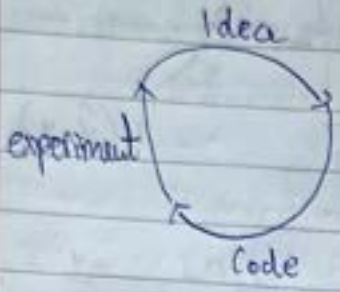
Applied deep learning is a very emperical process

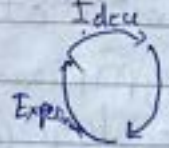


Applying DL is an iterative process by doing trial and error:
→ Set initial α value. Start with guesses for hyperparameters. $\alpha = 0.01$
→ Train and Evaluate: Observe the behaviour of cost function J:
- if J converges slowly, α might be too small
- if J diverges, α might be large
→ Adjust the hyperparameters based on observations

~~What does this have to do with the brain?~~

Course 2: Improving Deep NN: hyperparameter tuning, regularization, optimization
Week 1
Setting up ML applications
Train / Dev / Test sets

Applied ML is a highly iterative process. When training a neural network, we need to make a lot of decisions:
→ no of layers NN will have
→ no of hidden units you want each layer to have
→ what activation function you want to use for each layer
→ what learning rate to use



⇒ Very hard to correctly guess the hyperparameters the very 1st time
→ Applied DL is an iterative process where you undergo the cycle many times to have a good choice of network for the application.
→ One thing we can determine is how quickly you can make progress around the cycle effectively.

Course 2 Improving Deep NN - Hyperparameter tuning, Regularization, Optimization

Workflow:

→ Train the algorithms on train set

→ Use the dev set to see which different models perform the best on your dev set.

→ When you have the final model that you want to evaluate. Use the best model you found on the test set to get an unbiased estimate of how well the algorithm is doing.

- Previous era we used: → 70% train, 30% test
  → 60% train, 20% dev, 20% test
  → This is ok if we have examples 10 - 10000

- In the Big Data era where we got 1 million examples, the trend is that your dev, test set has to be smaller percentage of the total
  Remember: Dev set only has to be big enough for you to evaluate different algorithm choices and decide which works better.
  ex, if we 1 mil ex for dev set we only need 10000 ex

→ Split for 1 mil ex, 98% train, 1% dev, 1% test
  we can also use 99.5% train, 0.25% dev, 0.25% test
  or 0.5% train, 0.4% dev, 0.1% test

Mismatched train / test distributions:

ex We build an app to detect photos of cats.

Train set: Cat pictures from webpages (This was professional, high res, nicely framed pictures)

Test / Dev set: Cat pictures from users using the app (users will take photo from their own camera → which maybe blurrier, low res)

Distribution of data is different

→ Rule thumb: Make sure the dev and tests come from the same distribution

→ ex for rule of thumb: if cat train ~~pic~~ x's are from internet and dev/test x's are user uploaded from the user's phone they will mismatch. Make sure ~~you use~~ that dev and test set are from the same distribution

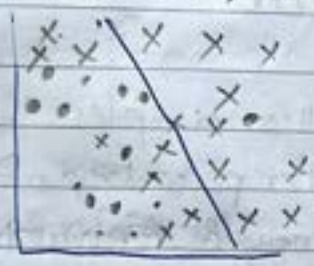
→ The dev set rule is to try some good models you created.

→ If you don't need an unbiased estimate, its fine not to have a test set. Since we don't have the test set, we train on train set and then try different model architectures. Evaluate them on dev set and then use to iterate and try to get good model. Since we fit data on dev set it no longer gives an unbiased estimate of performance

→ In the above case we call dev set as test set

Bias and Variance

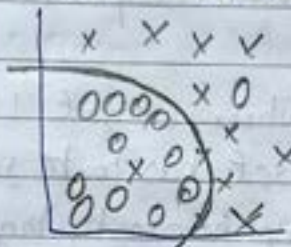
Logistic regression



high bias
Underfitting

- Doesn't fit very good to data has a high bias
- ~~high bias~~ Underfits

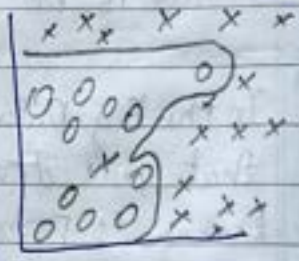
clf with medium complexity



just right

- Fits just right

complex ~~clf~~ classifier
Deep NN (a lot of hidden units)



high variance
Overfitting

- May fit data perfectly
- Not a great fit
- High variance and overfits

ex, Dog / Cat classification:

Case 1:	Case 2:	Case 3	Case 4
y=1 (cats), y=0 (dog)			
- Train set error : 1%	Train error : 15%	Train err : 15%	Train err : 0.5
- dev set error : 11%	Dev set error : 16%	Dev set err : 30%	Test err : 1%
- We have high variance model has high dev set error and low train set error.	- Underfitting - High bias - Doesn't do well on both train, dev set	high bias high variance	low bias low variance

- These assumptions come from the baseline that human has 0% error or more generally they are the optimal error / Bayes error

Case 5:
Train error : 15% ← Pay attention to
Dev error : 16% ← the metrics
human error : 15%
We say the above case is low variance because the human err itself is 15%.

In the case of analysing bias / variance when no clf can do well :
→ ex, we have very blur images, even human or the system can't do well ~~than~~
→ ~~Human~~ Bayes Then the bayes error is too high

* Note:
→ By looking at Jtrain it gives an idea of how well your ~~fitting~~ model fitting to the train set it tells if we will have a bias problem.
→ Looking at how high error goes when you go from train set to dev set, that gives a you sense of how ~~well~~ bad the variance problem is.
→ All this is under the assumption ~~that: Bayes error is small, train~~
- Bayes error is small
- Train and Dev sets are drawn from the same distribution
-

High bias and high variance:

Black line denotes high bias and high variance
High bias, because by being a linear clf, it doesn't fit the quadratic shape well - in middle
But by having too much flexibility it gets ~~that shape~~ those ex : high variance.

Basic Recipe for ML

After training an initial model we ask if algorithm has:

Repeat these until it can fit the train set well

High bias? (Train data performance) → Yes → Try bigger network (always helps) (more hidden layers, hidden units) train longer (doesn't always help) try advanced optimization algorithms, use another NN architecture better suited for problem)

↓ No

High variance? (dev set performance) → Yes → More data, Regularization, Find better NN architecture

↓ No

↓

Done

Key points

⇒ Use the train and dev set to determine if you got a bias or variance problem, then you can do the appropriate things.

⇒ ~~If we have~~ High bias problem:

- Getting more train data would help (not efficient to do either)
- Try bigger network (more hidden layers & hidden units)
- Try advanced optimization algorithms
- NN architecture search (May help)

High variance:

- More data
- Regularization
- NN architecture search

-> Bias-Variance tradeoff: -> Increase bias and reduce variance
-> reduce bias and increase variance
-> In the ~~modern Big Data era~~ Pre DL era, not many tools were there that reduce bias or reduce variance without hurting the other one.
-> In modern Big Data era, you can train a bigger network and so, as long as you can keep getting more data is it always the case for either of these, but if it is the case, then getting the bigger net always reduces bias without necessarily hurting variance, as long as you regularize it appropriately.
-> Getting more data ~~or~~, always reduces variance and doesn't hurt bias
-> With bigger net and more data we now have tools to drive down bias or drive down variance without hurting the other thing that used. This is one of big reasons that DL is useful for supervised learning and we have a lesser tradeoff where you carefully have to balance bias and variance ~~but sometimes you got more options to reduce bias or variance without hurting the other~~.
-> Training a big net never hurts but the cost of training a NN that's too big is just computational time as long as you regularize it.

~~Regularizing your NN~~
Regularization

-> If NN is overfitting the data, we have high variance
-> ~~We can use~~ To address this we can try:
- regularization
- Get more training data, that's reliable but that's expensive and not always possible

Note: We will use Regularization on Logistic Regression

Logistic Regression: Recall the cost fnc $\min J(w,b)$, $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

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

To add regularization we add $\frac{\lambda}{2m}\|w\|_2^2$ to the cost fn.

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

L1 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

Why don't we regularise b ?

→ W is a high dimensional parameter, especially if its high variance problem.

→ W has a lot of parameters, while b is a single no.

→ If we were to add regularization to b it won't make "differe because b is only 1 parameter

L2 regularization : $\frac{\lambda}{2m}\|w\|_2^2$; $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ (L2 norm)
(most commonly used)

L1 regularization : $\frac{\lambda}{2m}\sum_{j=1}^{n}|w_j| = \frac{\lambda}{2m}\|w\|_1$ (we can use either λ/m or $\lambda/2m$, m is just a scaling constant)

→ If we use L1 reg w will end up being sparse. This means w vector will have lots of zeros in it. This can help compress the model because the set of parameters are 0, then you need less memory to store the model. (In practise it helps making model sparse only a little)

λ: regularization parameter
- This is set using hold out / validation set / dev set
- Try a variety of values for λ, see which gives the best result in terms of trading off between doing well on train set vs setting the norm of parameters to be small to prevent overfitting.

- lambda λ is another hyperparameter that has to be tuned
Note: lambda is a Python keyword

Regularization for NN

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

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

rows i.e. of the matrix should be no of neurons in current layer $n^{[l]}$
cols j of the wt matrix should equal the no of neurons in the prev layer $n^{[l-1]}$

$w : (n^{[l]} \times n^{[l-1]})$

no of hidden units in layer l and ← prev layer l-1

Computing gradient descent:

$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$ $\quad \frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$

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

⇒ with regularization coeff of w slightly < 1 in which case its called weight decay. L2 regularization is called weight decay

$$w^{[l]} := w^{[l]} - \alpha \left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]} \right]$$
$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha\, (\text{from backprop})$$

This term shows whatever wt mat $w^{[l]}$ is we are making it smaller

$$= w^{[l]} \left(1 - \frac{\alpha\lambda}{m}\right) - \alpha\, (\text{from backprop})$$

we are multiplying wt mat w with $\left(1 - \frac{\alpha\lambda}{m}\right)$ which will be <1, this is why L2 reg is called weight decay

Why does regularization prevent overfitting?

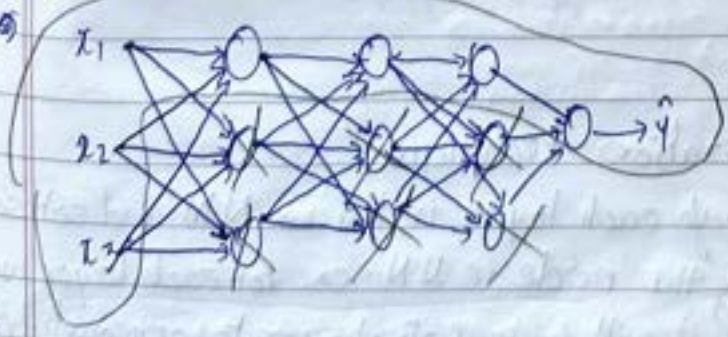
⇒ Why does shrinking the L2 norm with parameters cause less overfitting?

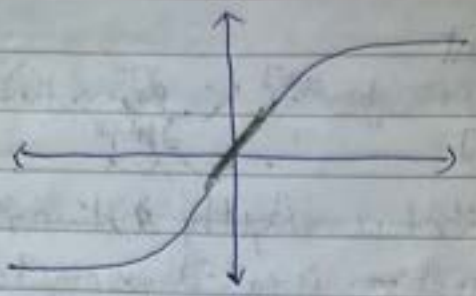
$$J(W^{[l]}, b^{[l]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

⇒ ~~if we make the regularization term~~ ~~λ~~ very

⇒ If we make the λ very big, then weight matrices W will be reasonably close to zero, ~~it will set the weights~~ effectively zeroing out the impact of hidden units. The simplified NN becomes a much smaller NN, eventually almost like logistic regression. We will end up with a small NN therefore is less prone to overfitting.

⇒ $w^{[l]} = 0$





→ Taking activation function ~~tanh~~ $g(z) = \tanh(z)$ as ex, if λ is large then weights w is small and subsequently z ends up taking relatively small values. where g and z will be roughly linear which ~~will~~ is not able to fit those very complicated decision boundary i.e less able to overfit.

- Every layer is roughly linear

Implementation tip:

When implementing regularization.

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

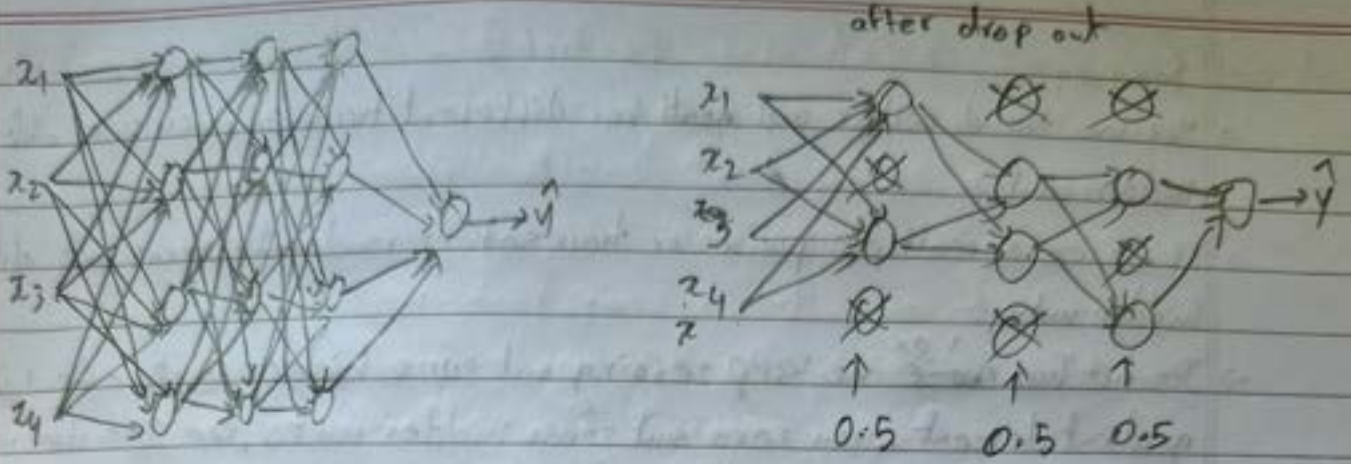
- We have $\frac{\lambda}{2m} \sum_{l} \|w\|_F^2$ which penalizes the weights being too large
- When we implement gradient descent, one of the steps to debug gradient descent is to plot cost function J as the no of ~~iterations~~ iterations for gradient descent, you want to see cost fn J decreare monotonically after every ~~iteration~~ iteration of gradient descent. Make sure you plot J with reg term, without it wont decreare monotonically

Dropout Regularization

→ Another powerful regularization technique

→ With dropout we go through each layer in the ~~NN~~ NN and set some probability for eliminating the node in NN. ex, for each layer we go through the nodes we set $p = 0.5$, it will have 0.5 chance to keep and drop the node, then we remove all ingoing and outgoing ~~x~~ connections from that node

→ After that we have a small diminished net ~~and~~ and after that we can do backprop on one ex on diminished net.



- For each train ex you would train it on the smaller neural net which has a regularizing effect

Implementing dropout (Inverted dropout):

- Illustrating with layer $l = 3$ (hidden layer) keep-prob = 0.8 (probability of keeping node) 0.2 → prob of eliminating
- Set a dropout vector, $d3$,

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

d3 will be a matrix, each ex have a hidden unit there is a 80% chance that will correspond to 1, 20% chance its 0

- We will take activations a3 in 3rd layer

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

`a3 /= keep_prob` → inverted drop out

↳ ex, we have 50 units in 3rd hidden layer

- a3 would be (50,1) or (50,m) dim
- 80% - keep, 20% - drop each unit
- We zero out 10 units

$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$ ↑ reduced by 20%

(In order to not reduce $z^{[4]}$, we can take $a^{[3]} /= 0.8$ because it roughly bumps it back to 20%. This does not change the expected value of a3)

→ Inverted dropout technique is `a3 /= keep_prob`, ensures expected value of a3 remains the same which makes test time easier because you have less scaling problem.

Summary:

→ You use the d vector and ~~you~~ for different train ex, you zero out diffrent hidden units.

→ If we make multiple passes on train set it randomly zeros out different hidden units.

→ Its not for ~~a~~ 1 ex we keep zeroing out same hidden units. On 1 iter of gradient descent, you zero out some hidden units, On 2nd iter we zero out a different pattern of hidden units.

→ vector d3 for 3rd layer decides what to zero out, only showing prob here.

Making predictions at test time:

→ At test time given some X we want to make a pred

$a^{[0]} = X$

→ ~~No~~ dropout in test time

$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = g^{[2]}(z^{[2]})$

⋮

Note: We are not using dropout in test time because while making preds we don't want o/p to be random - if we were using dropout we would be adding noise to preds. Also if we dropped out hidden units randomly its computationally inefficient and ~~gives~~ gives the same result.

Understanding Dropout

Why does dropout work?

Intuition: Can't rely on any one feature so have to spread out weights

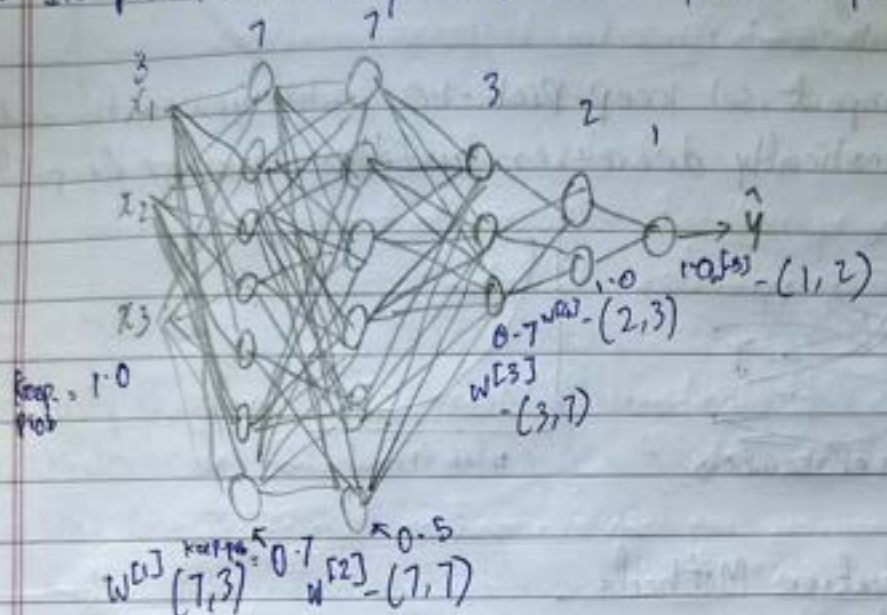
As we can see from this ex with drop out the i/p units get randomly eliminated.

→ By spreading the weights ~~it~~ it gives a bit of weight to each of 4 i/p's.

→ Spreading the weights would shrink the ~~L2~~ squared norm of weights similar to L2 regularization helping to prevent overfitting

→ L2 regularization can be applied in different ways can be more adaptative to scalable to different i/p

→ Its possible to vary keep-prob by the layer



→ $W^{[2]}$ has the largest set of parameters and has the biggest risk of overfitting for $w^{[2]}$ we might have keep-prob relatively low
• keep-prob = 0.5, different layers can have keep-prob = 0.7 (overfitting won't be a concern)

→ keep-prob is different for every layer

→ layers that have keep-prob = 1 we would drop them.

→ For layers that overfit and have a lot of parameters keep-prob is smaller and you can apply powerful form of dropout (similar to regularization)

→ We can also apply dropout to i/p layer but it isn't done often in practice, it usually uses keep-prob = 1.0 or 0.9.

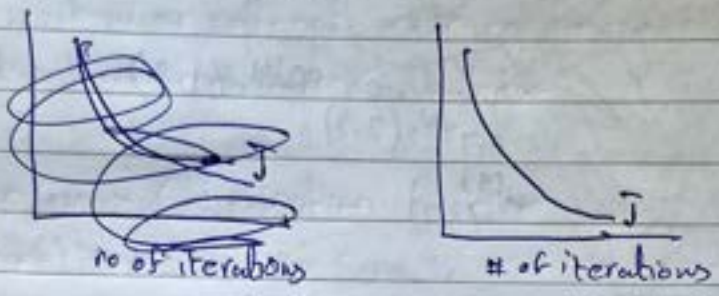
→ For layers you are more worried about overfitting, you can set lower keep-prob for some layers than others, you apply dropout - The downside is that we will more hyperparameters to search for using cross validation
- An alternative can be for some layers you apply dropout and some layers where you don't apply dropout. Then just have 1 hyperparameter to search for

→ Many of 1st successful implementations of dropout are in computer vision
- In CV the i/p sizes of pixels are too big so never have enough data prone to overfitting.

→ Downside of dropout is cost function J is no longer well defined, on every iteration we zeroing out a bunch of nodes; its harder to check performance of gradient descent.

Soln for the above:

-Turn off dropout, set keep-prob = 1.0, when you run the code, monitor J monotically decreases and then turn on dropout.



Other Regularization Methods

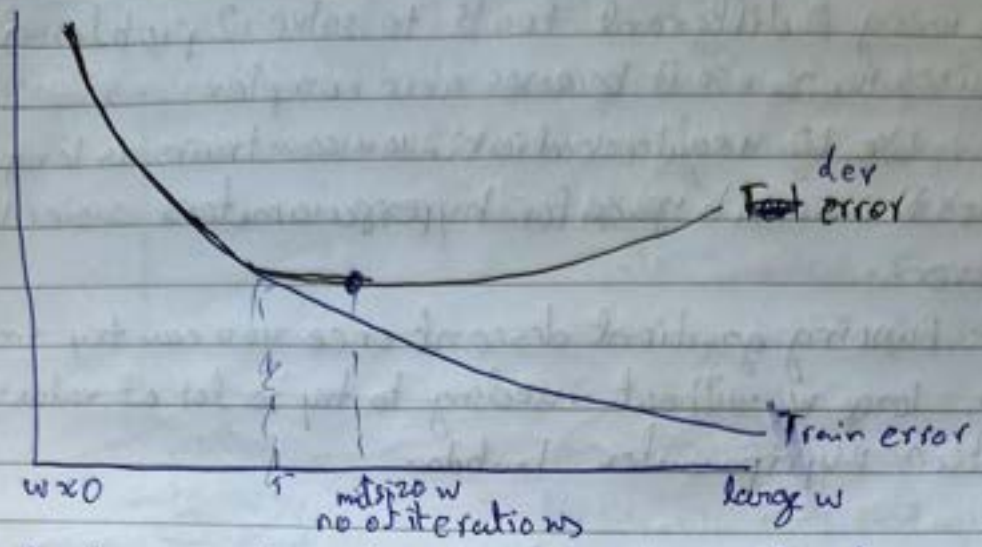
Data Augmentation

Data Augmentation:

- Getting more training data is expensive sometimes its not possible
- We can augment the train data by flipping it horizontally and we can add to our train set doubling the train set
- Train set is, however a bit redundant it isn't as good as collecting a new independent example
- We can do this without any expense,
  ex, - random flips, - randomly zooming
  - random crops
  - random distortion and translation of images
- These methods can be used to augment dataset and generate more fake training ex.
- This is an inexpensive way to give your algo more data and sort of regularize it to reduce overfitting

Early stopping:

- Ex, we run gradient descent, plot the train error J_{train} on train set it should monotonically decrease
- We can plot the dev set error



→ Early stopping stops training your neural network halfway and takes whatever dev set error at that value.

Why does it work ?

- We didn't run many iterations for the NN ~~the w parameters are close to 0~~ w x 0 because
- With random initialization w has small random values and as we iterate w gets bigger and bigger parameter values

→ Early stopping stops training midway to get mid size w.

~~Parameter Tuning~~

Orthogonalization : ML Process comprises of several steps :

- Optimize cost fn J
  - Gradient descent, RMSprop, Adam
- Not ~~over~~overfit
  - Regularization

- You have 1 set of tools for optimizing J cost fn, we only care finding w, b ~~that~~, so that J(w,b) is as small as possible. We only focus on this
- Reducing variance is a seperate task, you use a seperate set of tools for it. This principle is orthogonalization.

Downside : → It couples the tasks optimizing cost fn J and not to overfit

→ We can't work on those problems independently because by stopping gradient descent early - ~~we breaking whatever~~ it doesn't do a great job with ~~reducing~~ optimizing cost fn ~~and we~~ Additionally we are trying not too overfit

⇒ Instead of using 2 different tools to solve 2 problems you use one that mixes the 2, so it becomes more complex

⇒ Alternative: Use L2 regularization, you can train as long as possible. It also makes the search space for hyperparameters easier to decompose and search over.

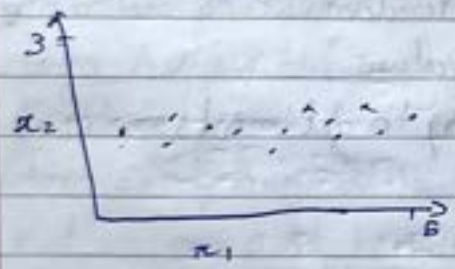
⇒ Advantage: Running gradient descent once you can try small w, mid size w, large w without needing to try a lot of values for L2 regularization hyperparameter lambda.

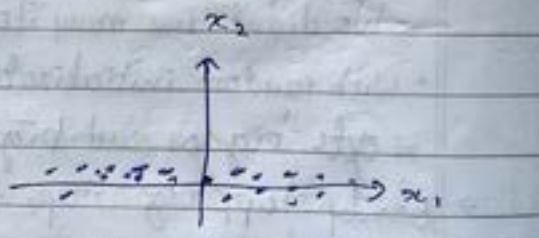
Setting Up your Optimisation Problem

Normalizing Inputs

- To speed up training we can normalize the inputs

Normalizing train sets





After subtracting mea

Normalization corresponds to 2 steps:

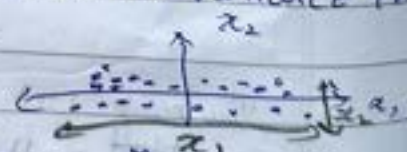
① Subtract the mean

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

$$x := x - \mu$$

② Normalize the variance

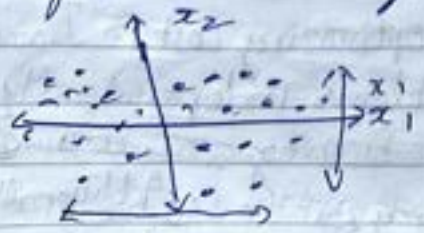
x_1 has more variance than x_2.



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

$$x /= \sigma$$

After doing ② we will get this



Variance of x_1 and x_2 is 1

Tip: If you use this μ, σ to scale training data use the same μ, σ to normalize the test data. Don't normalize train and test set seperately. We want the data to go through the same transformation on training and test samples

Why normalize inputs?

Unnormalized

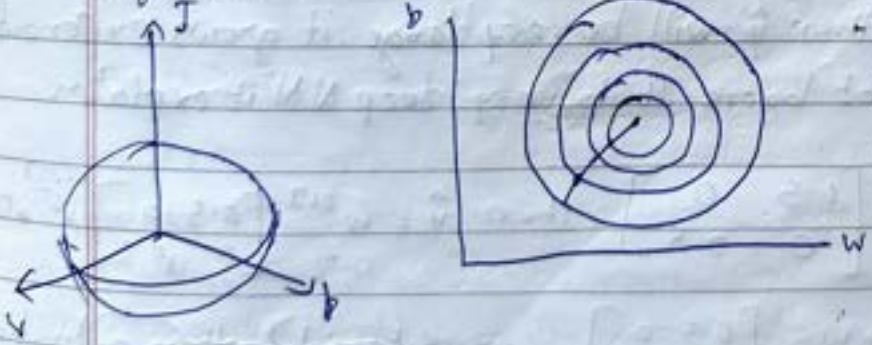
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}, y)$$

- cost fn looks very elongated
- if features are on very different scales. For ex,
  - x_1 ranges from (1, 1000)
  - $x_2 \rightarrow [0, 1]$
- Ratios for range of values of parameters for w_1, w_2 will end up taking different values

When we plot the contours of the function we can have a very elongated function.

- If we run gradient descent we need to use a small learning rate because the gradient descent might need a lot of steps to oscillate back and forth, before it finally reaches the minimum.

Normalized: if normalize features the cost fn will look more symmetric



- When we have more spherical contours the gradient descent can go straight to the minimum
- We can take larger steps for gradient descent rather than needing to oscillate

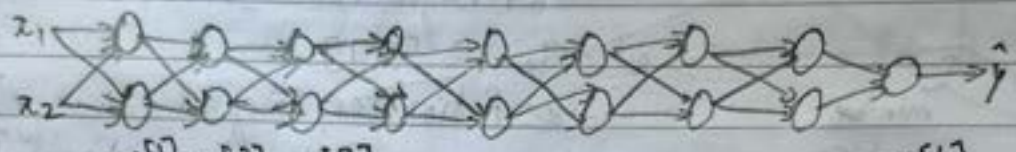
Intuition: cost function is easier to optimise when features are in similar scale. ex instead of $x_1 \rightarrow [0, 1000]$, $x_1 \rightarrow [0, 1]$ makes it easier and faster to optimise

→ We can set all of them to have 0 mean and 1 variance, sets all features to similar scale and normalizes them.

Vanishing / Exploding Gradients

⇒ When your training a deep NN your derivatives or slopes can sometimes be either very very large or small or maybe even exponentially small and this makes training difficult

⇒ ex,



$W^{[1]}$ $W^{[2]}$ $W^{[3]}$ ... $W^{[L]}$

$g(z) = z$ $\quad b^{[l]} = 0$

(N.te: we are using linear activation)

$\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \ldots W^{[3]} W^{[2]} W^{[1]} x \rightarrow a^{[3]}$

$z^{[1]} = W^{[1]} x$

$a^{[1]} = g(z^{[1]}) = W^{[1]} x$

$a^{[2]} = g(z^{[2]}) = g(W^{[2]} a^{[1]})$

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

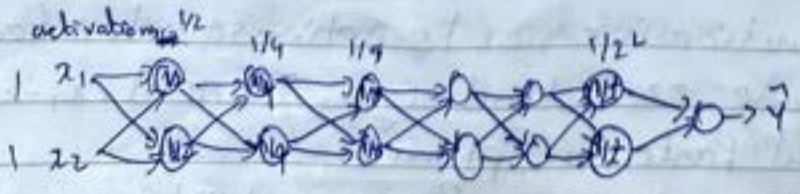
$$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{[L-1]} x = 1.5^{L-1} x$$

⇒ ~~The weights if they're little bit bigger than 1 or identity matrix then with very deep nn it will~~

⇒ if L is very high then $\hat{y}$ will be very large, it grows exponentially like 1.5 to number of layers. If its very deep NN it explodes

⇒ if we replace $W = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ then $\hat{y} = W^{[L]} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{[L-1]} x = 0.5^{L} x$

· ex, $x_1, x_2 = 1$, $W = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ (weights < 1) then activations become



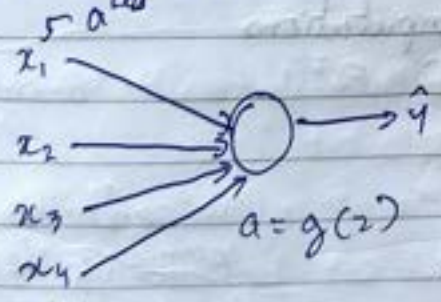
the activation values decrease exponentially

Important note: The weights W, if they are just a little bit bigger than 1 or the identity matrix then with deep NN the activations will explode. If W is little less than identity, then with deep NN, ~~they~~ the activations decrease exponentially.
- Similarly we show that the derivatives or gradients will also decrease or increase exponentially as function of no of layers.
~~=with deep NN if activations increase~~

→ In a deep NN if activations or gradients increase or decrease exponentially as a function of L then these values can get really big or really small. This makes the training difficult especially if gradients are exponentially smaller than L, ~~then~~ gradient descent takes small steps. It will take a long time for gradient descent to learn anything

Weight Initialization for Deep Networks

- A partial solution to exploding or vanishing gradient is a better or more careful choice of random initialization of weights
- ex, Single Neuron



$$Var(w_i) = \frac{1}{n}$$

larger n → smaller w

$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$ ($b = 0$)

→ In order to prevent z from blowing up and also not becoming too small, larger the n the smaller the w. We can set variance $= \frac{1}{n}$ and we can set

$W^{[l]} =$ ~~np.random.function~~ np.random.randn(shape of mat) * $np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

* no of features fed in unit at l

if we use ReLU activation we can set, $Var(w_i) = \frac{2}{n}$

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

→ So if i/p features ~~are roughly~~ have $\mu = 0$, $\sigma = 1$, $var = 1$ it would cause z to take similar scale and it doesn't solve but it reduces the vanishing / exploding gradient problems

⇒ It tries to set each of the weight matrices w so that its not too much bigger than 1 or not too less than 1. So it doesn't explode / vanish too quickly

- This is by ReLU.

Tanh Variant:

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

This is ~~the~~ called Xavier Initialization

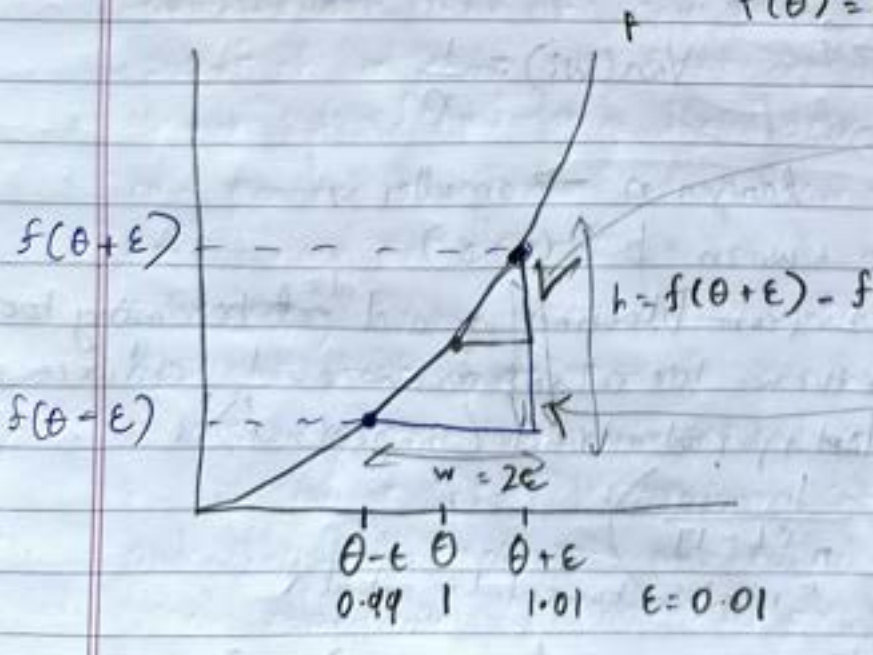
Yoshua Bengio Variant:

$W^{[l]} = np.random.randn(shape) * np.sqrt(2/(n^{[l-1]} + n^{[l]}))$

⇒ Variance can be hyperparameter $Var(w) = 1/n$ or $2/n$, but it's not as important as other hyperparameters.

Numerical Approximation of gradients

Checking your derivative computations



instead this ~~tri~~ triangle and computing slope. We can the points marked in blue and compute the slope. This slope gives a better approx of θ

$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(z)$$

$$\frac{1.01 - 0.99}{2(0.01)} = 3.001$$

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

approx error = 0.0001 from prev. slide $g(\theta) = 3.0301$

error = 0.03

→ This method for gradient checking and backprop, its run twice as slow. In practice its worth using the method because its more accurate.

Two sided difference formula is more accurate:

- Two sided case, $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ error term $O(\varepsilon^2)$
- One sided case, $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$ error $\sim O(\varepsilon)$

Goal: Numerically verify implementation of derivative of fn is correct and to check if there is a bug in backprop implementation

Gradient Checking

Gradient check for a NN:

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

concat $J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = J(\theta)$

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

concatenate

Is $d\theta$ the gradient of $J(\theta)$?

Gradient Checking (Grad Check):

J is a fn of giant parameter θ $J(\theta) = J(\theta_1, \theta_2, \theta_3, \ldots)$

To implement this:

for each i: # each component of θ

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

we get 2 vectors $d\theta_{approx} \approx d\theta$

$\approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i}$

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ $\varepsilon = 10^{-7}$ $\approx 10^{-7}$ great, 10^{-5} 10^{-3} worry (check bugs)

Gradient checking implementation notes

→ Don't use in training - only to debug $d\theta_{approx}[i]$ $d\theta$

→ If algorithm fails grad checking, look at the components and try to ~~debug~~ identify the bug.

$db^{[l]}$, $dW^{[l]}$

→ Remember regularization.

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

$d\theta$ = gradient of J wrt θ

→ Doesn't work with dropout. keep-prob = 1.0

→ ~~Random~~ Run at random initialization perhaps again after some ~~time~~ training.

$w, b \approx 0$

Week-2 - Optimization Algorithms

Mini Batch Gradient Descent

Batch vs Mini batch Gradient descent:

→ Vectorization allows you to efficiently compute on m ex.

$X = [x^{(1)}\ x^{(2)} \ldots x^{(m)}]$ (n_x, m) $\quad Y = [y^{(1)}\ y^{(2)} \ldots y^{(m)}]$ $(1, m)$

→ However if m is very large then it can be slow.

What if we have $m = 5{,}000{,}000$?

→ While implementing gradient descent on the whole train set, you can process the ~~train~~ entire train set before you take one little step of gradient descent

→ By doing this you can get a faster algorithm if you let gradient descent start to make some progress even before you finish processing the full train set of $m = 5 \times 10^6$

→ Lets say we split the training set into smaller 'baby train sets' and these baby train sets are called mini batches.

→ Each baby train set has 1000 ex.

$X = [\underbrace{x^{(1)} x^{(2)} \dots x^{(1000)}}_{X^{\{1\}}(n_x, 1000)} \mid \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}(n_x, 1000)} \mid \dots \mid \underbrace{\dots x^{(m)}}_{X^{\{5000\}}(n_x, 1000)}]$

↑ minibatch

$Y = [\underbrace{y^{(1)} y^{(2)} \dots y^{(1000)}}_{y^{\{1\}}(1,1000)} \mid \underbrace{y^{(1001)} \dots y^{(2000)}}_{y^{\{2\}}(1,1000)} \mid \dots \mid \underbrace{\dots y^{(m)}}_{y^{\{5000\}}(1,1000)}]$

~~We have~~ ~~minibatch $X^{\{t\}}, Y^{\{t\}}$~~

We have 5000 mini batches each having 1000 ex.

minibatch t: $X^{\{t\}}, Y^{\{t\}}$

$x^{(i)}$: i^{th} train ex
$z^{[l]}$: l^{th} layer of train
$X^{\{t\}}, Y^{\{t\}}$: t^{th} mini batches

→ Batch gradient descent: This is the gradient descent where we process the entire training set all at the same time. We process the entire batch of train ex all the same time.

→ Mini batch gradient descent: We process single mini batch X_t, Y_t at the same time rather than processing your entire ~~batch of~~ train set XY at the same time.

Mini batch Gradient descent:

repeat {
  For $t = 1, \dots, 5000$ {
    Forward prop on $X^{\{t\}}$
      $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
      $A^{[1]} = g^{[1]}(Z^{[1]})$
      ⋮
      $A^{[L]} = g^{[L]}(Z^{[L]})$
    Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$
    Backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)
    $W^{[l]} := W^{[l]} - \alpha dW^{[l]}$, $b^{[l]} := b^{[l]} - \alpha db^{[l]}$
  }
}

inside the for loop implement 1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$. (as $m = 1000$)

Vectorized implementation (1000 ex)

we are doing the same work as before

l : no of training ex

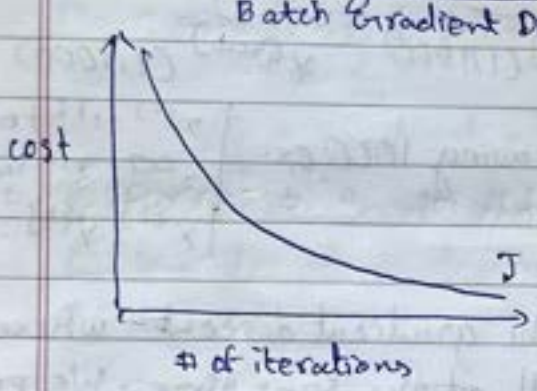
from mini batch $(X^{\{t\}}, Y^{\{t\}})$

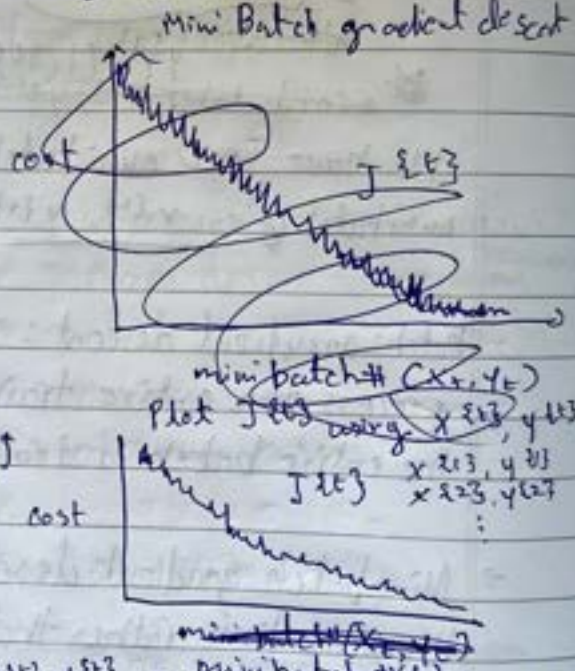
"1 epoch": 1 pass through the training set

→ With batch gradient descent single pass through train set allows you to take only # 1 step in gradient descent.

→ Mini batch gradient descent a single pass through the train set, 1 epoch allows you to take 5000 gradient descent steps.

Understanding Mini Batch Gradient Descent





→ On Batch gradient descent the cost fn J should decrease on every iteration

→ On mini batch gradient descent:
- when we plot cost fn $J^{\{t\}}$, it uses $X^{\{t\}}, Y^{\{t\}}$, $J^{\{t\}}$ computed using $X^{\{t\}}, Y^{\{t\}}$ then on every iteration we are training on a different train set, or mini batch.
- The trend is downward but noisy, because maybe $X^{\{1\}}, Y^{\{1\}}$ is relatively easy mini batch so cost might be a bit lower; and $X^{\{2\}}, Y^{\{2\}}$ mislabeled examples harder batch, cost will be higher. So we get these oscillations while running mini batch gradient descent.

Choosing your mini batch size:

If mini batch size = m : Batch gradient descent $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

If mini-batch size = 1 : Stochastic gradient descent every ex has its own mini batch
$(X^{\{1\}}, Y^{\{1\}}) \rightarrow (X^{(1)}, Y^{(1)}) \ldots (X^{(2)}, Y^{(2)})$

In practice : Somewhere between 1 and m

too small ↑ ↑ too large

ex the cost function we are trying to minimise on the contours of

→ : batch gradient descent
→ : stochastic gradient descent
→ : somewhere between [1, m]

→ Batch gradient descent can start somewhere and can take relatively low noise, relatively long steps. It goes towards the minimum.

→ Stochastic gradient descent starts at some point. On every iteration of gradient descent with a single ex, most of the times you hit global minimum. Sometimes it'll head in wrong direction if that ex points in bad direction.

→ Stochastic gradient can be extremely noisy and on an avg it can take you to a good or wrong directions. It doesn't converge, it oscillates and wanders around the region of minimum

→ Why do we use batch size → [1, m] ?

- If we use batch gradient descent, minibatch size = m. We are processing a huge training set on every iteration. Disadvantage is it will take too long per iteration, i.e we have a big train set. If we have a small train set then batch gradient descent is fine.
- If we use stochastic gradient descent, we make progress after 1 ex. The noisiness can be reduced by just using a smaller learning rate. Disadvantage is we lose almost all the speed up from vectorization because we are processing a single ex at a time which is inefficient.
- In practise if we use mini batch_size → [1, m], where minibatch size is not too big or too small it gives you faster learning. We get a lot of vectorization ex, mini batch_size = 1000 we can vectorize across 1000 ex which is going to be much faster than processing 1 ex at a time. We can also make progress without needing to wait to process the entire train set. It doesn't guarantee to always head towards the minimum but it tends to head more consistently in direction of minimum than consequent descent. It doesn't always oscillate in a very small region, for this we can reduce learning rate.

Guidelines: ~~to choosing [1, m] for mini batch size~~:

→ If train set is small, use batch gradient descent ($m \leq 2000$)

→ Bigger train set: Typical mini batch size → 64, 128, 256, 512
2^6 2^7 2^8 2^9

→ Make sure mini batch fit in CPU / GPU memory.
$X^{\{t\}}, Y^{\{t\}}$

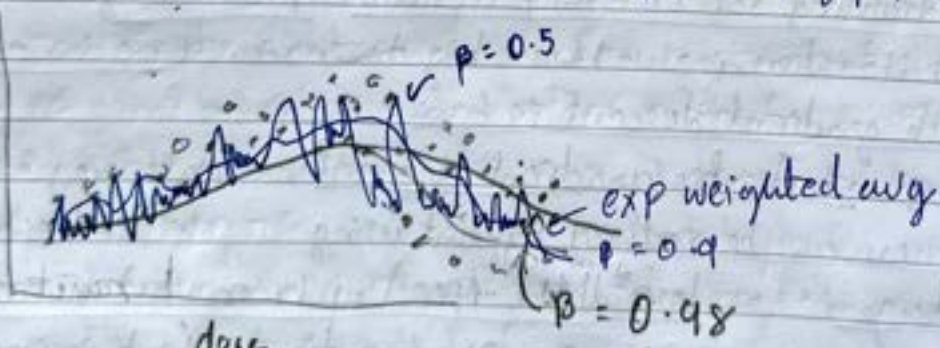
Exponentially weighted Averages or Exponential weighted moving Avg

ex Temp in London

$\theta_1 = 40°F$ 4°C
$\theta_2 = 49°F$ 9°C
$\theta_3 = 45°F$
⋮
$\theta_{180} = 60°F$ 15°C
$\theta_{181} = 56°F$
⋮

approach,

$V_0 = 0$
$V_1 = 0.9 V_0 + 0.1(\theta_1)$
$V_2 = 0.9 V_1 + 0.1(\theta_2)$
$V_3 = 0.9 V_2 + 0.1(\theta_3)$
⋮
$V_t = 0.9 V_{t-1} + 0.1(\theta_t)$



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

$\beta = 0.9 \approx$ 10 days temperature

We can think of V_t as averaging over $\approx \frac{1}{1-\beta}$ days temp

$\frac{1}{1-0.9} = 10$

$\beta = 0.98$ $\quad \frac{1}{1-0.98} = 50$

→ As β is increased we get a smoother plot since we are averaging more days of temperature. curve shifted to right

latency: time delay bet the moment an i/p is provided to a model and model produces output

⇒ By averaging over a larger window the exp weighted avg formula adapts more slowly when temperature changes. So there is a bit more latency
- When $\beta = 0.98$, it gives a lot of weight to the previous value and smaller weight 0.02 to the current value.
- When $\beta = 0.5$, $1/1-0.5 = 2$ days
its like avging over just 2 days temp. ~~Ita~~ averaging over a shorter window. Its more noisy and more susceptible to outliers. But it adapts more quickly to temp changes

Understanding exponentially weighted avg

$V_t = \beta V_{t-1} + (1-\beta)\theta_t \quad \beta = 0.9$

~~$V_{100} = 0.1$~~

$V_{100} = 0.9 V_{99} + 0.1 \theta_{100}$

$V_{99} = 0.9 V_{98} + 0.1 \theta_{99}$

$V_{98} = 0.9 V_{97} + 0.1 \theta_{98}$

...

$V_{100} = 0.1\theta_{100} + 0.9 V_{99}$

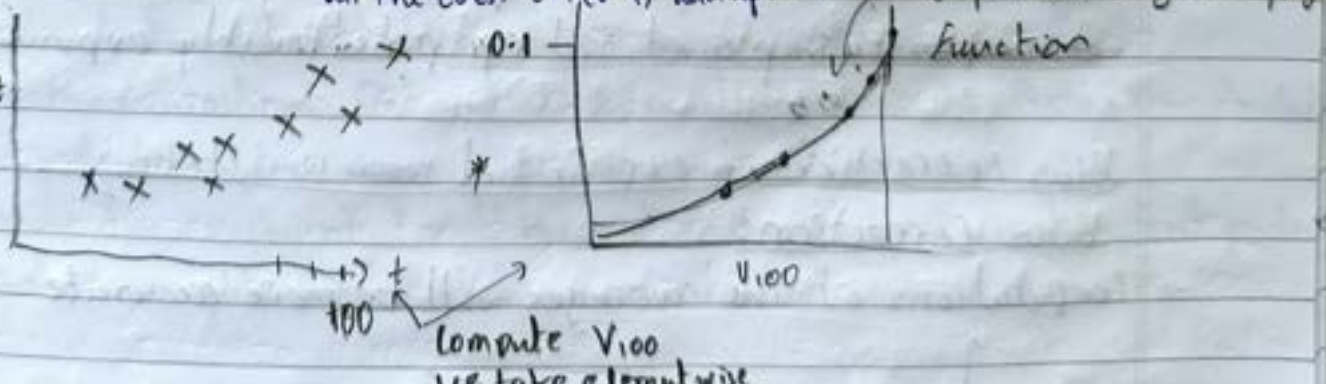
$V_{99} = 0.9 V_{98} + 0.1\theta_{99}$

$= 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9 V_{98})$

$V_{98} = 0.1\theta_{98} + 0.9 V_{97}$

$= 0.1\theta_{100} + 0.1(0.9)\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + 0.1(0.9)^4\theta_{96} + \dots$

Visualizing this



compute V_{100} we take element wise product these 2 and sum it

How many days are we averaging over?

$0.9^{10} \approx 0.33 \approx 1/e$ $\quad (1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$, $\varepsilon = 0.1$

10 days

Implementing exp weighted avg :

$v_0 = 0$

$v_1 = \beta v_0 + (1-\beta)\theta_1$

$v_2 = \beta v_1 + (1-\beta)\theta_2$

$v_3 = \beta v_2 + (1-\beta)\theta_3$

⋮

$v_\theta := 0$

$v_\theta := \beta v + (1-\beta)\theta_1$

$v_\theta := \beta v + (1-\beta)\theta_2$

⋮

$v_\theta = 0$

Repeat {

  Get next θ_t

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

}

Advantages :

→ takes very little memory, you just need to keep just 1 row no in computer to compute exp wa memory, and override it with the formula - $v_\theta := \beta v_\theta + (1-\beta)\theta_t$ based on the latest values.

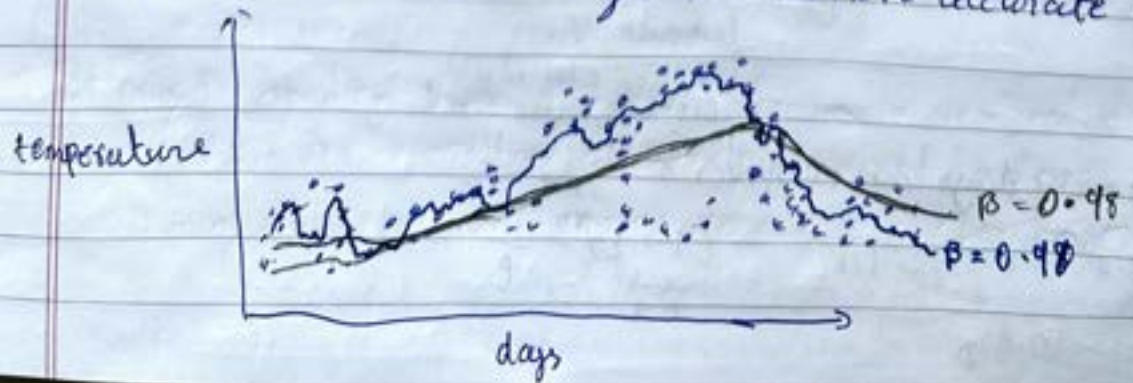
Disadvantage :

→ Not really the best or most accurate way compute an avg. or. if we compute the moving window for the last 10 days, we divide by 10. that usually gives better estimate. But the disadvantage that is of explicitely keeping temp around and sum of the last 10 days it requires more memory, this more complicated ~~and, complex~~ to implement and computationally expensive.

Bias correction in exponential ~~mean~~ weighted avg

Bias Correction:

→ Computations of the averages will be more accurate - Bias Correction



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

→ When we use $\beta = 0.98$ we won't actually get the black curve we get blue color

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

$$V_0 = 0$$

$$V_1 = 0.98 V_0 + 0.02\theta_t$$

$$V_2 = 0.98 V_1 + 0.02\theta_2$$

$$= 0.98(0.02)\theta_1 + 0.02\theta_2$$

$$= 0.0196\theta_1 + 0.02\theta_2$$

$$\frac{V_t}{1-\beta^t}$$

$$t = 2 : 1-\beta^2 = 1 - 0.98^2 = 0.0396$$

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

assuming θ_1, θ_2 are +ve no's, V_2 is not a very good estimate since it'll be less than θ_1, θ_2.

→ Assuming θ_1, θ_2 are +ve no's, V_2 is not a good estimate because its less than θ_1 and θ_2. There is a way to modify the estimate which makes it much better and accurate shown on the right.

→ This becomes the weighted avg and removes the bias.

→ As t becomes large, β^t approaches 0;

→ When t is large enough bias correction makes no difference. this line and this line overlap.

→ During the initial phase of learning when your warming up your estimates, bias correction helps you obtain a better estimate of your temp.

Gradient Descent with Momentum

→ Momentum / Gradient Descent with momentum works faster than standard gradient descent.

→ Compute the exponential weighted avg of the gradients and use that gradient to update weights instead.

3:50

Gradient Descent example:

ex, we are optimizing a cost fn that has contours.

↕ slower learning

↔ faster learning

→ if we take an iteration of batch or mini batch gradient descent

→ The gradient descent takes a lot of steps and slowly oscillate towards the minimum.

→ Up and down oscillations prevent you from using a large learning rate. It might overshoot and end up diverging. so to prevent oscillation it forces you to use not too large learning rate.

→ Another way to view this problem is that on vertical axis you want α a bit small because we don't want oscillations and on horizontal axis we want faster learning

Momentum:

On iteration t:

Compute dW, db on current mini batch

$V_{dw} = \beta V_{dw} + (1-\beta)\, dw$ "$V_\theta = \beta V_{\theta-1} + (1-\beta)\theta_t$

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

$W := W - \alpha V_{dw}$, $b := b - \alpha V_{db}$

→ If we avg the gradients the oscillations in vertical direction tends to avg to 0. In the vertical direction, you want to slow things down this will avg out +ve and -ve no's, so avg is closer to 0.

→ On the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the avg in horizontal direction will be pretty big

→ In this algo, with a few ~~steps~~ iterations we will find that gradient descent with momentum, ends up eventually taking smaller oscillations in vertical direction, but more directed to just moving quickly in horizontal direction. This allows the algorithm to take a more straightforward path or to damp out the oscillations in the path to minimum.

Momentum intuition: (Optional)

if we try to minimise a bowl shaped fn

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

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

* friction

velocity

acceleration

→ The derivative terms dW, db can be thought as providing acceleration to a ball you're rolling downhill

$W := W - \alpha V_{dw}$

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

→ Imagine taking a ball and little ball is rolling down the hill, it imparts acceleration to the ball

⇒ Rather than gradient descent just taking every single step independently of all previous steps. Little ball can roll downhill and gain momentum it can accelerate down the bowl and gain momentum.

Implementation details

$V_{dw} = 0$, $V_{db} = 0$

On iteration t:

Compute dW, db on the current mini-batch

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

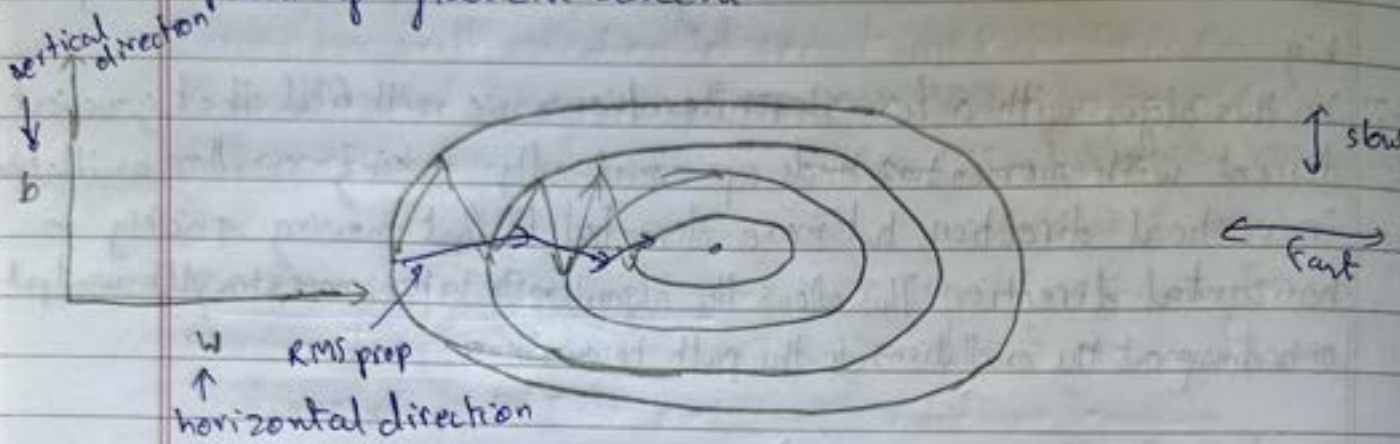
$V_{db} = \beta V_{db} + (1-\beta) db$

$W = W - \alpha V_{dw}$, $b = b - \alpha V_{db}$

Hyperparameters: α, β $\beta = 0.9$

RMSprop (Root Mean Square prop)

→ Speeds up gradient descent



→ We want to slow down learning in the b direction or vertical direction. Speed up learning in the horizontal direction

→ RMS prop does this on iteration t

On iteration t:

Compute dW, db on current mini batch

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2$ ← element wise square ← small

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

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}}}$ $\quad b := b - \alpha \frac{db}{\sqrt{S_{db}}}$

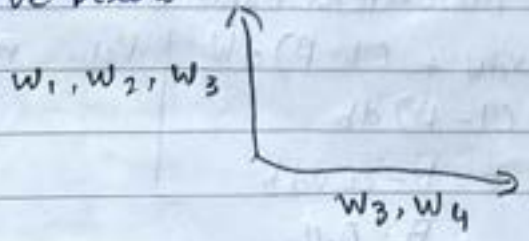
→ We want S_{dw} to be small and S_{db} to be large

→ The net effect is updates in vertical direction (dw) are divided by larger no that helps dump out oscillations. Updates in horizontal direction (db) are divided by a smaller no.

→ Updates in horizontal and vertical direction keep going

→ Use larger α to get faster learning without diverging in vertical direction

→ In practice we are in a very high dimensional space of parameters observe below



⇒ In practice dW and db are high dimensional parameter vectors. But the idea is that in dimensions you're getting these oscillations, you end up computing a larger sum.

⇒ A weighted avg for these squares and derivatives we damp out the directions in which there are oscillations

⇒ Hyperparameter : β_2

⇒ What if $\sqrt{dW}$ is too small?

- ~~Things could~~ Things ca blow up
- Add a very small ε to deromination for ensuring numerical stability

⇒ RMSprop is similar to momentum, has the effects of damping out oscillations in gradient descent, in minibatch gradient descent. May be allow you to use larger α and speeds up your learning speed of algo.

Adam Optimization algorithm

⇒ Adam algo is a ~~combination of~~ puts RMSprop and momentum together

Implementation

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

On iteration t:

Compute dW, db using current minibatch

momentum β_1

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dW$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

RMSprop β_2

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

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

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

$$W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \qquad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Hyperparameters choice

-> α: needs to be tuned

-> β_1 : 0.9 (dW)

-> β_2 : 0.999 (dW^2)

-> ε : 10^{-8}

Adam: Adaptive Moment for estimation

- Combines the effect of gradient descent with momentum together with gradient descent with RMSprop.
- → Commonly used learning algorithm, proven to be effective many different NN's of a wide variety architectures

Learning rate Decay

ex we implement mini batch gradient descent with small mini-batches of 64, 128 ex.

Slowly reduce α

- As we iterate the steps are noisy, tending to the minimum and it wont converge because we are using a fixed α there is noise in different batches
- We can slowly reduce α during the initial ~~phases~~ step of learning, so we can take much bigger steps but then as learning approaches convergence, then having slower learning rate allows you to take smaller steps.

Implementation

1 epoch = 1 pass through data

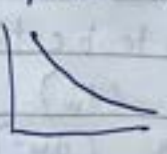
$$\alpha = \frac{1}{1 + \text{decay Rate} \times \text{epoch number}} \alpha_0$$

α_0: initial learning rate

epoch	α
1	0.1
2	0.067
3	0.05
4	0.04

$\alpha_0 = 0.2$

decay rate = 1



→ Hyperparameters : α_0, decay rate

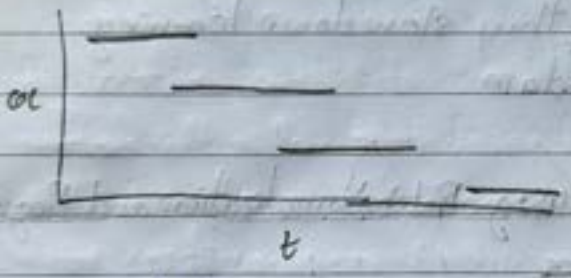
Other learning rate decay methods:

$\alpha = 0.95^{\text{epoch\_num}} * \alpha_0$ ← exponential decay

$\alpha = \frac{k}{\sqrt{\text{epoch num}}} \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$ ← Polynomial decay

Piecewise constant

discrete staircase



manual decay

The Problem of local Optima

→ Algorithms get stuck in local optimum rather than finding its way to global optimum.

→ In NNs most points of zero gradients are not local optima.

→ Most points of zero gradients in cost function are called saddle points.

→ A function of high dimensional space if gradient is zero then in each direction it can be a convex light function or concave light function

convex ∪ concave ∩

→ We are more likely to run into saddle points in high dimensional spaces.

→ Derivative is zero at saddle points

Problem of Plateaus

Problem of Plateaus :

- Plateaus slow down learning
- Region where the derivative is very close to 0 for long time

Key Points :

→ You won't get stuck at local optima as long as your training a reasonably large NN; save a lot of parameters and cost function J is defined over a relatively high dimensional space

→ Plateaus are problem and they slow down learning.

Soln: Use RMSProp or Adam

Week-3 : Hyperparameter tuning, Batch Normalization and Programming Frameworks

Hyperparameter Tuning

Tuning Process

Hyperparameters :

1. α (learning rate)
2. $\beta \approx 0.9$ (momentum term) 0.9 is usually a good estimate
3. $\beta_1, \beta_2, \varepsilon$ (Adam optimization) → we never fine tune this $\beta_1 = 0.9$, $\beta_2 \approx 0.999$, $\varepsilon \approx 10^{-8}$ use these by default.
4. no of layers
5. no of hidden units
6. learning rate decay
7. minibatch size

Most important hyperparameters to finetune : (order of importance)

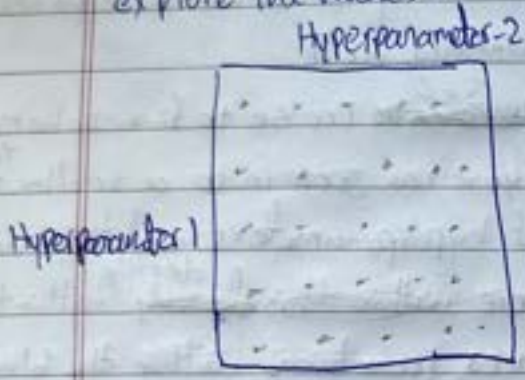
1. → 1. α
2. 2. β
3. 3. no of hidden units
4. 4. Minibatch size
5. 5. no of layers
6. 6. learning rate decay

How do we select a set of values to explore?

-> ~~In earlier ML algorithms, its a common practice sample points in a grid~~

- Try random values; Don't use a grid

- In earlier ML algorithms, if we had 2 hyperparameters, hyperparameter-1 and hyperparameter-2. We sample the points on a grid and systematically explore the values.



-> We are placing in a 5×5 grid in this ex all 25 points and then we pick whichever hyperpa that works best. This works only if we have a small no of hyperparameters

- In RL choose points at random, ~~we choose the~~

-> Choose 25 points

-> Try out hyperparameters on the randomly chosen set of points.

-> Reason why we do this is because its difficult to know in advance which hyperparameters are going to be the most important for the problem.

-> Some hyperparameters can be more important than others

-> ex, hyperparameter-1 is α and hyperparameter-2 is ε. The choice of α matters more than ε. If we used sampling from grid, tried 5 diff values of α and we will find 5 values of ε gives you the same answ

- We ~~had~~ trained 25 models and only got ~~into~~ 5 trial values for α

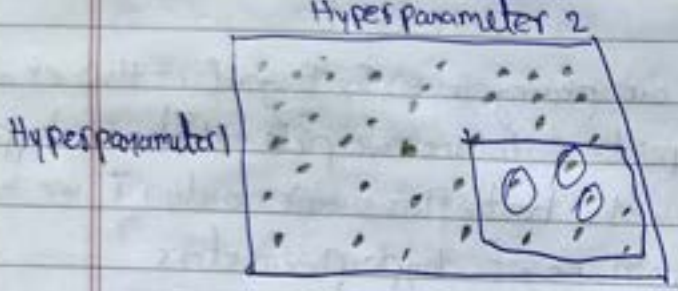
- If we tried sampling at random then we tried 25 distinct values of α and you are more likely able to find a value that really works well

-> In practice we might search over more than 3 hyperparameters, its hard to know in advance which are more important for applicato. ~~so sampling in rand~~

→ Sampling in random rather than grid allows to richely explore the set of possible values for the most important hyperparameter

Another common practice to find hyperparameters:

Coarse to fine



- Lets say we found these points working well
- In Coarse to fine scheme you can zoom in to a smaller region of hyperparameters and then sample more densely within this space

or maybe at random but focus more resources searching within the blue space if we think its a best setting, the hyperparameters maybe in this region

- We coarse the sample of the entire square that tells you to focus smaller square, we sample more densly in smaller square.

→ By trying diff values of diff hyperparameters you can pick whatever value allows you to do best on your training objective or dev set or any hyperparameter your trying to optimise in the search process.

Using an appropriate scale to pick hyperparameters

Picking hyperparameters at random:

- $n^{[2]} = 50, \ldots, 100$

50 ——— 100

→ For no of hidden units we can pick a no at random to search for the particular hyperparameter.

- no of layers L : 2 - 4

→ Sampling uniformly at random 2, 3, 4 might be reasonable

→ Grid Search to evaluate the values 2, 3, 4

→ The above examples is where we can apply sampling at random over the range you contemplating is reasonable. But this is not true for all hyperparameters

Appropriate scale for hyperparameters:

ex, we search for α:

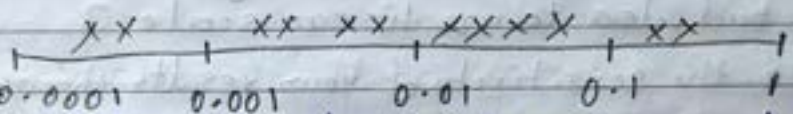
$\alpha = 0.0001, \ldots, 1$

We sample the values uniformly at random

90% of values we sample → - This means 90% of the resources are used to search between 0.1 to 1.

- 10% of resources to search between 0.0001 to 0.1
- This isn't right.

- ✱ Use log scale to search for hyperparameters instead of linear scale.



- Using sampling at random on logarithmic scale ✱ we have more resources dedicated to searching between $0.0001 - 0.001$, $0.001 - 0.01$, $0.01 - 0.1$, $0.1 - 1.0$

Implementing in Python:

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

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

ex if we are sampling between 10^a and 10^b on log scale

$10^a = 0.0001$ ✱ $10^b = 1.0$

$a = \log_{10} 0.0001$ $b = \log_{10} 1.0 = 0$

$a = -4$ $b = 0$

$r = 10$ ← After this sample uniformly at random $r = [-4, 0]$

Hyperparameters for exponentially weighted avg:

ex, $\beta = 0.9 \ldots\ldots 0.999$

We avg over 10 values — we avg over 1000 values

0.9 — 0.999

0.9 — 0.99 — 0.999

$1 - \beta = 1 - 0.9 = 0.1$

$0.1 \ldots 0.001$ — 0.1 — 0.01 — 0.001

$r \in [-3, -1]$ — 10^{-1} — 10^{-3}

$1 - \beta = 10^r$

$\beta = 1 - 10^r$

- We spend an ~~same~~ amt of resources exploring from 0.9 - 0.99 and 0.99 - 1
⇒ Why is it a bad idea to use linear scale ?
- When $\beta \approx 1.0$, the sensitivity of your results changes even with very small changes to β. If β goes from 0.9 to 0.9005, doesn't change the results. If β goes from 0.999 to 0.9995 it will have a huge impact on what the algorithm is doing.

⇒ $\frac{1}{1-\beta}$ formula is sensitive to small changes in β, when $\beta \approx 1$, so what the whole sampling process does is it causes you to sample more densly in the region where $\beta \approx 1$ or ~~β~~ $1 - \beta \approx 0$.

- This will be more efficient on how distribute the samples to explore the space of possible outcomes more effectively.

Hyperparameters Tuning in Practice : Pandas vs Caviar:

Retest hyperparameters occasionally:

⇒ There is a lot of cross fertilization over diff domains. ex ConvNet, ResNet in CV ~~ed~~.

Idea → Code → Experiment

⇒ People from diff application domains read research papers from other application domains to look for inspiration for cross fertilization

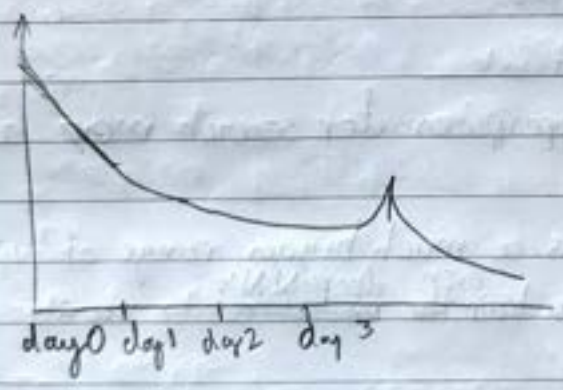
⇒ Intuitions do get stale. Re-evaluate occasionally

er, we have a logistics problem:
- we have a good set of hyperparameters, kept developing the algorithm or maybe ~~we have~~ the data gradually changes over time because of that hyperparameters get stale
- Retest hyperparameters every several months

How do we search for hyperparameters?
There are 2 approaches:

1. Babysitting one model: (Panda)

→ Usually done when we have huge dataset but not a lot of computational resources to train a lot of models at the same time.
→ Not a lot of CPU and GPU.
→ You can afford training only one model or a very small no of models at a time
→ We will babysit the model as its training.



Day 0: Random parameter initialization and start training. Gradually see the learning curve, cost fn J decreasing gradually
Day 1: Learning is ~~good~~ good. We try increasing α a little bit and see how it goes
Day 2: goes well, may ~~increase~~ decrease the momentum term. or decrease α.

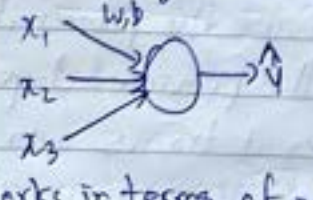
→ Every day we will observe and nudge up or down the parameters
→ Maybe one day, α is too big we go back to the previous days model.
→ We Babysit the model one day at a time as it trains over a several days or weeks.

2. Training many models parallely (Caviar)
→ Ideal if we have enough computing resources
→ Simultaneously train models with varied hyperparameters
→ Each model produces a distinct learning curve
→ At the end of training, compare the performance and select the best model.

Choosing Between approaches
→ Best approach depends on computational resources
→ Caviar approach is preffered when abundant computing resources are available to test multiple hyperparameters at once.
→ In scenarios with massive datasets and resource intensive models the panda approach is preferred where one invests significantly in fine-tuning a single model might be more feasible.

Batch Normalization
Normalizing Activations in a network
→ Batch Normalization makes hyperparameter search more easier
→ Makes NN more robust
→ The choice of hyperparameters is a much bigger range of hyperparameters that work well and allows you to train even deep NN's easily.

Normalizing inputs to speed up learning:
ex, In logistic regression we normalize input features to speed up learning

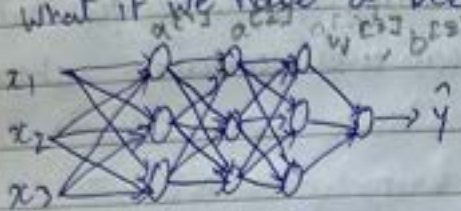


$$\mu = \frac{1}{m}\sum x^{(i)}$$
$$X = X - \mu$$
$$\sigma^2 = \frac{1}{m}\sum x^{(i)2}$$
$$X = X / \sigma$$

This works in terms of normalizing the I/P features values into NN after regression

→ This makes contours of gradient descent from elongated to more round makes with easier for gradient descent to optimize

What if we have a Deep NN ?



- we have i/p features and activations
- ~~if we want to normalize~~

Can we normalize $a^{[2]}$ so as to train $w^{[3]}$, $b^{[3]}$ faster ?

→ $a^{[2]}$ is i/p to next layer it affects training of $w^{[3]}, b^{[3]}$

→ Batch Norm will normalize the values of ~~$a^{[2]}$~~ $z^{[2]}$ not $a^{[2]}$.

→ In practice normalizing $z^{[2]}$ is done more often.

Implementing Batch Norm :

Given some intermediate values in NN $z^{(1)} \dots z^{(m)}$ ↖ $z^{[l](i)}$

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

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

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

↖ z will have 0 mean and 1 std dev

tilde → $\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$ ← learnable parameters of model

We use this for hidden units (we don't want hidden units to have 0 mean and 1 std dev because hidden units should have different dist)

If

$\gamma = \sqrt{\sigma^2 + \varepsilon}$

$\beta = \mu$

then $\tilde{z}^{(i)} = z^{(i)}$

Update Param : γ, β

We use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$

→ BatchNorm applies this type of normalization process not only to i/p layer ~~to normalize mean~~ but even some values deep in some hidden ~~and variance of some hidden units values~~ ? layer in NN. It applies this normalization to normalize mean and variance of some hidden units

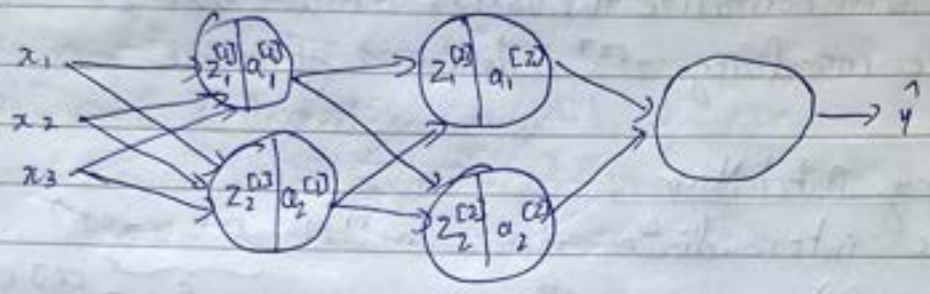
→ Difference bet training i/p ~~and~~ and hidden unit $z^{(i)}$ is we don't want our values to be forced to have mean 0, var 1. ex if we have sigmoid we don't want the values to be clustered there. You might want them to have larger variance or mean that's different from 0 in order to take advantage of non linearity of sigmoid rather than having all values in linear region

Parameters

→ γ and β make sure your $z^{(i)}$ values have the range you want. It ... hidden units have standardized mean and variance, where they ... controlled γ and β.

Fitting Batch Norm into a NN

Adding BatchNorm to a network:



$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow \dots$$

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[l]}, b^{[l]}$
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[l]}, b^{[l]}$
← Update parameters using optimization algo Adam, RMSProp etc

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

we can implement in tensorflow like,
tf.nn.batch_normalization

Working with mini batches:

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

$$X^{\{2\}} \longrightarrow z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow \dots$$

Parameters: $w^{[l]}$, ~~$b^{[l]}$~~, $\beta^{[l]}$, $\gamma^{[l]}$
$(n^{[l]}, 1)$ $(n^{[l]}, 1)$ $(n^{[l]}, 1)$

$z^{[l]}$
$(n^{[l]}, 1)$

$z^{[l]} = w^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$
$z^{[l]} = w^{[l]} a^{[l-1]}$
$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$

classmate
Date
Page

Implementing gradient descent :

for t = 1 ... num MiniBatches
Compute forward prop on $X^{\{t\}}$
In each hidden layer use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
Use backprop to compute $dw^{[l]}, db^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$
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 RMSProp, Momentum, Adam

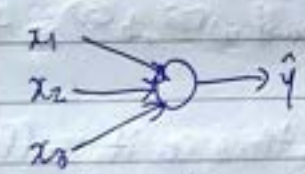
Why does BatchNorm work?

→ Batch Norm normalizes all input features X to take a similar range of values to speed up learning. It does this for hidden units and input features.

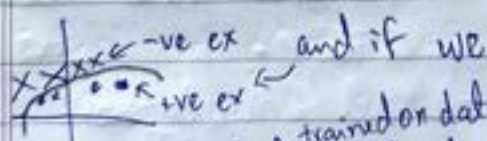
Learning on shifting input distribution :

→ It makes weights that are deeper in NN more robust to changes in weights to earlier layers

ex. we train a NN on cat detection (Cat $y = 1$, Non Cat $y = 0$)



we train the clf on images of black cat and if we test on colored cats the clf might not do very well · ex if our train set was like this



and if we tried to generalize this to dataset where

trained on data on the left

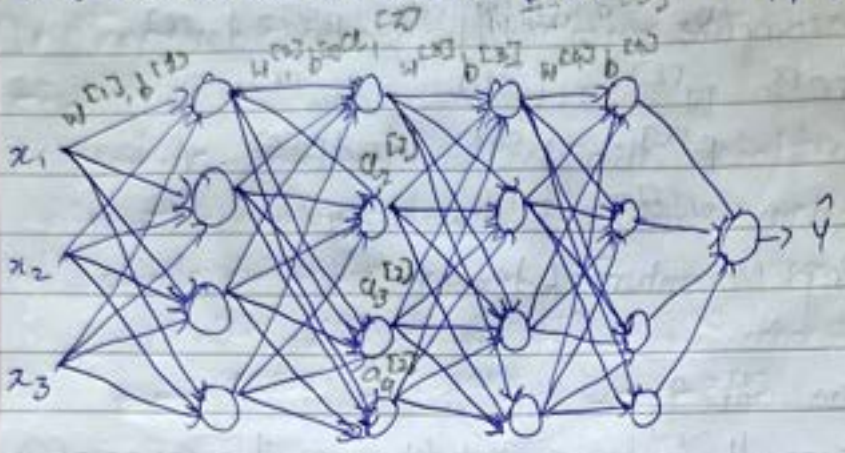
then the model won't do well on the data on the right even though they are same function that works well

→ The idea of data distribution changing is called covariate shift. if you learn $X \to Y$ mapping, if the dist of X changes then we need to retrain the learning algo.

→ This is true when the mapping of $X \to Y$ doesn't change

→ ~~Retrain the function~~ The need to retain your function becomes worse if mapping $X \to Y$ changes

How does covariate shift problem apply to NN's?

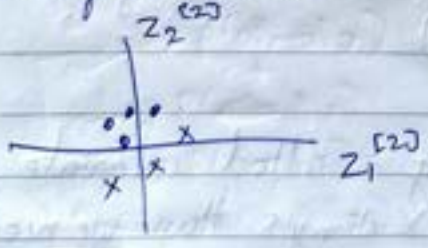


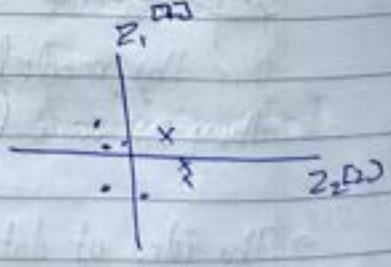
From the perspective of 3rd layer,

- The input features are $a_1^{[2]}, a_2^{[2]}, a_3^{[2]}, a_4^{[2]}$
- The job of the 3rd hidden layer is to take the i/p features, and find a way to map them to $\hat{y}$.
- Gradient descent learns parameters so $w^{[3]}, b^{[3]}, w^{[4]}, b^{[4]}$ so that the nt does a good job mapping from $X \to \hat{y}$
- The net also adapts these values $w^{[1]}, b^{[1]}$ and $w^{[2]}, b^{[2]}$ so the values of $a_1^{[2]}, \ldots$ change
- The hidden unit values keep changing all the time so it suffers from the problem of covariate shift.

→ BatchNorm reduces the amount of distribution these hidden unit values shift around

Plotting dist values





- The values for $z_1^{[2]}, z_2^{[2]}$ will change when parameters get updated in the earlier layers. Batch Norm ensures mean and variance of $z_1^{[2]}, z_2^{[2]}$ remains the same

- mean=0, variance=1 or whatever value governed by $\beta^{[2]}, \gamma^{[2]}$
- This limits the amount that updating parameters in the earlier layers can affect the distribution of values that 3rd layer sees.
- Batch Norm reduces the problem i/p values changing, it makes the values become more stable, so later layers will have a more firm ground to stand on
- If i/p dist changes a little bit ~~and~~, even as earlier layers keep learning the amounts that forces this later layers to adapt to ~~changes~~ earlier layers changes is reduced or if we will, it weakens the coupling between what the early layers parameters has to do and what later layer parameters has to do.
- It allows each layer of the net to learn by itself a little bit more independently of others layers and this speeds up learning in the whole net

→ Batch Norm from the perspective of later layers in NN, the earlier layers don't get shifted as much because they are constrainted to the same mean and variance and ~~this makes the job~~ of learning on later layers become easier

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 mini batch. So similar to dropout it adds noise to each hidden layers activations

→ This has a slight regularization effect.

Batch Norm at test time

Recall:

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

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

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

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

At test time we may not have mini batches to process at the same time

μ, σ^2 : estimate using exp weighted avg

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

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

$\theta_1 \quad \theta_2 \quad \theta_3$ $\quad \sigma^2$

$\sigma^{2\{1\}[l]} \quad \sigma^{2\{2\}[l]}$

$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \tilde{z} = \gamma z_{norm} + \beta$$

→ During training time μ and σ^2 are computed on the entire batch of 64 ex

→ In test time we process one ex at a time, we can do that by estimating μ, σ^2 from train set.

→ In theory we can run whole train set through the final net to get μ, σ^2.

→ In practice, you can implement exp weighted avg where you keep track of μ, σ^2 and ~~exp wt avg~~ sometimes we call this running avg to get rough estimate of μ, σ^2 and then use these to scale at test time

$\mu^{\{3\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]} \rightarrow \mu^{[l]}$

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

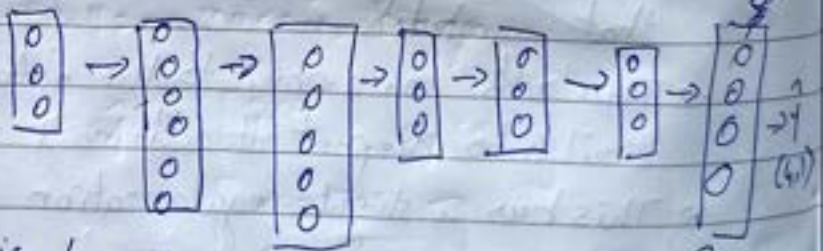
Multi-class Classification

Softmax Regression:

→ Softmax Regression is a generalization of logistic regression

→ We try to recognise one of multiple classes instead of just ~~one~~ 2 class ex, Recognising cats, dogs, baby chicks

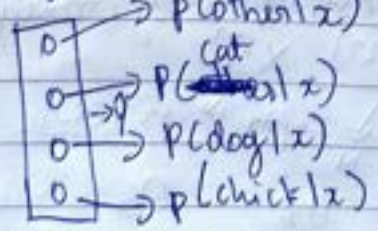
label		
0	others	C = no of classes = 4 (0..3)
1	cat	
2	dog	
3	chick	

$x \rightarrow$ 

→ No of units in o/p layer is 4 or equal to C. $n^{[L]} = C$

→ Output layer tells to the probability of each of the classes.

o/p layer

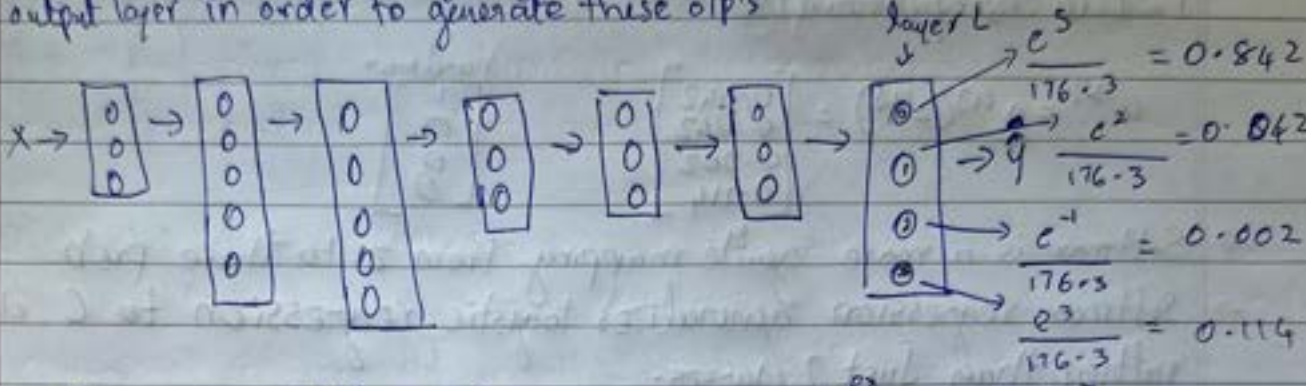


The probabilities ($\hat{y}$) should sum to 1.

~~The standard~~

Softmax layer:

The standard model for getting your nn to do this is called Softmax layer and output layer in order to generate these o/p's



$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]}$ $(4,1)$

Activation function

$t = e^{(z^{[L]})}$ $(4,1)$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i}, \quad a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

$(4,1)$

$a^{[L]} = g^{[L]}(z^{[L]})$ ↳ $(4,1)$

Eg

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

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \quad \sum_{j=1}^{4} t_i = 176.3$$

$a^{[L]} = \frac{t}{176.3}$

⇒ Unusual behavior of softmax activation layer is it needs to be normalized across different possible outputs and needs to take a vector put in output a vector.

Training a Softmax Classifier

Understanding Softmax:

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}, \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \quad (4,1)$$

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/e^5+e^2+e^{-1}+e^3 \\ e^2/e^5+e^2+e^{-1}+e^3 \\ e^{-1}/e^5+e^2+e^{-1}+e^3 \\ e^3/e^5+e^2+e^{-1}+e^3 \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

→ The biggest element in the z vector is 5, it has the highest prob
→ Softmax comes from hardmax where it looks the biggest elements in z and gi 1 and 0's everywhere else.

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \qquad \text{hardmax} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

→ Softmax is a more gentle mapping from z to these prob
→ Softmax regression generalizes logistic regression to C classes rather than just 2 classes.
→ If C=2, softmax reduces to logistic regression, $a^{[L]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$, these probs need to sum to 1, we dont need to compute 2 of them, just need 1. $a^{[L]} = \begin{bmatrix} 0.842 \\ \cancel{0.158} \end{bmatrix}$

Loss function :

Notice $y_1, y_3, y_4 = 0$ & $y_2 = 1$

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \text{cat} \quad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

$$L(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j \qquad J(w^{[L]}, b^{[L]}, ..) = \frac{1}{m} \sum_{m} L(\hat{y}^{(i)}, y^{(i)})$$

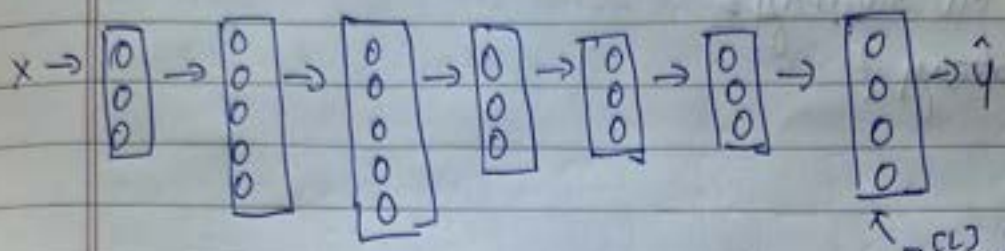
$$= -y_2 \log \hat{y}_2 = -\log \hat{y}_2 \quad \text{make } \hat{y}_2 \text{ bigger}$$

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

$$= \begin{bmatrix} 0 & 0 & 1 & .. \\ 1 & 0 & 0 & .. \\ 0 & 1 & 0 & .. \\ 0 & 0 & 0 & .. \end{bmatrix} \qquad = \begin{bmatrix} 0.3 & & & \\ 0.2 & & & \ldots \\ 0.1 & & & \\ 0.4 & & & \end{bmatrix}$$

$(4, m)$

Gradient Descent with softmax:



$z^{[L]} \rightarrow a^{[L]} = \hat{y} \rightarrow L(\hat{y}, y)$
$(4,1)$

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

$\frac{\partial J}{\partial z^{[L]}}$

Introduction to Programming Frameworks

Deep Learning frameworks:

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

Choosing DL Frameworks:

→ Ease of programming (development and deployment)
→ Running speed
→ Truly open (open source with good governance)