oscillates** up and down while moving towards the minima. This would increase the learning time and is inefficient. And if we increase the learning rate more it can go out of the path and start to **shoot up and down** as shown in purple line.

To reduce this, we can average out the positive and negative values in the vertical direction, thereby reducing the oscillations. It won't affect the learning in horizontal direction as the direction for all the values are the same.

So, we update our gradient descent steps like this:

$v_{dw=0}$

$v_{db=0}$

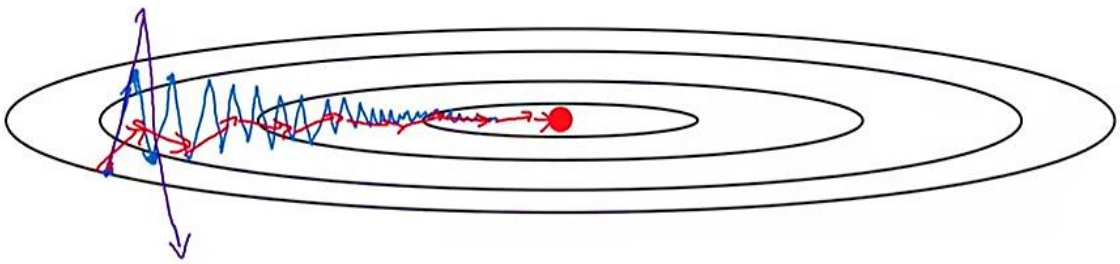
On iteration t:

Compute dW, db on the current mini-batch

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

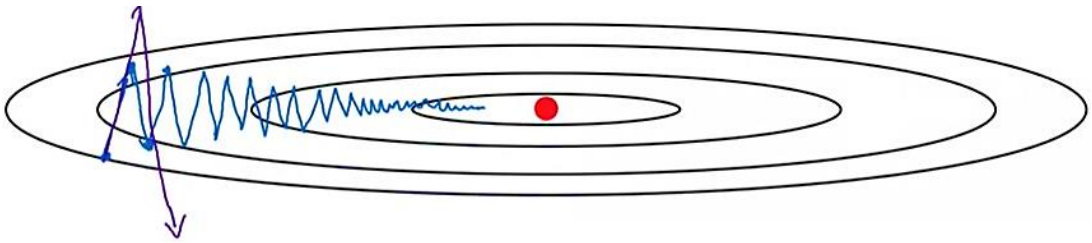
$v_{db} = \beta v_{db} + (1-\beta)db$

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



So, by applying exponentially weighted moving average to our gradient descent steps we have averaged out the oscillations to make our learning faster. Also, we can use β too as a hyperparameter. But usually setting β = 0.9 works mostly fine.

RMSprop (Root Mean Square Propagation)



This is another way to reduce the oscillations and making the learning faster. Here we update our terms for gradient descent like this:

First let's imagine that the term that makes the oscillations on vertical axis is b and the horizontal is w. In normal cases it might be combinations of lots of features. So, we have to increase w and decrease b.

On iteration t:

Compute dw, db on current mini batch

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

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

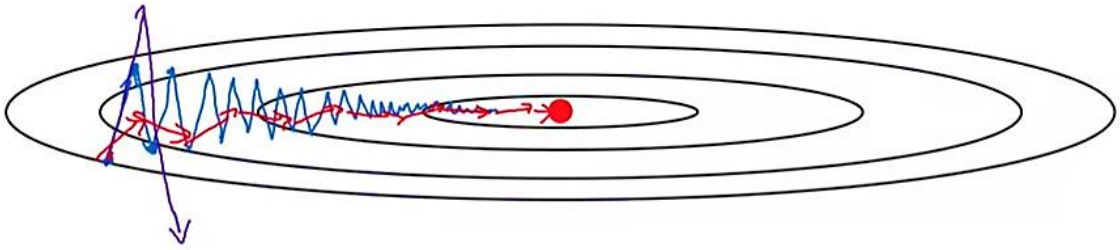
$$W := W - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon} \quad , \quad b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

Where ε is just a small term to make sure that the denominator is never zero.

Let's imagine that b is in the horizontal axis and w is in the vertical axis. So, if we need to reduce the oscillations, we need to change

the value of b slowly, for this we have to subtract it with smaller values. Here we can see that we are finding the exponential average using the square of the derivative term. So that if the slope is larger its square would be much larger. If it is smaller the square would be much smaller. In our case the slope for b is large. So db would be large. So db when squared would be larger. Then we are dividing the derivative term which we are going to subtract from b using the weighted average term which is large thereby reducing the value of b slowly.

On the other hand, since w is small the value of weighted average would be small too therefore when we divide the derivative with smaller value, if it is smaller than 1 it would increase the derivative term therefore getting larger changes and faster learning horizontally.



Now we would see an algorithm that combines the use of gradient momentum and RMSprop together for better optimization,

Adam (Adaptive Moment Estimation) Optimization

Adam optimization combines both momentum and RMSprop algorithms to optimize the gradient descent. It is implemented as follows:

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1-\beta_1)\dfrac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v_{dW^{[l]}}^{corrected} = \dfrac{v_{dW^{[l]}}}{1-(\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1-\beta_2)(\dfrac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s_{dW^{[l]}}^{corrected} = \dfrac{s_{dW^{[l]}}}{1-(\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \dfrac{v_{dW^{[l]}}^{corrected}}{\sqrt{s_{dW^{[l]}}^{corrected}}+\varepsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

Hyperparameter choices:

α – needs to be tuned

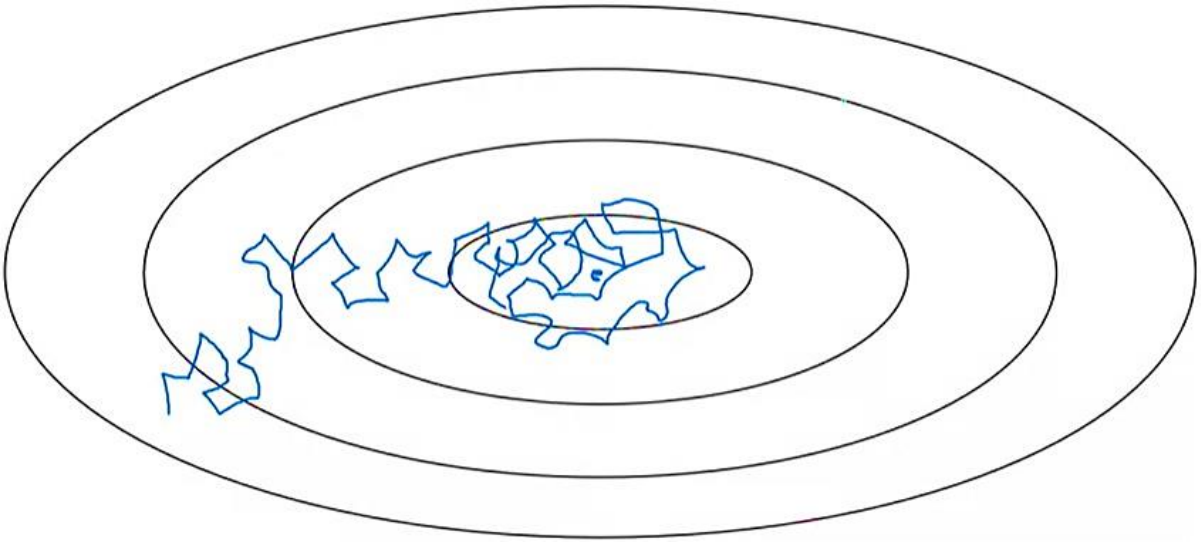
β_1 = 0.9

β_2 = 0.999

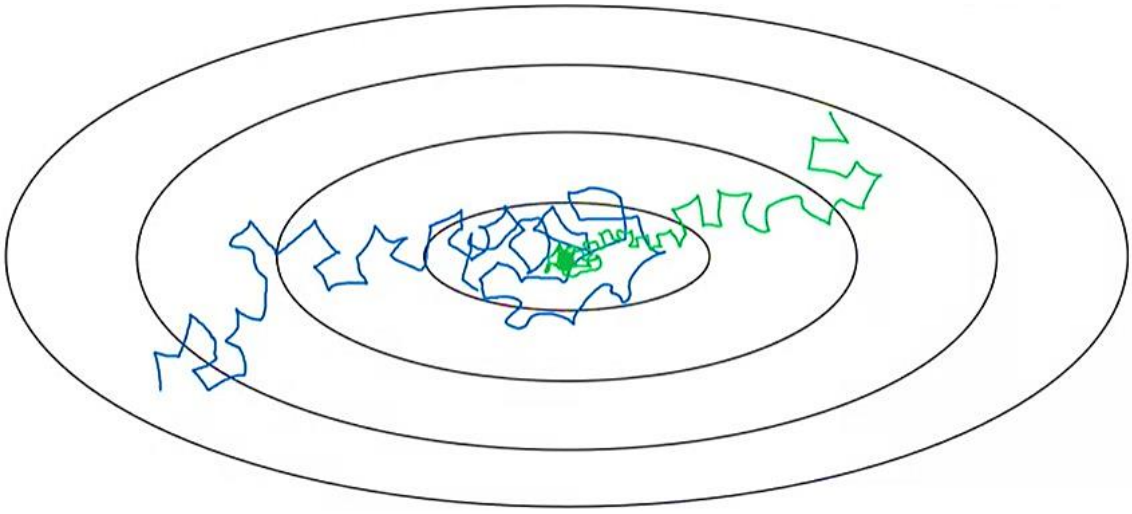
$\varepsilon = 10^{-8}$

Learning Rate Decay

Suppose we are doing gradient descent on a small mini-batch of training set. Then the steps would tend to be little bit noisy and it would go down towards the minima but it wouldn't converge to it and instead **it would revolve around a larger region around the minima**.

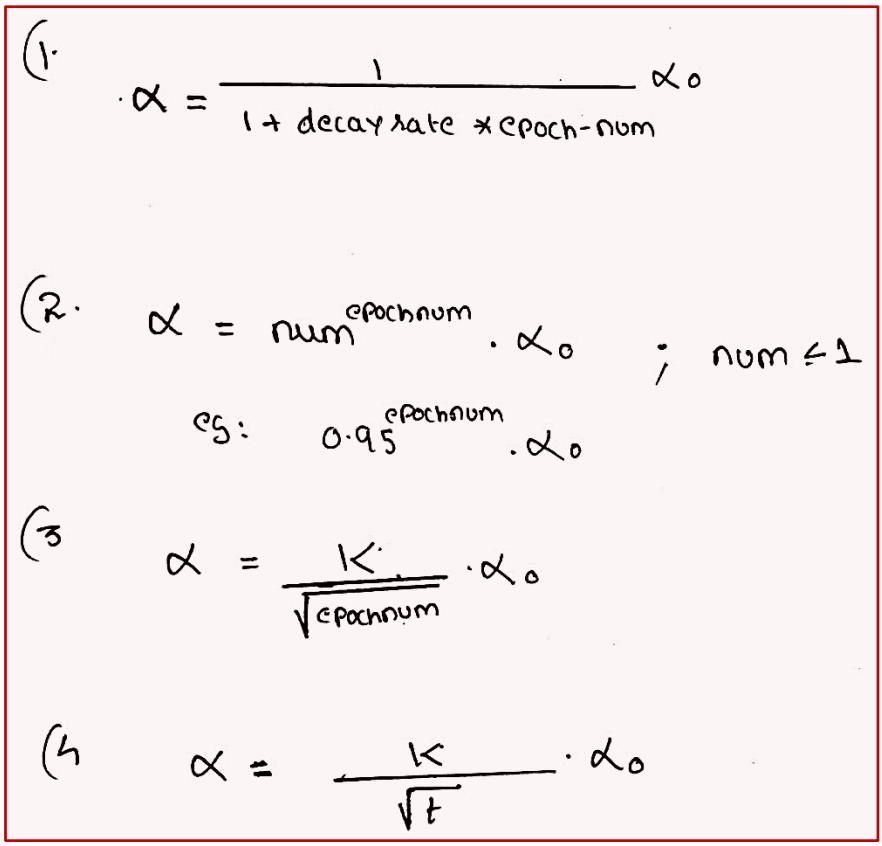


So, one thing we can do to get it closer to the minima is to reduce the learning rate as the number of epochs increases. So, for **smaller learning rate** the steps we take would be slower and smaller and the **steps would revolve around a much smaller region close to the minima**.



By doing this we can afford to have larger learning rate in the beginning of the training and as we get closer to the minima, we can reduce the learning rate to get better convergence.

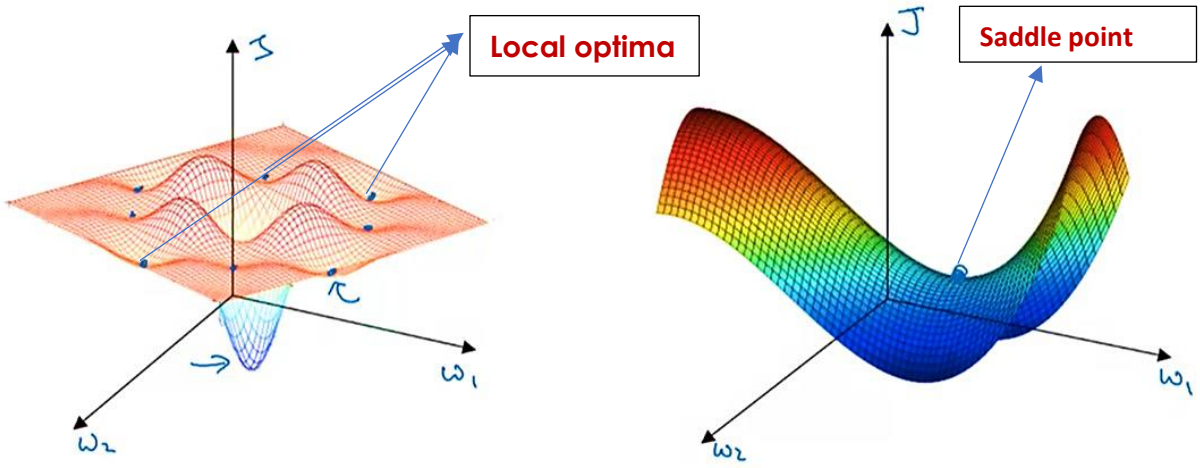
The different commonly used to ways to do this are:



We can try out different values for α_0 , decay rate, num and k and tune it as hyperparameters.

The problem of local optima

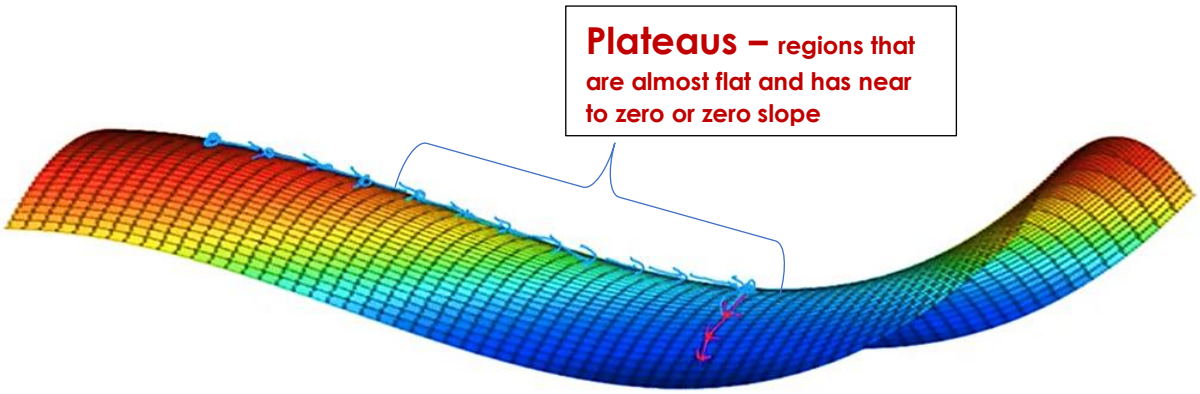
- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow



Getting stuck at local optima is problem that occurs mostly when we have dimensions like 2 or 3 for the training data features. It

usually won't happen for a training set with lots of dimensions. Instead there can be saddle points where derivative would be zero.

But one problem that can happen is because of plateaus. These are regions which are almost flat and has almost zero slope. So, it would take long time to get from these regions to global optima.



This problem can be reduced by using the optimization algorithms that we learned about.

WEEK 3 – Hyperparameter Tuning

Hyperparameters that are most important to be tuned:

α

β

$\beta_1, \beta_2, \varepsilon$

Layers

Hidden units

Learning rate decay

Mini-batch size

RED → Always tune

ORANGE→ 2nd Most important hyperparameter to tune

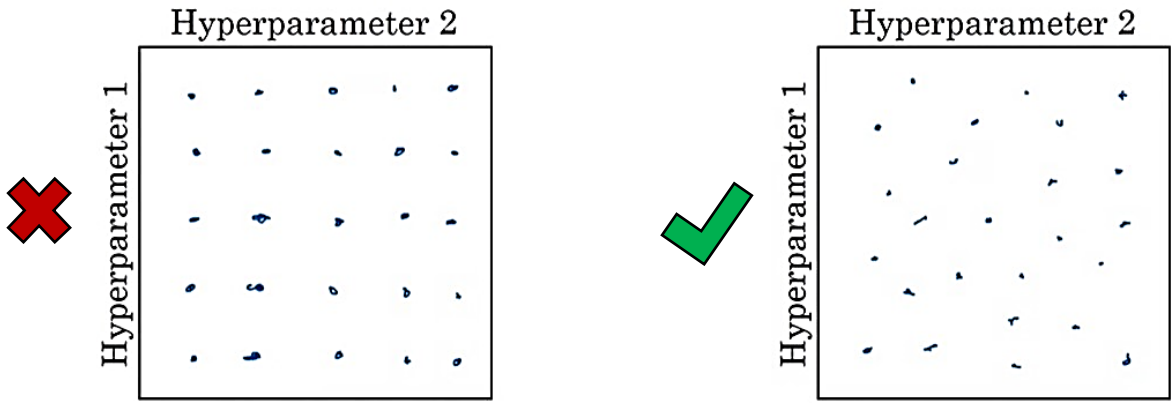
VIOLET→3rd Most Important

BLACK→ Usually not tuned $(0.9, 0.99, 10^{-8})$

Choosing parameters from different samples

In early days of deep learning people tend to choose parameters by placing them as samples in grid form and then trying out each of the pairs, as shown in the figure left.

Try random values: Don't use a grid

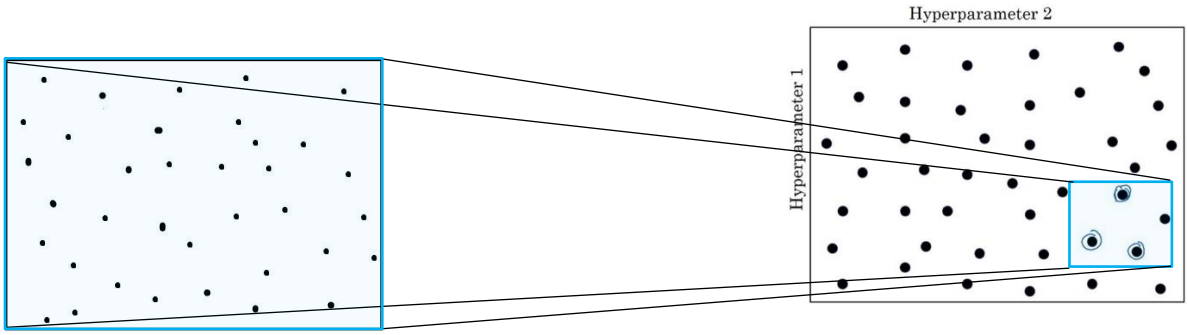


One disadvantage of putting hyperparameters in a grid for is that we would be able to test few different values of a particular hyperparameter even after we tests the entire grid. For example, if the grid contains 25 samples then each hyperparameter would be having 5 different values. So, we would only be seeing 5 different values after looking at 25 samples.

And for example, let's say hyperparameter 1 is learning rate alpha and hyperparameter 2 is epsilon. We know that alpha is more important than epsilon. So even after we are seeing 5 different values of alpha the value of epsilon might not be changing much as it has less importance.

So, when we test hyperparameters in sample we often place them randomly instead of in a grid format to get 25 unique samples with 25 unique values for each hyperparameters. In this case after testing all 25 samples we would be seeing 25 different values.

Another thing that works well in finding hyperparameter is **coarse to fine tuning**. We would focus on the area where more samples are giving good results and we take more samples in that region and test with it further.

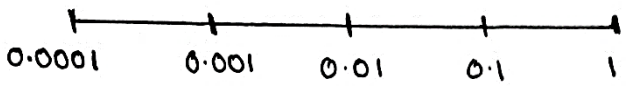


Using appropriate scale for Hyperparameter tuning

While searching for hyperparameters like $n^{[l]}$ or L we can use linear scale and choose values uniformly at random. Like for $n^{[l]}$ it can range from 50 to 100 and for L it can be around 2 to 4 or so. But when we are dealing with hyperparameters like α *or* β if we choose to tune uniformly at random in a linear scale then we would end up wasting lots of resources tuning a particular range closer to 1 more than the other range of values. So here we use logarithmic scale to tune.

Set $\alpha = 10^r$

Here we will be checking values in log scale



ie from $10^{-4} \rightarrow 10^0$

Generally we take values from 10^a to 10^b

here $a = \log_{10} 0.0001 = -4$, $b = \log_{10} 1 = 0$

$\therefore r = [a, b] = [-4, 0]$

$r = -4, \quad \alpha = 10^{-4} = 0.0001$

$r = 0, \quad \alpha = 10^0 = 1$

Set $\beta = 1 - 10^{r}$

β range from 0.9 to 0.999

$1-\beta$ range from 0.1 to 0.001

$= 10^{-1}$ to 10^{-3}

$\therefore$ r ranges from -1 to -3 $(a = \log_{10} 0.1, b = \log_{10} 0.001)$

when $r = -1$ $\beta = 1 - 10^{-1} = 0.9$

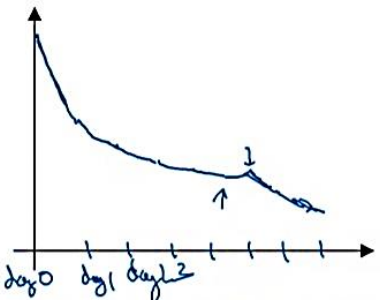
$r = -3$ $\beta = 1 - 10^{-3} = 0.999$

Hyperparameter choices can vary over time due to changes in datasets or changes in sources of data. Therefore, it is always good to retrain your model with the latest data time to time.

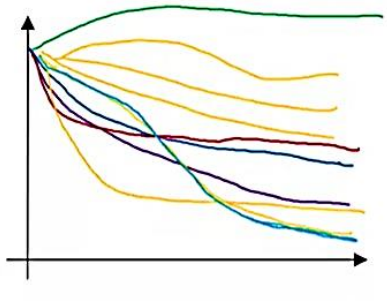
Two ways of training:

1. One model train over long time changing hyperparameters→If you have less computational power and has big dataset.
2. Multiple models with different hyperparameters trained overtime and choosing the best one→When you have lot of computational resources.

Babysitting one model

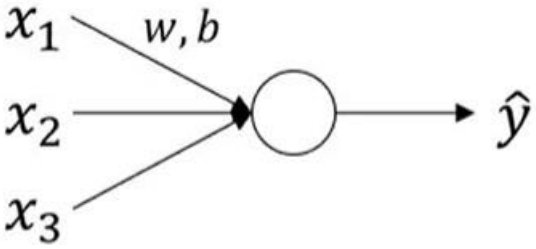


Training many models in parallel



Batch Normalization

We have seen that for neural network model that resembles logistic regression which doesn't have any hidden layers like the one shown below we normalize only the input features.



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

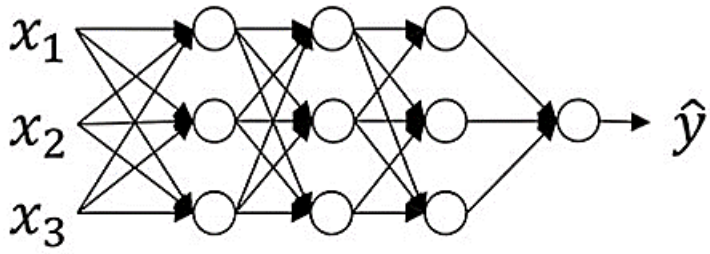
$$X = X - \mu$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$$

$$X = X / \sigma^2$$

This would transform the features to have similar mean and variance.

But in the case of neural net frameworks with hidden layers as shown below we have to normalize the hidden layer features too.



Here we need to normalize $a^{[l-1]}$ to improve the learning of $\mathrm{w}^{[l]}$ and $\mathrm{b}^{[l]}$. We normalize $z^{[l-1]}$ instead of $a^{[l-1]}$.

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

$$X = X - \mu$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$$

$$X = X / \sigma^2$$

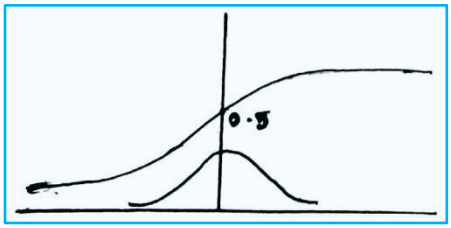
$$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{[l](i)}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (z^{[l](i)} - \mu)^2$$

$$z^{[l](i)}_{norm} = \frac{z^{[l](i)}_{norm} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

{ ε is used to make sure that denominator is never 0

But by doing this would set the mean and variance forcefully to 0 and 1. For example if we are using the activation function sigmoid then we would lose the advantage of the non linear functions capability to learn complex features because our features would be laying in a region closer to 0.



So if we don't want that to happen then we do the following:

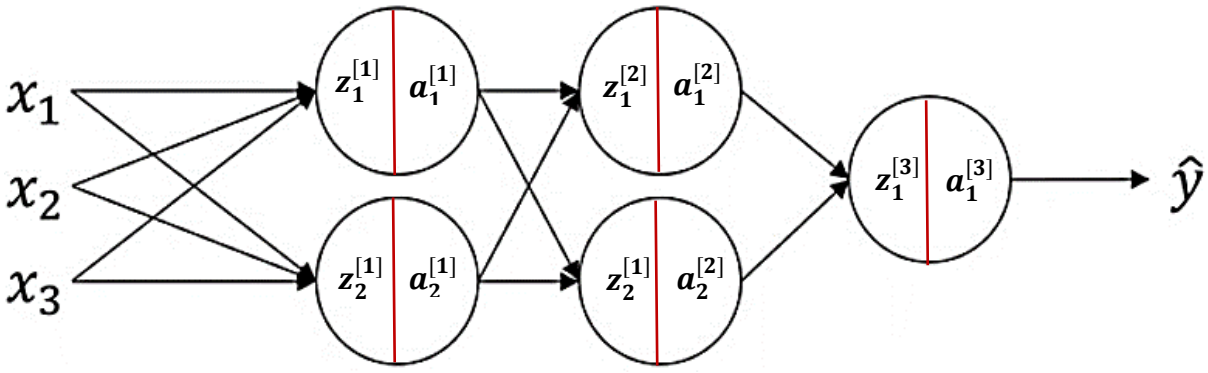
$$\tilde{z}^{[l](i)} = \gamma z^{[l](i)}_{norm} + \beta$$

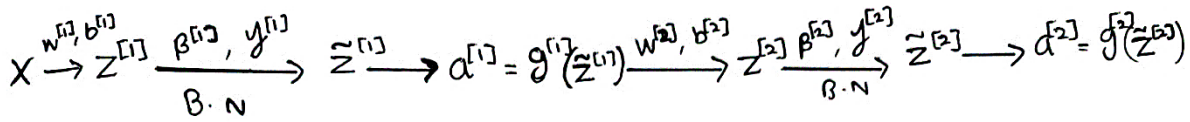
Where γ and β are learnable parameters which we would be learning using gradient descent maybe with some optimizers.

If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$ then $\tilde{z}^{[l](i)} = z^{[l](i)}$. Here in this case we are calculating the identity. But if we choose other values then we can normalize the hidden layer features with different mean and variances.

Fitting batch normalization to neural network

While training the model there are mainly two steps involved, forward propagation and backward propagation. We apply normalizations before we do forward propagation and we apply optimization before we do backward propagation. So, we can divide the forward propagation further into two halves, 1st finding the linear function Z and then finding the activation function a. We apply normalization to the z value we calculated. The whole process is depicted below.



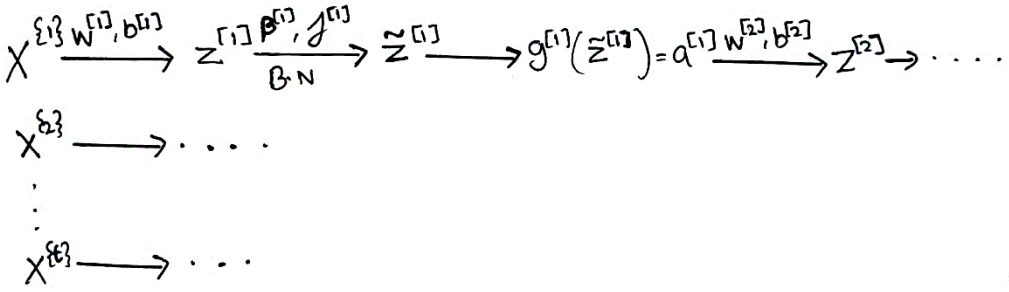


Parameters:

$$W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots, W^{[L]}, b^{[L]}$$

$$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots, \beta^{[L]}, \gamma^{[L]} \quad \{ dim = (n^{[l]}, 1)$$

In batch gradient descent



For t=1 to no of batches:

Forward prop on $X^{\{t\}}$

For each hidden layer use B.N to find $\tilde{Z}$ t then a.

Backprop to compute $dW^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

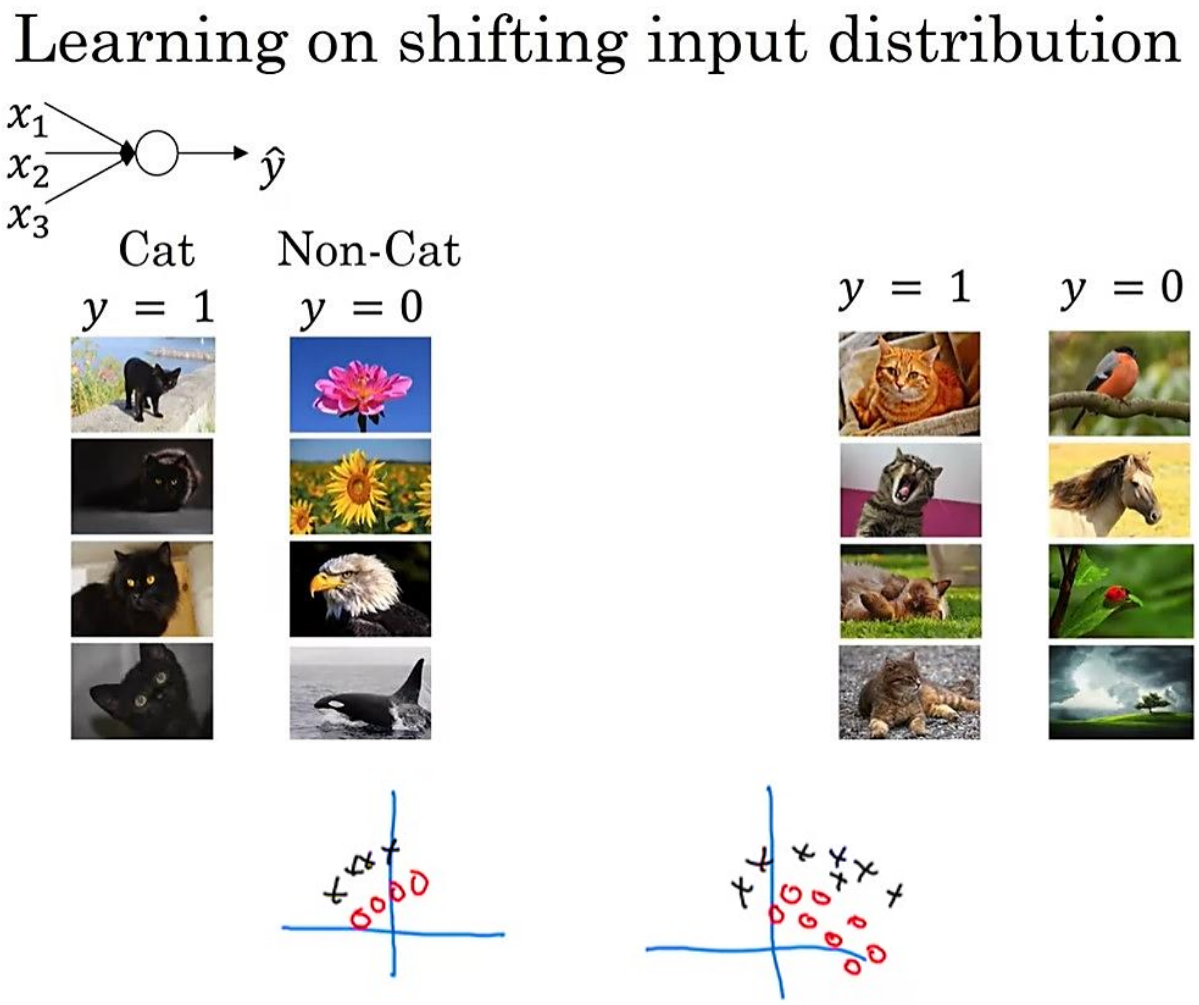
Update parameters $W^{[l]}$, $\beta^{[l]}$, $\gamma^{[l]}$

We omit b because since we are using batch normalization the term b would be cancelled out when we normalize the values.

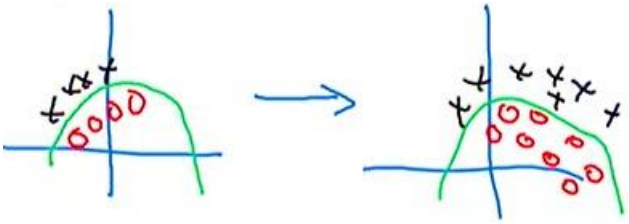
Why batch norm works

We know that normalizing our inputs would reduce the impact of high valued features on lower valued ones and it would also speed up the learning. But there are more reasons why we should do batch normalization and why it works.

Let's imagine we are training our deep learning model which classifies pictures to cat or non-cat. Our training dataset only contains black cats while we were training. In this case our model will not be performing well on a test set where the cat images are coloured.



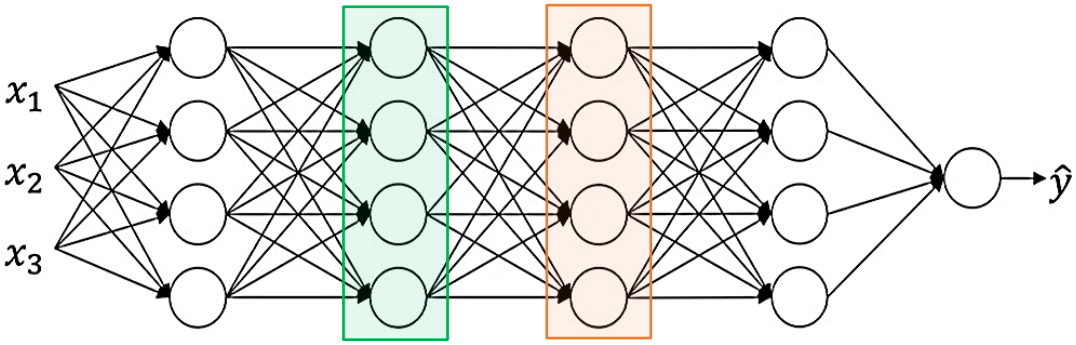
We can see that the distributions are totally different from each other. In this case we can't expect our model to it a line as shown below which can work well on both the distributions.



This problem of change in the distribution of input values is termed as covariate shift, where changes in distribution would result in

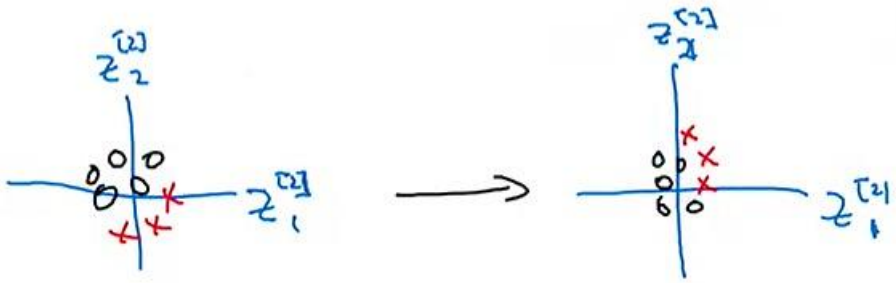
retraining the model. Also, if the ground truth function which maps the input to output changes, we have to retrain the model.

Why this is a problem with neural networks?



Consider **layer 3** of this neural network. It gets input from the former layers as $a_1^{[2]}$, $a_2^{[2]}$, $a_3^{[2]}$ **and** $a_4^{[2]}$ and it helps the parameters $W^{[3]}$, $b^{[3]}$, $W^{[4]}$, $b^{[4]}$ and $W^{[5]}$, $b^{[5]}$ to learn from these features and try to map them to the final output $\hat{y}$ and make it closer to y. But these features $a_1^{[2]}$, $a_2^{[2]}$, $a_3^{[2]}$ **and** $a_4^{[2]}$ are again learned by the features from earlier layers and it changes when the parameters $W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$ changes. So, if these parameters change as they get updated it would eventually change the features passed to the third layer. So the parameters of that layer and the proceeding layers have to adapt to this new change in distribution of features it gets to learn from.

What normalising the features does is that it wouldn't allow the features to change its distribution by keeping its mean and variance same as before even after the values are changed when we do gradient descent and updating. So even thought the values are changing it would still be having the same distribution as shown in the below graph. So this would speed up the learning.



Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

So as the noise is added to hidden layer activations the model cannot relay upon any of the nodes as any of it can have more noise in the future. So, this would make the weights to be spitted among the different nodes more uniformly without giving any particular node more weight.

Also, this noise would get reduced as we use bigger batch sizes. The noise occurs due to the change in z to $\tilde{z}$ and the mean and variance we find are only taking into consideration the values present in the batch and not the entire training set. So as batch size decreases the noise increases and vice versa.

Batch Norm at test time

At test time we won't be having batches to do mini batch. We would probably have single training sample to do normalization. To in order to do that we would find the mean and variance of the distribution as we train the data and we use that to do the mini batch normalization. But instead of finding the mean and variance by running the dataset on the whole model we would keep track of the mean and variance as exponentially weighted averages and would use these as mean and variance to normalize the values at test time.

So similarly, like we found the average of temperature changes over a given window, we would find the mean and variance of the data over a number of iterations.

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

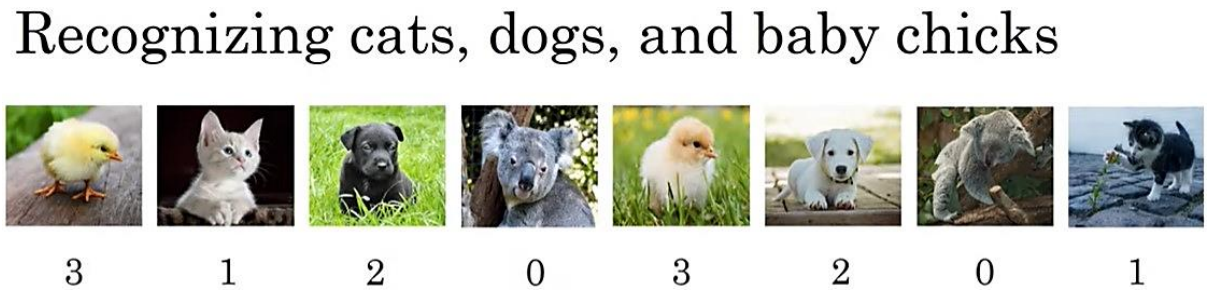
$$\tilde{z}^{(i)} = \gamma z^{(i)}_{\text{norm}} + \beta$$

This is the equation to do batch normalization. Now we would find μ and σ^2 by finding exponentially weighted average across mini-batches $X^{\{1\}}$, $X^{\{2\}}$, $X^{\{3\}}$ etc. as $\mu^{\{1\}[l]}$, $\mu^{\{2\}[l]}$,..etc. and similarly the variance too and use this to normalize the test inputs.

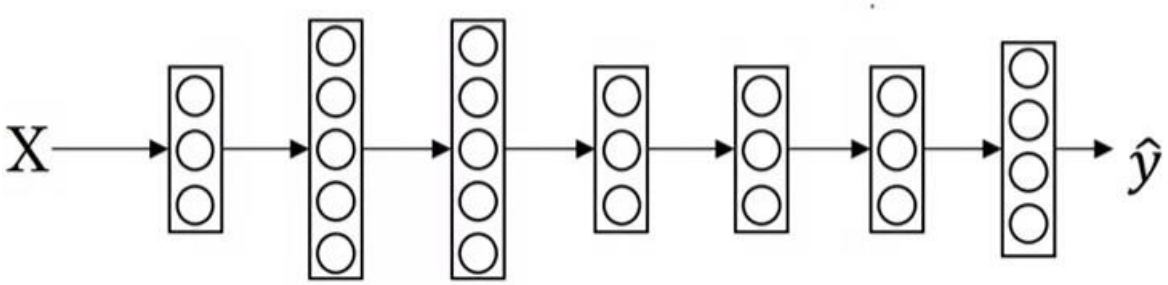
Softmax Regression

Softmax is a generalization of logistic regression, here we classify our output to one among more than one class or c number of classes.

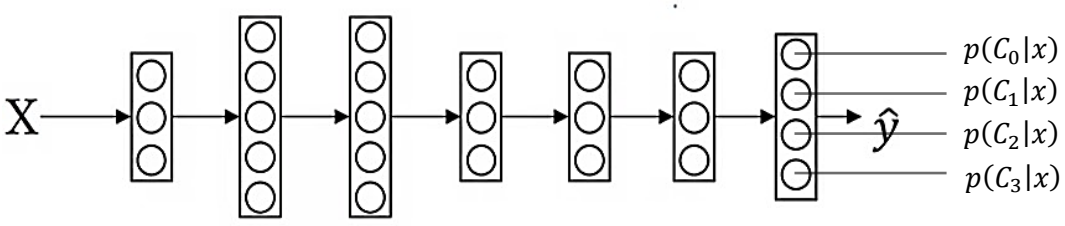
Before when we used logistic regression to predict cat-vs non cat. Here we can take an example of 3 animals. Class 3 for baby chick's, class 2 for dogs, class 1 for cats and class 0 for other animals. Here we use Softmax as the activation for the whole final layer vector to predict which class our output belongs to. In Softmax regression the number of units in output layer = $n^{[L]}$ = c, the number of classes.



In this case we would use a neural network with 4 output units.



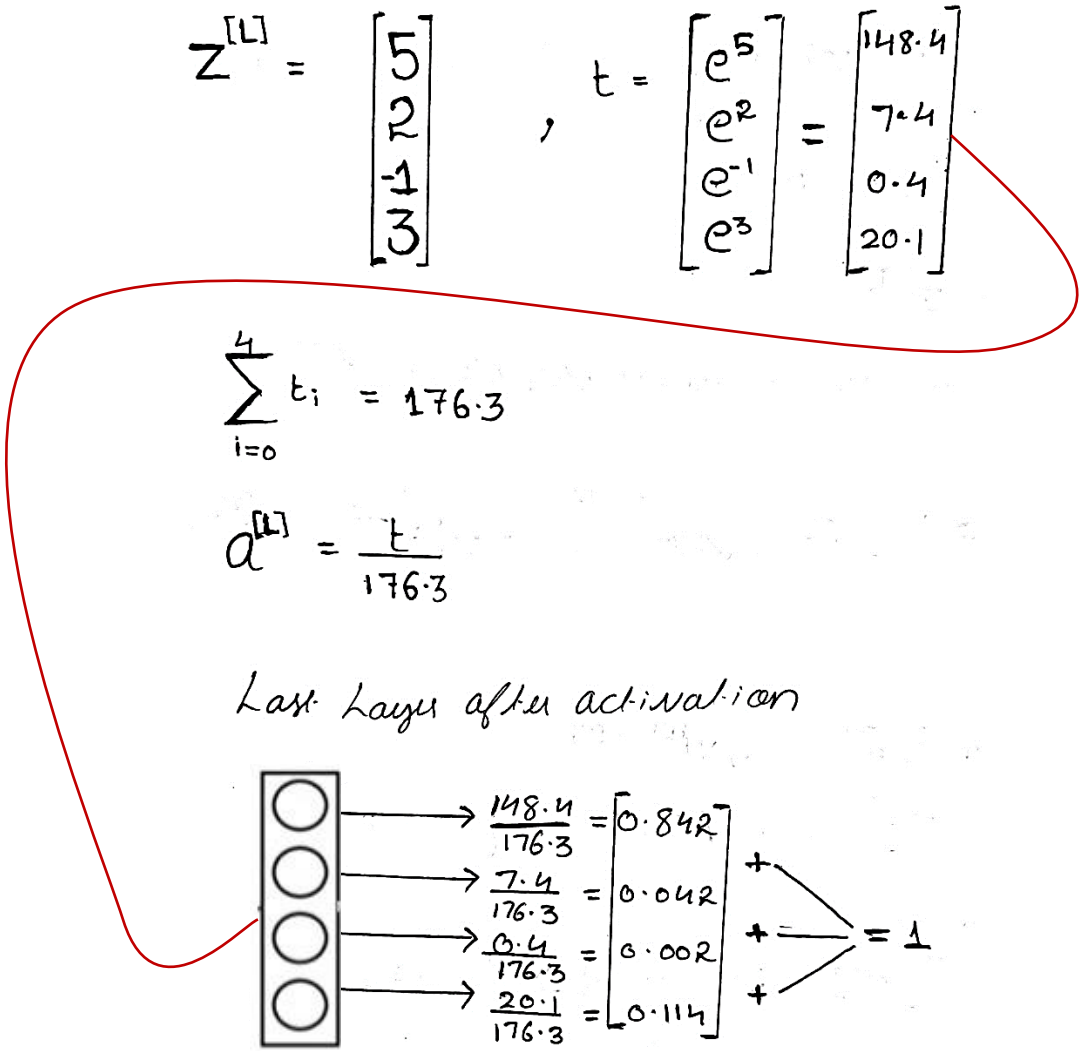
Here the outputs will be probability of each class being the correct output given x



Here c_1, c_2, c_3, c_4 represents the four classes.

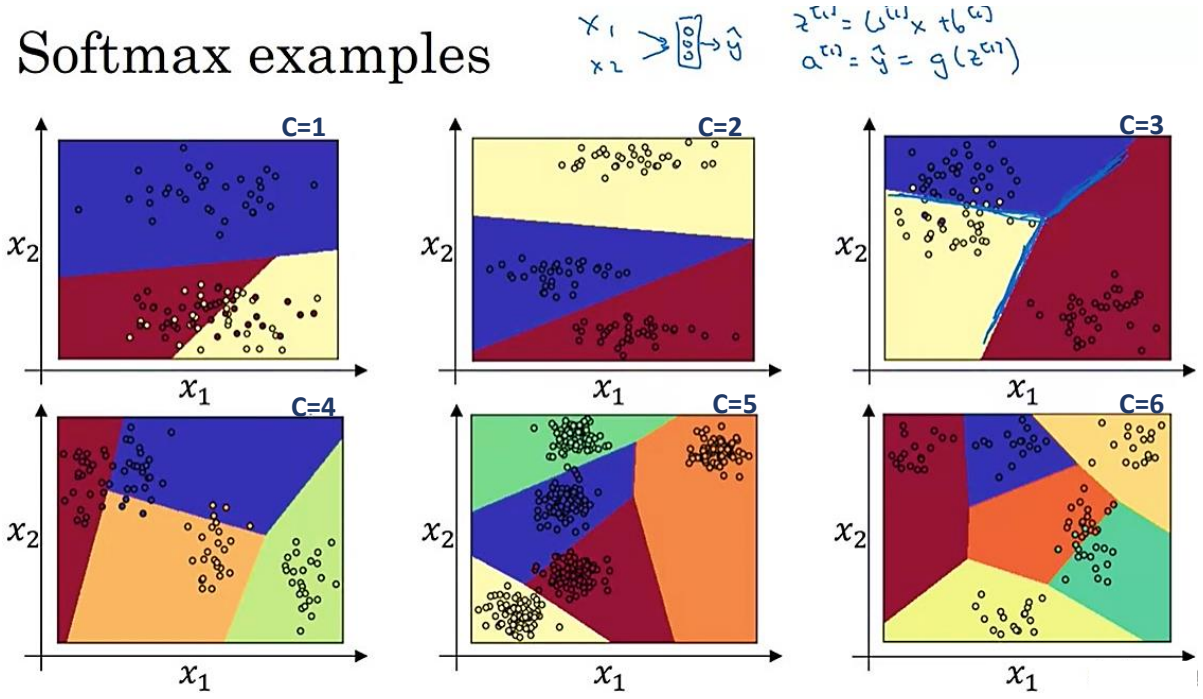
In order to generate these probabilities, we use a layer called Softmax layer in the output layer. Let's see how this works:

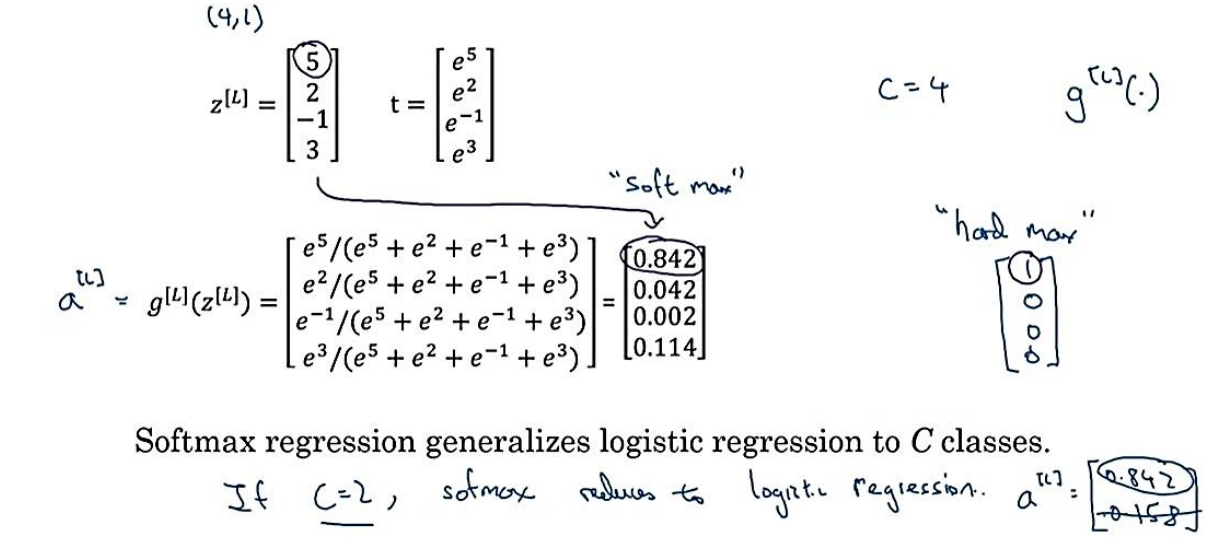
First, we would calculate a temporary variable named $\mathbf{t} = \boldsymbol{e}^{\boldsymbol{z}^{[L]}}$, The final output $\boldsymbol{a}^{[L]}$ will be equal to $\frac{e^{z^{[L]}}}{\sum_{i=0}^{4} t_i}$ so $\boldsymbol{a}_{\boldsymbol{i}}^{[L]} = \frac{t_i}{\sum_{i=0}^{4} t_i}$. Lets take an example to explain this.



Here we can see the different probabilities of different cases being the output that is closer to the true value.

Now let's see some examples of Softmax function applied to a neural network with no hidden layer to classify different classes of outputs





The name Softmax is used in contrast to Hardmax where we give 1 to the element with highest probability and 0 to the rest. Also another thong to note is that if the number of classes c = 2 then

instead of calculating two outputs we can calculate the probability of just one to understand the probability of the other since both would sum up to one. Therefor Softmax with c = 2 is essentially logistic regression.

Now let's take a look at the loss function for Softmax classifier:

Let $Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}] =$

$$\begin{bmatrix} 0 & 0 & 1 & \cdots \\ 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \end{bmatrix}_{(4,m),c=4}$$

Let $\hat{y}^{(i)} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$

$\therefore \hat{Y} = [\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(m)}] =$

$$\begin{bmatrix} 0.3 & \cdots \\ 0.2 & \cdots \\ 0.1 & \cdots \\ 0.4 & \cdots \end{bmatrix}_{(4,m)\ c=4}$$

Loss function $\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j$

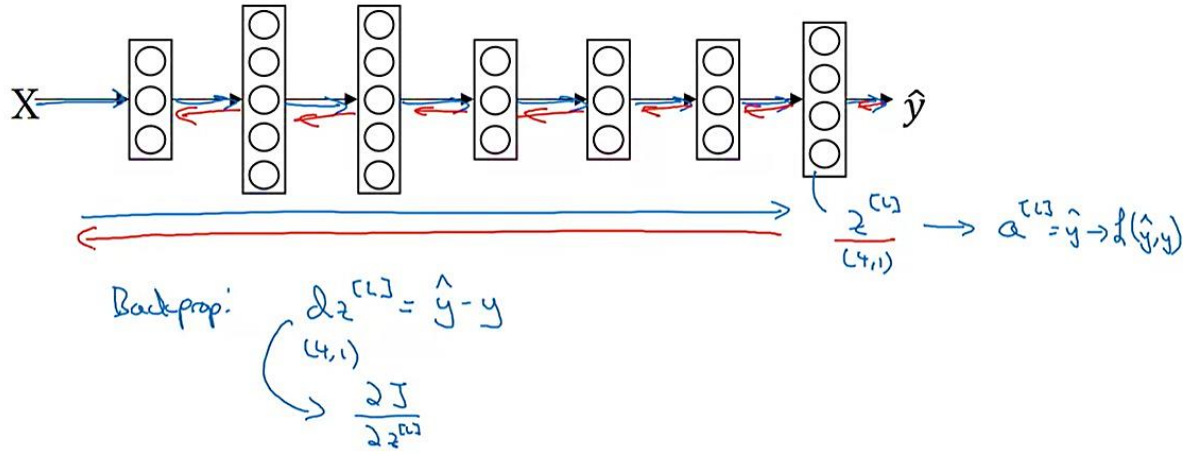
for $j=2$ $y_j = y_2 = 1$ for y' and all others are 0

$\therefore \mathcal{L}(\hat{y}, y) = -\log \hat{y}_2$

So to reduce loss we need to increase $\hat{y}_2$ ie increase $\hat{y}_{y=2}$ to a much bigger value so that the probability would be close to 1.

in general cost $J(w^{(1)}, b^{(1)}, \ldots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y)$

Gradient descent with softmax



Gradient descent with Softmax is exactly same as that of gradient descent we did before.

***Note**

For those who are trying to do TensorFlow calculations as shown by Andrew Ng in the lecture video on their own system, please note that Andrew Ng is using TensorFlow 1 instead of 2 which the latest at the time I write this. So, follow these steps if you encounter any trouble during working out those codes shown in video.

*Assumption: - OS is windows 10

1. Install latest visual C++ distribution from Microsoft
2. If conda install doesn't work try pip install as shown in TensorFlow documentation.
3. No need to install older version instead do this while importing to get version 1 methods:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```