Week #3:- Hyperparameter Tuning

Tuning Process:-

So, far we have seen that, training a neural network involves setting up different hyperparameters. So, what's the setting of these hyperparameters that might lead you to optimized model. Let's find out about that.

→ One of the painful things is to find out about optimal setting of hyperparameters ranging from learning rate to many other independent variables like.

- α ⇒ learning rate ; β ⇒ momentum term
- $\beta_1, \beta_2, \varepsilon$ ⇒ Adam's parameters
- # of layers in neural network
- # of hidden units
- Learning rate decay.
- Mini-batch size

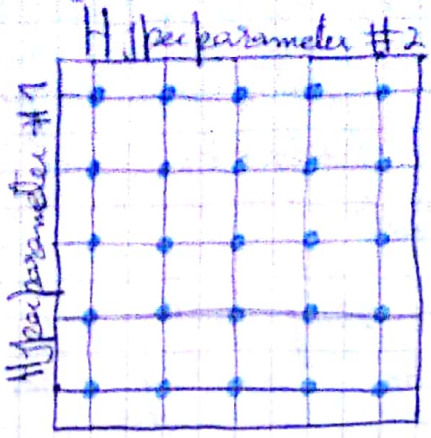
→ Some of them are more important than the others. Learning rate is the most important one.

→ Now if you are tuning a set of values for these hyperparameters, then how do you go about choosing the values

→ In earlier generations of deep learning algorithms if you had two hyperparameters, it was common practice to sample the values in grid and systematically check all of them and pick

which hyperparameter works best

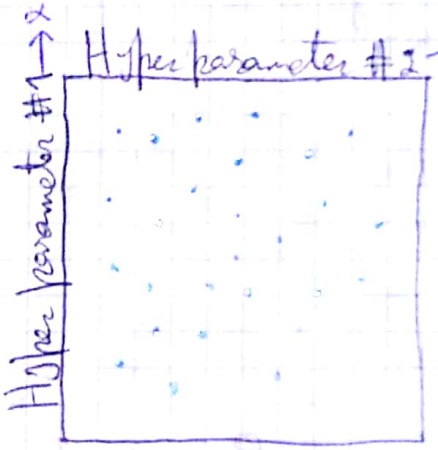
Scheme #1



This practice works OK! when no of parameters are small.

What is recommended in practice is that, you build your line for points and choose points at random. Like previously we choose to test 25 points and choose them at random.

Scheme #2



The reason you do that is because you don't know which hyperparameters are going to be important for your problems.

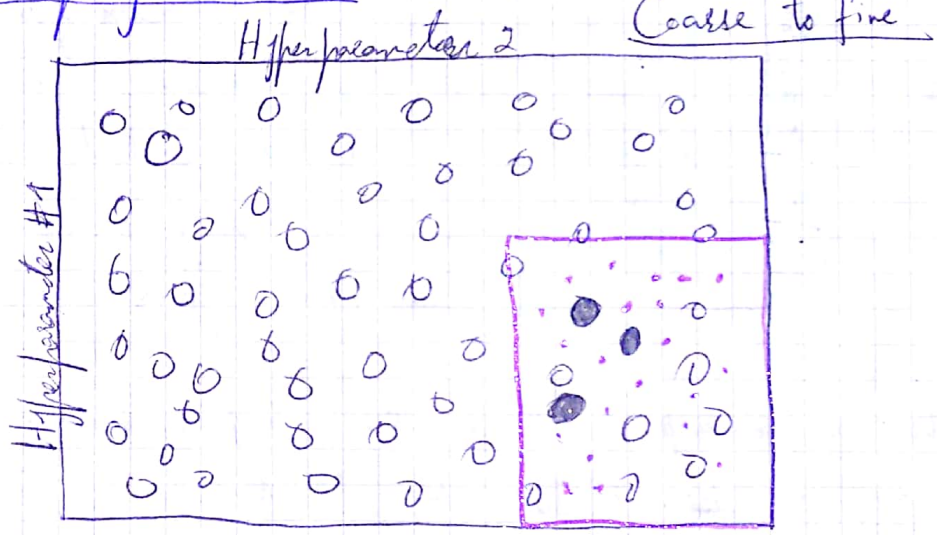
- To take an example, let's say hyperparameter #1 turns out to be alpha the learning rate. To take the extreme, let's say that hyperparameter #2 is the ε from Adam optimization algorithm.

Following the scheme #1, you try out 25 different values for α and ε. You try 25 different models and find out that, all of them give you the same cost. Here you try 5 different values of α.

whereas in contrast, if you follow scheme #2 then you'll have checked 25 different values of α and you'll be more likely to find a value that works really well.

→ I have explained this using just 2 examples, in practice, you might be searching over many hyperparameters. So, if you are dealing with 3 hyperparameters, instead of searching over a square, you might be searching over a cube

→ Another technique in hyperparameter search is <u>coarse to fine sampling scheme</u>.



What you do here is that, you check this coarse grid and find that the bold points tend to work really best. What you'll do than is that, you'll select the area (legibally) around it and by optimizing you model there (selecting points randomly of course)

Using appropriate scale to pick hyperparameters:-

Lastly we saw that, how sampling the parameters for hyperparameters' values could allow you to pick your hyperparameters' values appropriatly. But it turns out that, sampling at random doesn't mean sampling through random values uniformly over the range of values. Instead it is important to pick the appropriate scale to the hyperparameters' values.

Let's see how to do that.

→ Let's say you are trying to choose the value of number of hidden units and let's say you think that a good range is between 50 – 100.

$$n^{[l]} = 50, \ldots, 100$$

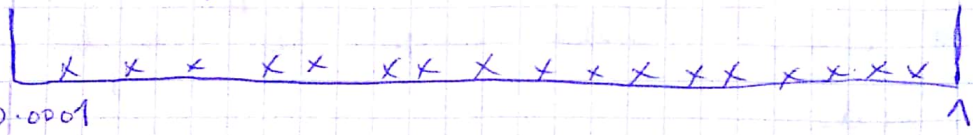
So picking a number for no. of hidden layers randomly between 50 . 100 might be ok for evaluation and see the results. But this is not ok for all hyperparameters.

Let's look at another example:-

Say you are searching for the hyperparameter "alpha"

You think that 0.0001 would be on the low and it be as high as 1.

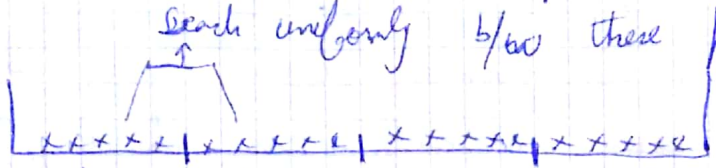
Now when you draw the number line



and sample values at random uniformly on this number line.

~~xxxxx~~ You sample 90% values between 0.1 and 1 and only 10% values between 0.0001 and 0.1

- That doesn't seem right. Instead it would be more reasonable to search the hyperparameters on the log scale.



$10^a \rightarrow$ 0.0001 0.001 0.01 0.1 1 $\leftarrow 10^b \Rightarrow b = \log_{10}(1) = 0$

$a = \log_{10}(0.0001) = -4$

This gives you more resources to search for your hyperparameters.

In python the way you implement this is:-

Let: $r = -4 * np.random.rand()$ $\rightarrow r \in [-4, 0]$

$\alpha = 10^r$ $\leftarrow 10^{-4} \dots 10^0$

$10^a \dots 10^b$ Sample r uniformly at random between a and b $r \in [a, b]$

Hyperparameters for exponentially weighted averages:-

Let's say you suspect $\beta = 0.9 \dots 0.999$

Using $\beta = 0.9$ is like averaging over last 10 values.

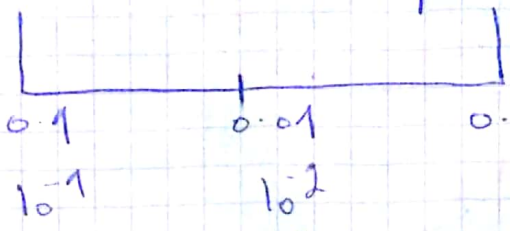
Remember $10^? = 1/e$ and $\beta = 0.999$ is like averaging over last 1000 values

→ as we saw if you want to sample b/w 0.9 and 0.999, it doesn't make sense to sample on a linear scale.

Better way to think of it is to sample values

for $(1-\beta)$ i.e between 0.1 and 0.0001

- So may be we will sample on the scale



$r \in [-3, -1]$

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

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

This way you would have much more resources to explore the value of β.

- Now question is why is this choosing of linear scale is a bad idea? What is the mathematical intuition behind it?

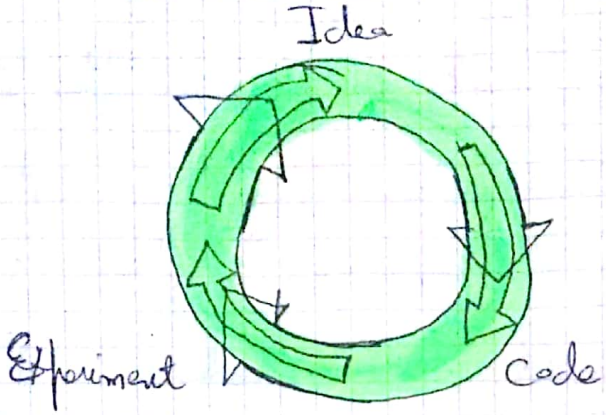
→ Now when β is close to 1. The sensitivity of your results changes even with a very small change in the input- i.e β

→ e.g if β goes from $0.9000 \rightarrow 0.9005$ it is fine no big deal.

→ But if β: $0.999 \rightarrow 0.9995$ then this will have a huge impact on your results.

Because in the 1st case you are averaging over ≈ 10 examples but in the second case you go $1000 \rightarrow 2000$ examples to average over.

Hyperparameter tuning in Practice: Pandas Vs Caviar:-



Deep learning is usually applied to many fields like NLP, Vision, Speech, Ads, logistics and knowledge from one field regarding optimized hyperparameters may or may not transfer to the other field.

- But some fields' ideas can be applicable amongst each other, like Vision's can be applied to Speech, NLP and vice versa.

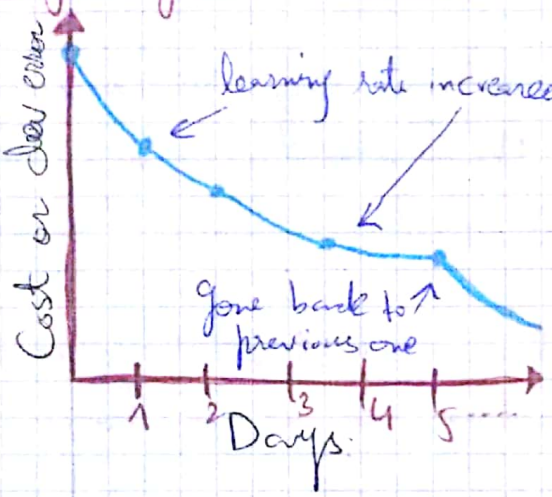
In terms of hyperparameters, Intuitions do get stale.

⇒ Even if you work on just one problem say logistics, you might find a good setting for your hyperparameters and kept on developing over the month on the updated data. So setting of hyperparameters can get stale, so I recommend to re-evaluate the setting of your hyper-parameters Once at-least every few months to make sure you are happy with the values you have.

There are two major school of thoughts for the tuning of hyper-parameters. One is Babysitting / one model.

- This can be the case if you have not a lot of powerful computational resources but have huge amount of data so you can afford to train only one model.

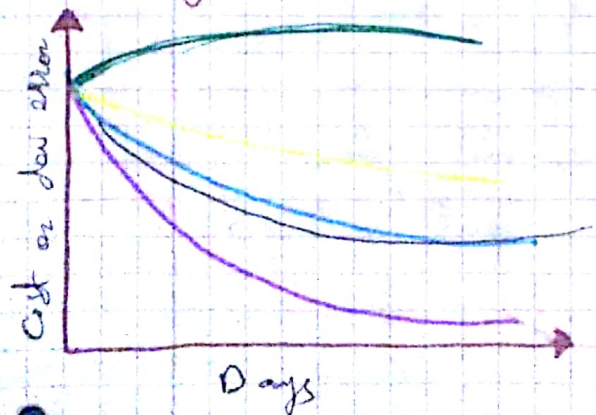
Babysitting One Model:-



The approach is that, you train the model and see its performance after pre-ct intervals, like after Day 1 you see your model doing pretty good so you think of increasing the learning rate. It's still doing fine you tune other paramters and so one work like this until you find optimal setting.

→ Other approach is that, you train many models in parallel.

Training many models in Parallel:-



Here you let the models run in parallel and analyze their performance with different hyperparameter settings

- Different colors
  Dipict different models

→ In order to make an analogy, let's call the 1st approach, The Panda approach.

When pandas have children not few children, so they make sure not keep all their effort on one child at a time so that, baby panda survives. That's more like baby sitting.

→ 2nd approach is what a Fish do. Let's call it Caviar strategy.

Some of the fish lay 100 million eggs in one mating season and keep their attention on a bunch of them that they survive.

→ The way to choose b/w them is that, it depends upon how much computation power you have.

Batch Normalization :-

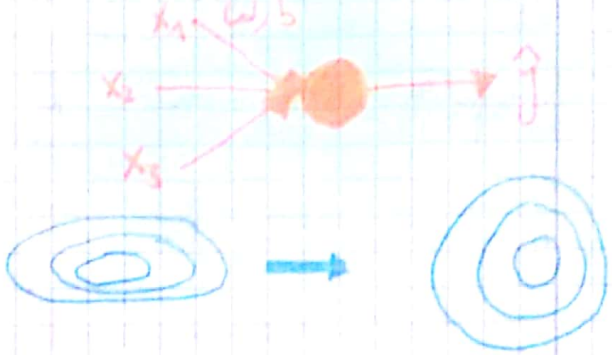
→ Normalizing activations in Network :-

One of the better ideas when it comes to deep learning is batch normalization. It makes your hyperparameter search problem much easier, makes your neural network much more robust. Choice of hyperparameter is a much bigger range of hyperparameters that work well, and will also enable you to train much more easily very deep neural networks.

Let's see how that works :-

→ When training a network, you know that, normalizing the input features can speed up your learning

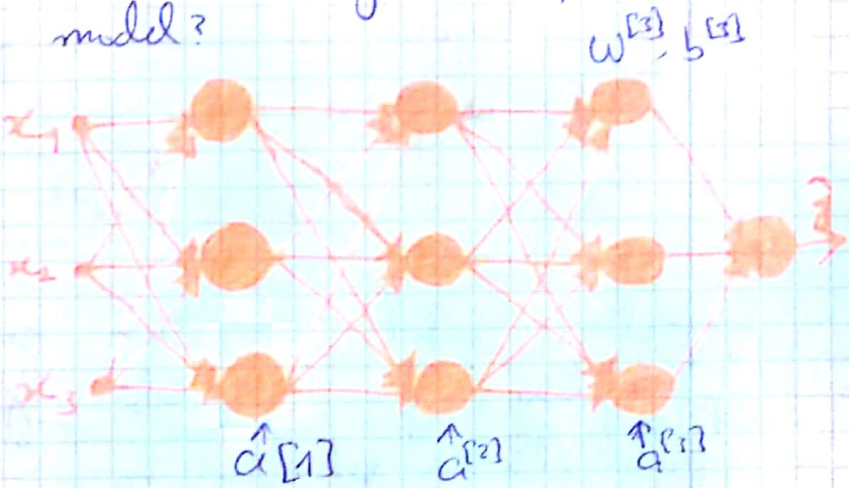
$$\mu = \frac{1}{m}\sum_i x^{(i)}$$
$$X = X - \mu$$
$$\sigma^2 = \frac{1}{m}\sum_i x^{(i)2}$$
$$X = X / \sigma^2$$



This works in normalizing the i/p feature values to a neural network or a logistic regression

- How about a deeper model?

Here you not only have i/p features but also 3 more layers. Suppose you want to train the parameters $W^{[3]}, b^{[3]}$. Wouldn't it be nice to train them by normalizing the input ($a^{[2]}$)?



Here you'll normalize $z^{[2]}$ not $a^{[2]}$.

⇒ There is a debate regarding whether to normalize activations of z's. But in practice more often z's are normalized.

Here is how you implement batch normalization.

Given some intermediate values in neural network

$z^{[1]}, \ldots, z^{[l]}$.

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

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

$$z^{[i]}_{norm} = \frac{z^{[i]} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$; for numerical stability.

This will give you mean zero and s.d 1 for every layer and we don't want that:-

So what we are gonna do is:- learnable parameters of model.

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

Regardless what algorithm you use using for optimization, you update γ and β just like you do w and b.

→ Effect of γ and β is that, you can set the mean of $\tilde{z}^{[i]}$ whatever you want it to be.

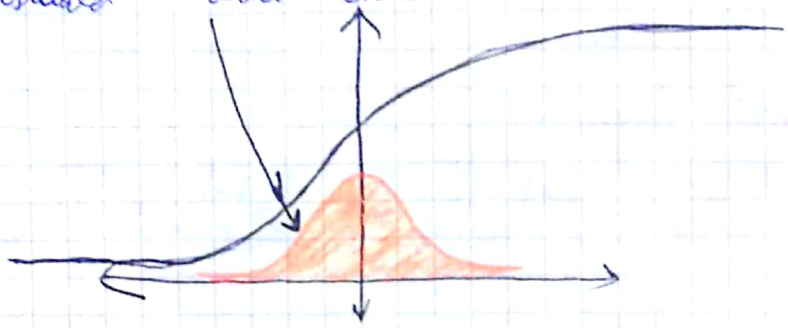
e.g. if $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, then

$$\tilde{z}^{[i]}_{norm} = z^{[i]}$$

→ Now you'll use $\tilde{z}^{[i]}$ instead of $z^{[i]}$ in your computations of neural network.

One difference though between input values and hidden units is that, you might not want your hidden unit values to be forced to have mean zero and variance 1

e.g. if you are using Sigmoid activation functions, you might not want all of your values to be clustered like this:-



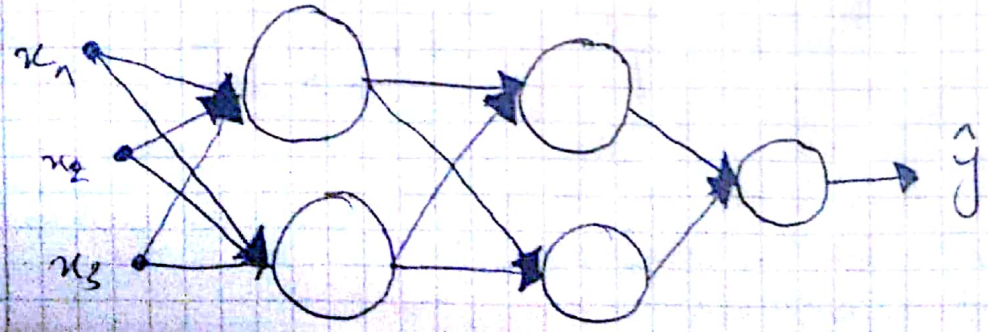
You might want it to have larger variance or mean that is different than zero, in order to take advantage of the non-linearity of sigmoid function in a better way.

That's why with parameters γ and β, you can make sure that, your network has the range of values you want.

Fitting batch norm into a neural network :-

Let's see how to fit a batch norm to a whole neural network.

Let's consider the following network



Each of the unit compute 2 things :-

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

Normally

But what we do in batch norm is that :-

$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{batch norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \xrightarrow{g(\tilde{z}^{[1]})} a^{[1]}$$

Now for next layer :-

$$a^{[1]} \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \xrightarrow{g(\tilde{z}^{[2]})} a^{[2]} \rightarrow \cdots$$

So in short for firing of neurons, instead of using Un-normalized z, we are going to use normalized values $\tilde{z}$

→ Now we have parameters, $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \ldots w^{[l]}, b^{[l]}$

$\beta^{[1]}, \beta^{[1]}, \beta^{[2]}, \beta^{[2]}, \ldots \beta^{[l]}, \beta^{[l]}$

→ Now that, you have other parameter too in your model rather than w and b, you will use optimization algorithm to update them too.

Like compute $d\beta^{[l]}$ and $d\gamma^{[l]}$ and

then update them usually like

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

In programming frameworks for Machine learning like Tensorflow or Pytorch you don't have to implement them by yourself from scratch, you can use pre-defined functions from these libraries.

Till now we have talked about batch normalization like we are using batch gradient descent. But in practice we apply this technique with mini-batches.

→ Now what you do is that, you take first mini-batch and apply batch-norm on that mini-batch only and follow the rest of the trend as it then repeat the whole process with other mini batches.

$X^{\{1\}} \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \xrightarrow{g(\tilde{z})} a^{[1]} \xrightarrow{W^{[2]} b^{[2]}} Z^{[2]} \rightarrow \cdots\cdots$

Now you do the same with other mini-batches.

→ One thing to be noted here is that, your values of β and γ here which are calculated from just using the data from second mini-batch.

$X^{\{2\}} \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \rightarrow \cdots\cdots$

⋮

One other detail regarding the parameterization of the neural network now is as follows:

We established that we have following parameters:

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

Notice that $Z^{[l]}$ is computed like this

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

Then normalize $z^{[l]}$ to first have mean zero and standard variance. Then rescale them by β and γ.

→ What that means is that, whatever is the value of $b^{[l]}$ is just gonna get subtracted out, because in that batch normalization, you are going to compute the mean of $z^{[l]}$ and subtract out the mean. So any constant you add will get canceled out by the 'mean subtraction step'.

So, what you can do is eliminate b term.

→ As batch norm zeros out the mean so there is no point in having "b" term there. Instead it is kind of replaced by β which now control the mean and variance.

$$\beta^{[l]}, \gamma^{[l]} \in \mathbb{R}^{(n^{[l]} \times 1)}$$

Implementing Gradient Descent Using mini batch

for $t = 1 \ldots$ num batches:

Compute forward prop on $X^{\{t\}}$

In each hidden layers use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

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

update parameters

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

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

$$\gamma^{[l]} := \gamma^{[l]} - \alpha\, d\gamma^{[l]}$$

Works with momentum, RMS as well

Why does Batch Norm Work?

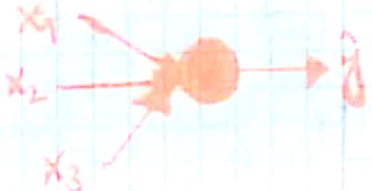
- You have already seen that, by normalizing input features you can speed up learning. One intuition is that, you are doing the same thing in the hidden units.

Now this is the shallow picture; there are couple of other things which can help you to gain better understanding of what batch norm is doing.

- One lesser is that, batch norm makes weights later in the neural network more robust to changes
- earlier in the neural network.

To explain what does it mean, let's look at this example:-

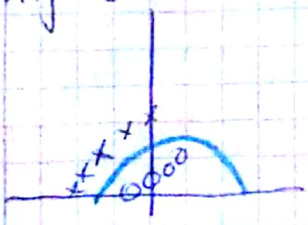
Let's see this shallow network, may be a logistic regression.



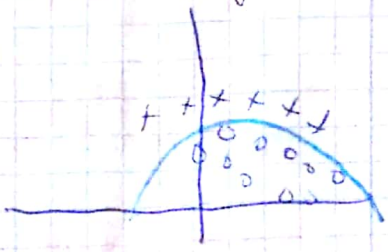
→ Let's say you have trained your neural network on all black cats. If you now apply your neural network to data with coloured cats then your

- classifier might not do very well

e.g. if you have an algorithm where you use training data like this :- (the one on left)



then your model can't perform well on this ~~mode~~ data ⟹



-

This idea of data distribution changing goes by the name of "Covariate shift"

The Idea is that :-

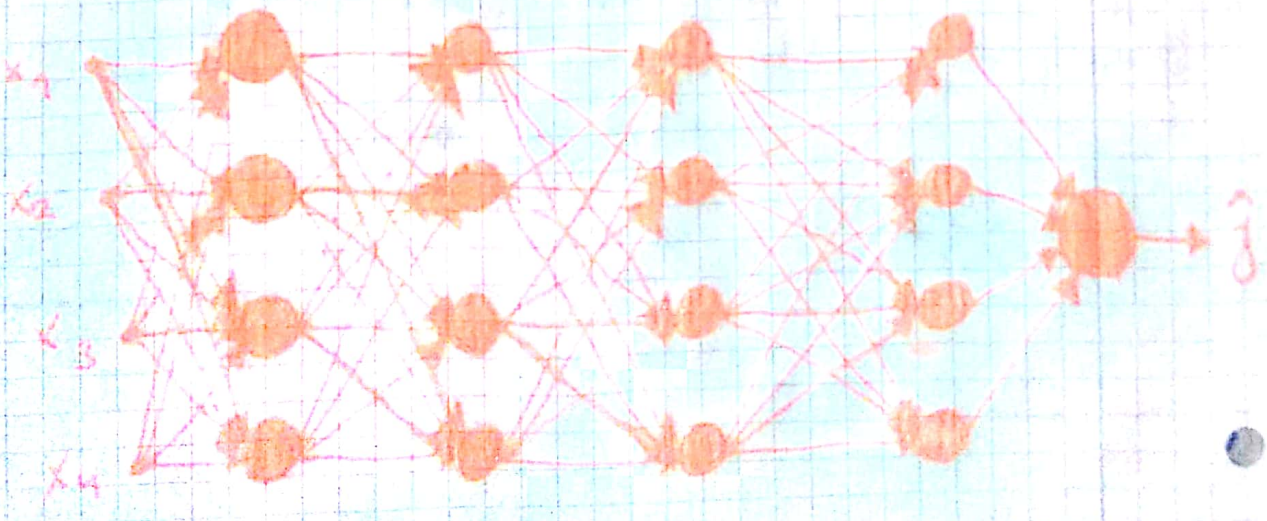
> If you are using some data to train model that shows $X \rightarrow y$ mapping and after training if your data distribution changes then you'll need to re-train your model.

→ This is true even if ground true function mapping from X to Y remains unchanged. Like in the earlier problems that we tackled in assignments the ground true function was, whether this is Cat or not.

If the ground true function changes then the need to retrain your function becomes even worse.

Now:- how does this problem of **Covariate Shift** apply to neural network?

Consider a deep neural network like this.



Let's look at the learning process from the perspective of 3rd hidden layer.

So, this network, has to learn parameter $W^{[3]}$, $b^{[3]}$

So, this layer get parameters $a_1^{[2]}$, $a_2^{[2]}$, $a_3^{[2]}$, $a_4^{[2]}$ from previous layer and the job of 3rd hidden layer is to find a mapping that takes these incoming values and map them to $\hat{y}$.

Network is also adapting parameters $W^{[2]}$, $b^{[2]}$ and $W^{[1]}$, $b^{[1]}$ so, as these values change, $a^{[2]}$ will also change and this is the problem of covariant shift.

what batch norm does is that, it reduces the amount of the distribution of ~~these~~ which these hidden units' values shift around with.

if on one iteration, if $a_1^{[2]}$, $a_2^{[2]}$ change then on the next iteration it will again change. Batch norm ensures that, the mean and variance remains the same even if the values change.

→ Batch norm reduces the problem of input values changing, it really causes these values to become more stable so that later values have much more stable grounds to stand on.

Even if the earlier distribution changes, it changes less, and what it does it forces less the later layers to changes as a result of change

in earlier layers. In loose terms, it loosens the coupling of earlier layers with the later ones and enable each layer to learn sort of independently. As a result this put in effect the speeding of neural network training.

→ It turns out batch norm has slight regularization effect too.

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 the mini-batch. So similar to dropout. it adds some noise to each hidden layer's activation
- This has a slight regularization effect.

→ Because by adding noise to the hidden units, it is forcing the downstream of hidden units to rely too much on any one hidden unit.

→ Because the added noise is small, this is not a huge regularization effect and you might wanna use batch norm together with dropout.

→ If you increase the batch size, you reduce the noise and hence the regularization effect gets dampen out.

Batch Norm at test time :-

→ Batch norm processes your data, one mini-batch at a time.

But at test time, you might need to process your examples one at a time.

Let's see how you adapt your network to do that.

Recall the equations you use for training of examples:-

→ $\mu = \frac{1}{m}\sum_i z^{[i]}$

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

→ Computed on entire mini-batch.

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

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

→ So, in order to apply it at test time, you need to Come up with different estimate of μ and σ^2

One estimate used is that, you estimate using exponentially weighted average (across mini-batches)

Let's pick one layer l and going through mini batches

$X^{\{1\}}$, $X^{\{2\}}$, $X^{\{3\}}$, ...

$\mu^{\{1\}[l]}$ $\mu^{\{2\}[l]}$ $\mu^{\{3\}[l]}$

θ_1 θ_2 θ_3

You saw earlier that how to use weighted average to Compute the mean. You would do that to keep track of what's the exponentially weighted

average becomes of this mean vector that is the latest. So that exponentially weighted average becomes your estimate for what the mean of Zs is for that hidden layer and similarly you use this exponentially weighted average to keep track of the values of sigma squared that you see on the first mini-batch in that layer, in the second mini batch and so on

→ At test time you would use that exponentially weighted average for μ and σ^2 to compute

Z_{norm}

$$Z_{norm}^{[i]} = \frac{Z^{[i]} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Then you would compute $\tilde{Z}^{[i]}$ using γ and β that you learned during your learning of neural network.

$$\tilde{Z}^{[i]} = \gamma Z_{norm}^{[i]} + \beta$$

→ You can also run your whole network (final) through training data and use those computed final μ and σ^2.

But in practice people use exponentially weighted average of μ and σ^2

Multi-class Classification:-

Softmax Regression:-

→ Till now we have considered only one class for classification, like is it a cat or not?

What if you have multiple classes.

There is a generalization of logistic regression called "Softmax regression", where you try to recognize one of multiple classes, rather than one.

→ Let's say instead of recognizing cats only, you want to recognize cats, dogs and baby chicks

Let's say Dogs ⇒ class 2

Cats ⇒ class 1

Baby chicks ⇒ class 3

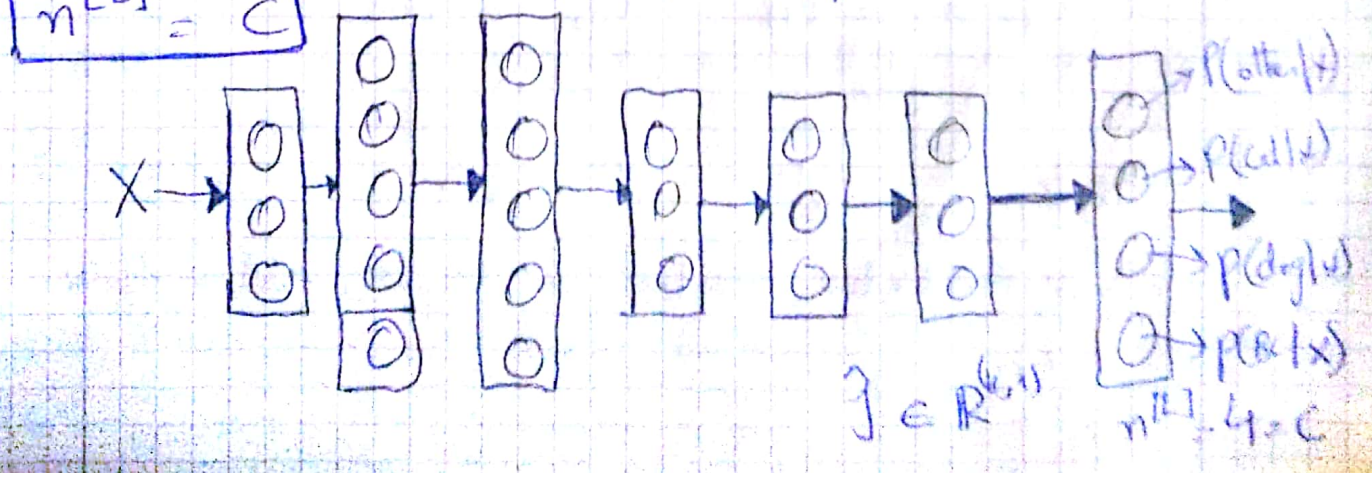
and if none of the above then class 0. or other

Notations:-

C = # of class ⇒ In this class we have 4.
In code it is from (0, ... 3)

In this example we are going to build a neural network with C output units.

$n^{[L]} = C$



$\vec{y}$ should sum to 1.

→ The standard model that a neural network uses to perform this task is called Softmax layer in the output layer in order to generate these outputs.

→ So, in the final layer: you are going to compute:

$$Z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

Now having computed $Z^{[L]}$, you now need to apply Softmax activation function.

Activation function:-

$$t = e^{Z^{[L]}} \Rightarrow \text{elementwise}$$

$$a^{[L]} = \frac{e^{Z^{[L]}}}{\sum_{i=1}^{4} t_i}$$

$t \in \mathbb{R}^{(4\times 1)}$ for our example

$$a_{(1)}^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

e.g: $Z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \Rightarrow t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$

So $\sum_{j=1}^{4} t_i = 176.3$ $\quad a^{[L]} = \frac{t}{176.3}$

So $a_{(1)}^{[L]} = 0.842$ $\quad a_{(2)}^{[L]} = 0.042$

$a_{(3)}^{[L]} = 0.002$ $\quad a_{(4)}^{[L]} = 0.114$

Summary: $a^{[L]} = g^{[L]}(Z^{[L]})$

Softmax activation, takes a vector and outputs a vector.

Training a Softmax Classifier:-

- Let's try to dig deeper into Softmax Classifier. Let's say we have $C = 4$ and recall our original example:- $z^{[L]} \in \mathbb{R}^{(4,1)}$

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \Rightarrow t \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \Rightarrow a^{[L]} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

The name softmax comes from in contrast to what comes as "hardmax"

- What hardmax does is takes a vector and puts 1 in place of the biggest element of the vector. like for this example it will be $[1 \ 0 \ 0 \ 0]^T$

⇒ whereas softmax is a gentler mapping from X to outputs.

Softmax regression generalizes logistic regression to "C" classes

- if $C = 2$ then softmax essentially reduces to logistic regression. So, C should be more than 2

→ Let's see how we train a neural network with a softmax output layer

In particular let's define the "**loss function**" you used to train your neural network.

- Let's say you have an example with a target output of ground true label of $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$

So, following previous example, this would be an image of a cat.

You get your output vector let's say

$a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ so clearly our model is not doing very well in this example

In softmax classification, the loss we typically use is:-

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

$y_1 = y_3 = y_4 = 0$

So only at $y=2$ this function will give some output.

→ As we are trying to minimize the loss so, $-\log(\hat{y}_2)$ should be bigger and for it to be bigger corresponding $\hat{y}$ value should be bigger also

→ This is the loss on one training example, how about the cost on the whole data.

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

Output vector:- $Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$.

$$= \begin{bmatrix} | & | & & | \\ y^{(1)} & \cdots & \cdots & y^{(m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{(4,m)}$$

Same dimensions will be of $\hat{Y}$.

Final detail. Let's take a look about how do you implement gradient descent when you have a softmax output layer.

Key point is to start back prop and you do it like this

$$dz^{[L]} = \hat{y} - y$$
$$(4,1)$$

Rest is the same.

Introduction to Programming frameworks:-

Till now we have learned to implement deep neural networks from scratch. But when we train very big models or heavy applications, it is not practical to do that, rather there are very sophisticated programming frameworks that allow you to do the aforementioned Job.

→ It's just like multiply 2 matrices by yourself via using explicit loop and using numerical linear algebra functions.

- **Different deep learning frameworks:-**

  Caffe/Caffe2, CNTK, DL4J, Keras, Lasagne, mxnet, Paddle Paddle, Tensorflow, Theano, Torch

- **Criterion for Choosing framework:-**
  - Ease of programming (development and deployment)
  - Running speed
  - Truly open (open source with good governance)

Tensor Flow

As a motivating problem, let's say ya have some function

$J(w) = w^2 - 10w + 25$ ——— ①

$J(w) = (w-5)^2$

The value that minimizes this is $w = 5$

Let's say we didn't know that and we have just ① function and let's see how we can implement tensor flow to minimize this.

```
Import tensorflow as tf
Import numpy as np
w = tf.Variable(0, dtype = tf.float32)
Cost = tf.add(tf.add(w**2, tf.multiply(-10, w)), 25)
train = tf.train.GradientDescentOptimizer(0.01).minimize(Cost)
                                  ↑
                              learning rate

init = tf.global_variables_initializer()
session = tf.Session()   OR   with tf.session() as session
  session.run(init)
Print(session.run(w))
>> 0.0
Session.run(train)
Print(session.run(w))
0.1
for i in range(1000):
      session.run(train)
Print(session.run(w))
>> 4.999
```

This was a fixed function, when training a neural network, training data can change. How to load that data

→ tells the program that, you'll provide the data for this variable

```
X = tf.placeholder(tf.float32, [3,1])
```

```
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
```

```
Coefficients = np.array([[1], [-10.], [25.]])
```

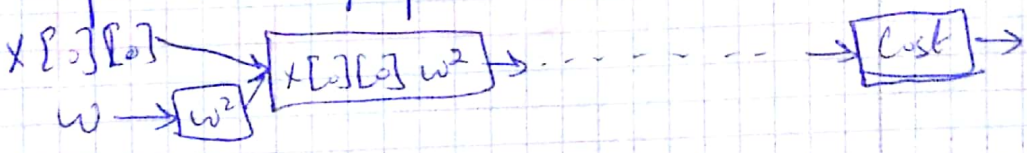
→ Now we have to get Coefficients into X

for that, we write follows :-

```
Session.run(train, feed_dict={x: coefficients})
Print (session.run(w))
```

→ You just have to give tensorflow the function and it will it self figure out the derivatives and all.

→ what highlighted line is doing is allowing tensorflow to build a computational graph



One nice thing about tensorflow is that you give it the functionality for forward propagation and it has already builtin functionality for backward propagation.

→ By using the builtin functions for forward function, it can automatically do the backward function as well to compute backpropagation

Writing a program in the tensorflow has the following steps:-

1- Create tensor (Variables) that are not yet executed/evalu
2- Write operations b/w those tensors.
3- Initialize your tensors.
4- Create a session.
5- Run the session. This will run the operations you'd written above.

E.g:-

```
y_hat = tf.constant(36, name='y_hat')
y = tf.constant(39, name='y')
loss = tf.Variable((y - y_hat)**2, name='loss')
init = tf.global_variables_initializer()
with tf.session() as session:
    session.run(init)
    Print(session.run(loss))
```