Coursera Deep learning.ai

Date
Page

26/7/18

Course-2

Improving DNN, Hyperparameter tuning, Regularization and Optimization.

⊙ Week 1 :

Data:

60%	20%	20%
train	C-V	test

If dataset is relatively small, traditional ratios might be okay but in case of a much larger dataset, we set c-v (devset) and test set to be much smaller than 20% or even 10% of entire data.

e.g. in 10,000 dataset, 6K, 2K, 2K distribu is fine i.e 60-20-20%
but in 10 million " , just 2% or even 0.02% data is enough for cross valide and test set.

Dev set is to check which of the different models applied on train set is the best to give results. If for example:

Train set	Dev/test set
Cat pictures from webpages (high resolu images)	Cat pictures from users using the cat recognition app (blurr and comparatively low resolu images)

In such a case, ensure that test set and dev set come from the same distribu of data. Doing this, progress in ML would be much faster.

Note: It is okay to not have a test set bcoz it is there just to have an unbiased estimate of performance. Many a times only train set and dev set are given, and people start calling this dev set the test set, which is not a very good use of terminology as they are then overfitting the test set, but it is okay if we don't need an unbiased estimate of the performance and all the data that we are concerned about is only that train and dev set.

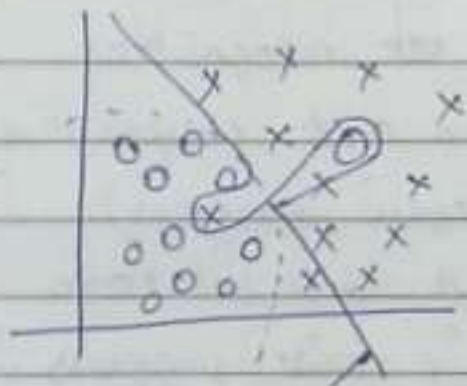
This train, dev and test setup will allow a more efficient measure of the bias and variance of our algorithm and better ways can be thus selected to improve our algorithm.

- Bias and Variance:

Train set error:	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	↓ high variance	high bias	high bias & high variance.	low bias & low variance.

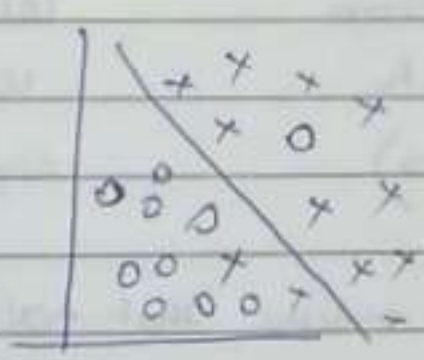
- assuming train & dev sets are drawn from the same distribution.
- assuming human error or Optimal bayes error = 0%

So, by looking at the training set error, one can get an idea of how well the model fits atleast the training set and tells us if we have a bias problem, and then seeing how much the error ↑es on going from train to dev set, we get a sense of how bad is the variance problem. (w. Sup)



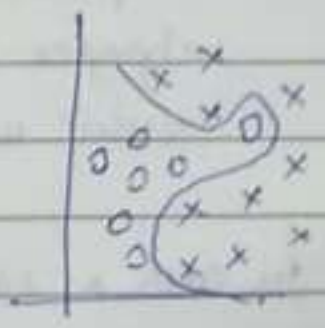
high bias & high variance model.

(bcoz in some region it shows high bias and in some it shows high variance by taking into consider^ even the mislabeled or outlier data)



high bias

to solve this:
- use bigger n/w
- train longer
- NN architecture search



high variance

to solve this:
- get more data
- do regularize^
- NN architecture search.

low bias, low variance — desired.

Note

Bias variance trade off is an old concept when we didn't have tools which could just reduce one error rate affecting the other

steps like ↑ing data or doing regularizaˆ just reduces variance without affecting bias, and getting a bigger NN for data just reduces bias without affecting variance much so long as regularizaˆ is done, which is a technique to reduce variance. Regularizaˆ ↑es the bias a bit but not much if we have a huge enough network.

→ Regularizaˆ: reduces overfitting and reduces errors in the network.

- in logistic regression -

$$\min_{w,b} J(w,b) \qquad w \in \mathbb{R}^{n_x},\ b \in \mathbb{R}$$

penalizes the weight matrices from being too large.

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

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

(norm of w squared)

[L2 regularizaˆ]
bcoz here euclidean norms are used also called L2 norm with parameter vector w.

Note regularizaˆ term is added only for w and not for b like $\frac{1}{2m}b^2$ because w is a high dimensional parameter vector, especially with a high variance problem while b is just a single number unlike w which might not be fitting all the parameters. So, adding $\frac{1}{2m}b^2$ won't make much of a difference because b is just 1 parameter over a very large no. of parameters in w.

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

$$= \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{1}{m}\|w\|_1 \qquad [L_1 \text{ regularization}]$$

↳ just a scaling constant.

In L1 regularization, w vector will end up being sparse i.e it will have a lot of zeroes in it, so takes less memory space. L2 is more often used than L1.

λ called the regularization parameter is set using the cross validation set and needs to be tuned.

- In Neural Network: {L = no. of layers}

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

↓ Squared norm

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

↓ no. of hidden units in layer l-1 and l.

F: "Frobenius norm"
(not called L2 norm bcoz of linear algebra technical reasons.)

Frobenius norm simply means sum of square of elements of a matrix.

Now, $dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$ — ①

$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$ — ②

↳ bcoz of regularized term.

we know, $dw^{[l]} = \frac{\partial J}{\partial w^{[l]}}$ (derivative of the cost fn w.r.t the parameters w)

Hence, L2 regularization is also sometimes called "weight decay".

Putting ① in ②:

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

↓

i.e whatever the matrix $w^{[l]}$ is, we are going to make it a little bit smaller.

→ little less than 1

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

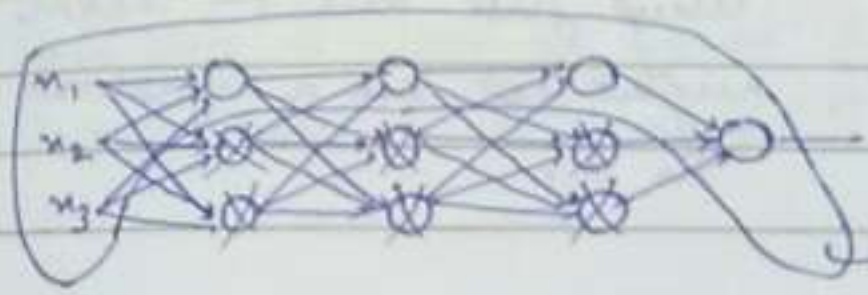
↓

this is why L2 regulariza^ is called "weight decay", bcoz α times the same gradient is subtracted but from a reduced value of $w^{[l]}$.

27/7/18

Q. why regularization help with overfitting?

Ans Regulariza^ shrinks the parameters and if made close to zero, its basically zeroing out a lot of the impact of the hidden units.

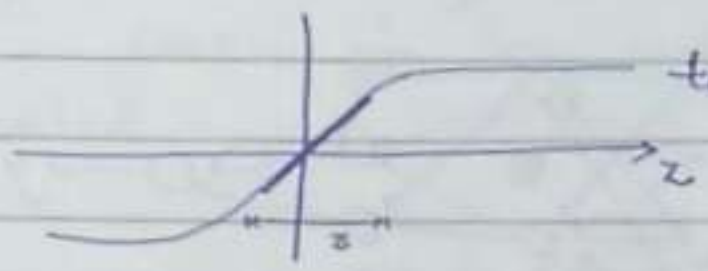


(NN reduced after regulariza^)

seems like logistic regression is being used.

Removing hidden units this way, shift the model from overfitting to the underfitting side and there exists λ which will settle it midway giving just the right fit. Hidden units are not completely removed, its just that their impact is reduced.

Also,



$g(z) = \tanh(z)$

If z takes on only a small range of parameters, tanh is linear only if larger values are allowed for z, the activa^ fn tanh starts to become less linear.

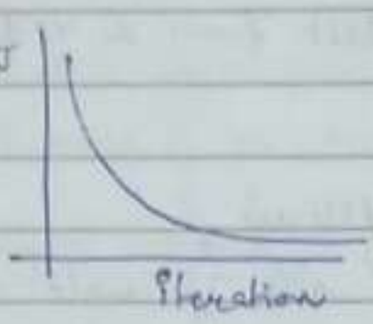
So, if λ is large, then our parameters will be relatively small bcoz they are penalized being large into a cost fn.

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

So, this makes the whole deep NN also a linear network, bcoz of the linear tendency of tanh for small z, this linearity is unable to fit very complex decisions thus allowing

it to avoid overfitting.

So, large λ ⟶ small w ⟶ small z ⟶ linear model which doesn't overfit.



J decreases monotonically after every iteration of gradient descent provided J is taken with regularization i.e.

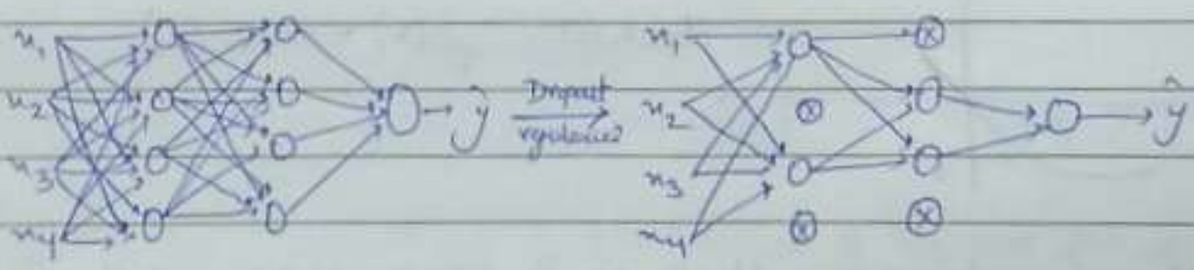
$$J = \sum L(\hat{y}, y) + \frac{\lambda}{2m} \sum_l \| w^{[l]} \|_F^2$$

Plot J as fn of no. of iterations of gradient descent to debug gradient descent.

Using just this 1st term might not see a monotonic decrease in J.

- Dropout regularization :

In this regularization, we go through each layer and set some probability of eliminating a node in neural network. Then some neurons are completely dropped and the backpropagation is applied on the new diminished network.



In this manner, different set of nodes are dropped each time and each training example is trained using one of these NN with dropout layers, resulting into regularization.

Implementing dropout ("Inverted dropout")

Step1: $d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\text{-}prob$

↓ dropout vector

↓ (probability that a given hidden unit will be kept.)

If keep-prob = 0.8 means that there is a 0.2 chance of eliminating any hidden unit.

Step 2: $a_3 = np.multiply(a_3, d_3)$ # $a_3 *= d_3$

(element wise multiplication)

for every element of d3 that's equal to zero (given there are 20% chance of it being 0), all the corresponding element of a3 end up being zero thus giving 0 activations.

Step 3: $a_3 /=$ keep-prob {'inverted dropout technique'}

This step is done so as to keep the value of $z^{[l]}$ intact.

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

↑ reduced by 20% due to dropout regularize^. so to compensate the loss 0.8 is divided to make up for the 20% loss.

So, inverted dropout is useful bcoz no matter what is the value of keep prob, it is ensured that the expected value of a3 remains the same. When this step 3 is not used, error in the test set becomes more and more complicated.

'd' vector ensures that for different training e.g.s, different hidden units are removed (bcoz of the random fn used) and intact for different iterations also, the same training example is trained by different network, with some different hidden units removed.

'd' decides which units to drop and which to keep during fwd and bwd propogation.

While doing predictions, no dropouts are done bcoz we don't want our output to be random. the complete original NN is used to make predic^ having weights set without getting overfitted as dropouts were made at time of training.

Adding dropouts at test time just adds noise to the predictions, nothing else. Also that step 2 was done previously only so that it was not needed at test time to keep a3, etc intact.

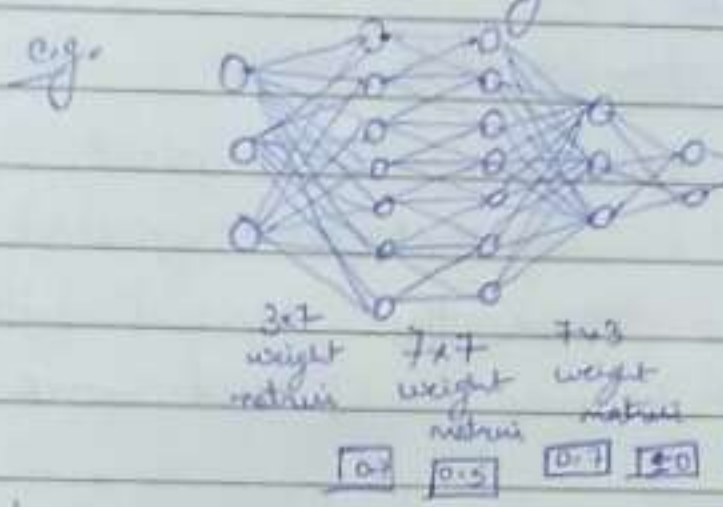
23/7/18

Q why does dropout work?

Ans Our network can't rely on any one feature, so the weights need to be spread out. Any one feature can go away at random and any one of its inputs could go away at random. So, it is unwise to put too much weight on any one input because it can go away. This dropouts will shrink the squared norm of the weights and does some L2 regularization that helps overfitting prevention. Results of L2 regularization and dropout regularization are similar.

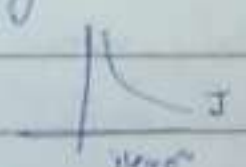
Note Keep-prob can vary with each layer also in the NN.

e.g.



In this case, keep-prob for layer 2 can be relatively low bcoz of the big 7x7 matrix, so as to prevent overfitting.

'Keep-prob' can be bigger like 0.7 or so when there is less fear of overfitting. Keep-prob = 1 means no dropout for that layer. This is like L2 only as here also different layers are having different regularization. The downside for this is that it ↑es the number of hyperparameters to be searched for using cross-validation, also J is not well defined. Sometimes, not very often keep-prob is applied to the input layer as well.



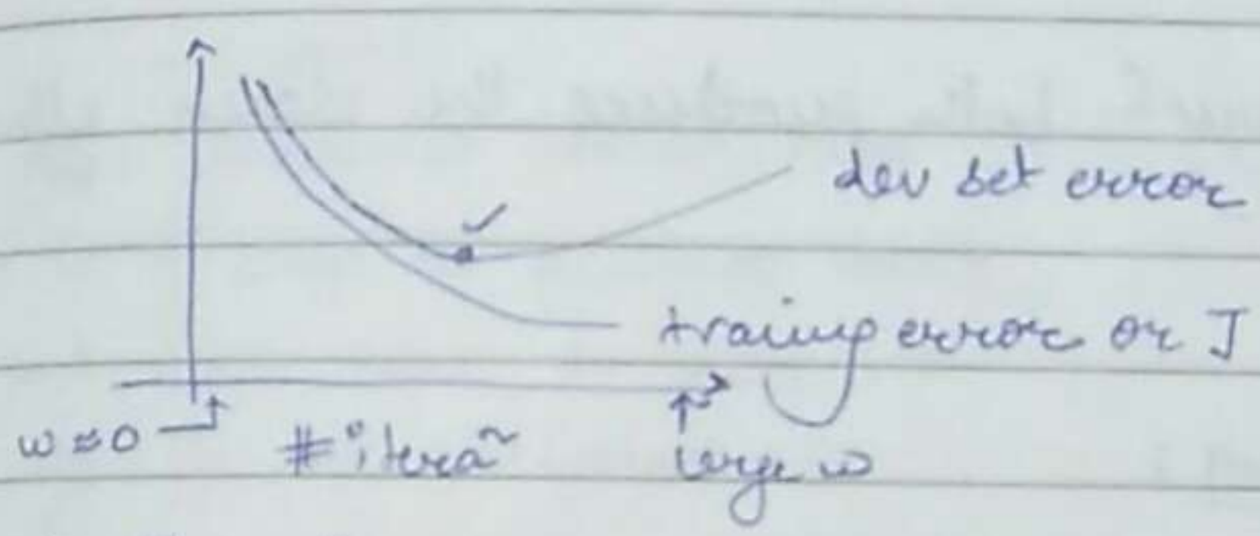
can't be now used to double check the performance of gradient descent, as now J is certainly hard to calculate bcoz of being less defined with regularization.

Date
Page

- Other regularize methods:

1. Data augmentation - bcoz every time collecting more data can be infeasible and expensive. Though this won't be as good as taking new pictures or new dataset but is free of any expenses. The augmented data docs not add that much info as a brand new image docs but is an inexpensive way to give more data to the algo.

2. Early stopping -



Usually the dev set error goes down for a while and it will ↑ from there. Early stopping means we stop the iterations at the point where it looks like the NN is doing best and take whatever value achieved this best dev set error. As the training is done, w gets bigger and bigger unlike the small values (usually 0) with which it was initialized. So, stopping the iteration midway give us a mid-sized w, and hence giving a less overfitting NN.

Note Downside of early stopping is that it couples the two tasks of optimizing cost fN and trying not to overfit thus hindering the orthogonality i.e in early stopping we no longer can work on these two problems independently. Bcoz stopping midway is not letting the $J(w,b)$ minimize in order for the model to not overfit.

Optimizing cost fN needs Gradient decent, adam, etc.
Preventing overfitting needs regularization, more data, etc.

Q. Difference b/w early stopping and L2 regularization?

Ans. In L2 regularization, we can train the NN as long as we want and that makes the search space of hyperparameters easier to decompose and easier to search over but the downside is that we might have to try a lot of values of λ making it computationally expensive.

Adv. of early stopping is that running the gradient descent process just once, we get to try out values of small w, mid-size w and large w without needing to try a lot of values as in L2.

L2 is still used more, though both produce the same effect more or less.

→ Setting up the optimization problem:

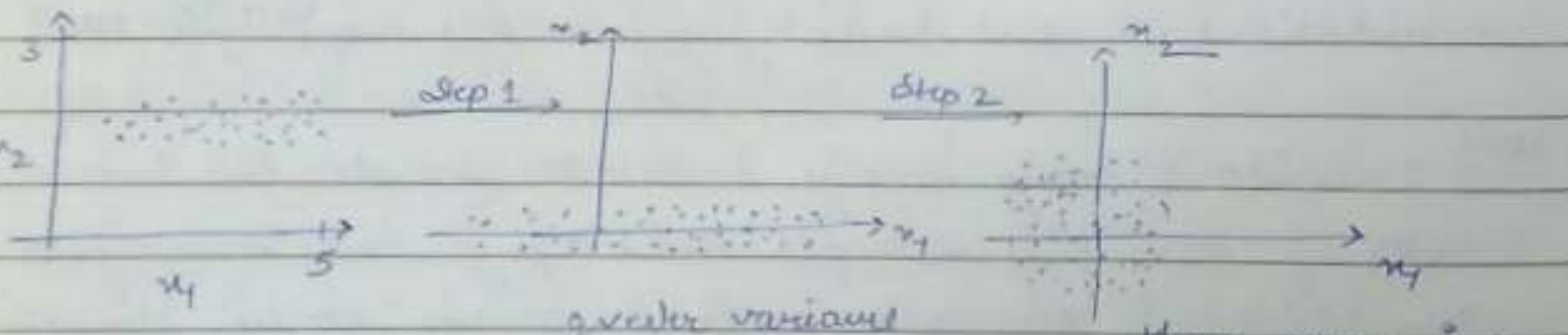
One of the techniques to speed up training is normalize your i/ps.

Step 1: Subtract mean $\{\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}\} \longrightarrow X = X - \mu$

Step 2: Normalize variance $\{\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)**2}\} \longrightarrow X /= \sigma^2$

↓ vector with variances of all the features.

↑ elementwise squaring



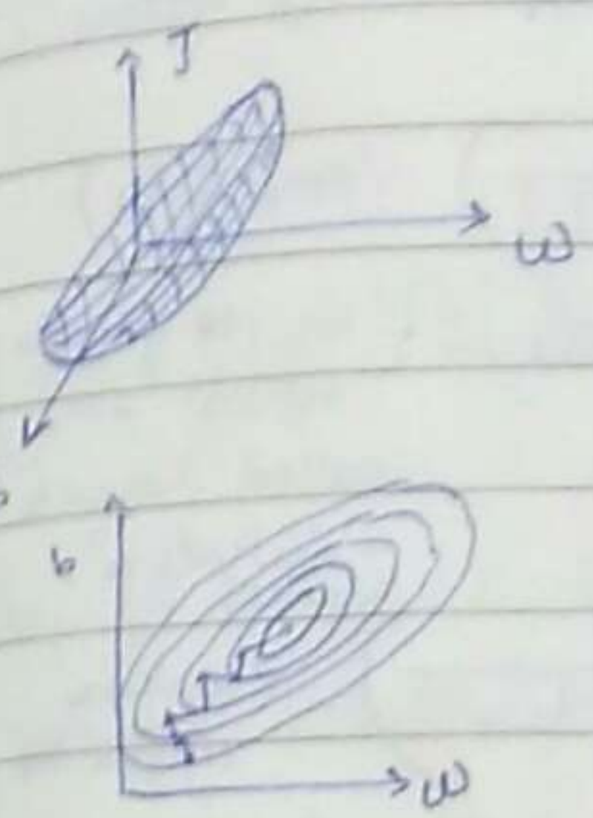
* Use same μ and σ for transforming the test set as well.

Q. Why normalize the dataset?

Ans. If the range of the different features vary too much like x1: 1 to 1000 and x2: 0 to 1 then cost fn comes out to be quite elongated. So, to

have a more symmetric J, normalize is done. (Note)

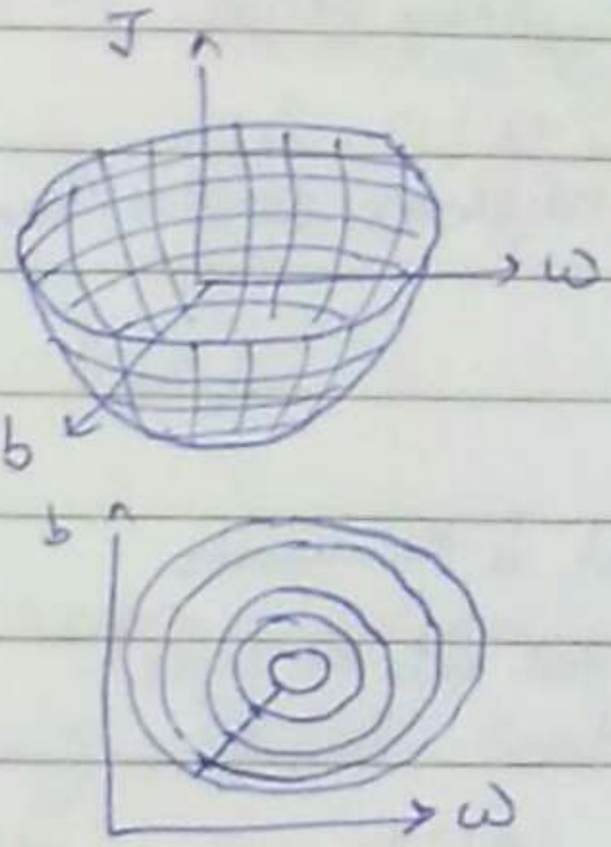
Unnormalized:



α : small

bcoz gradient descent takes a lot of steps to oscillate back and forth here to find its way to the minimum.

Normalized: (makes the cost fn easier and faster to optimize.)



α : large

bcoz here gradient descent can take much larger steps to reach the minimum.

- Vanishing / Exploding Gradient Decent :

Problem with training a very deep network, is that the derivatives or slopes can sometimes get either very big or very small, thus making training difficult.

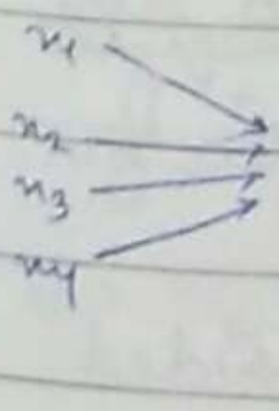
When $w^{[l]} > 1$, activations can explode. as $a^{[l]} = g(z^{[l]})$

when $w^{[l]} < 1$, activations can vanish. as a fn of L (no. of layers).

In a similar manner, the gradient and slopes vanish or explode, can be shown by using the same formulas for dw & all.

Note If gradients are smaller than 1, then gradient descent will take tiny little steps and it will take a long time for gradient decent to learn anything.

This problem can be solved by carefully initializing the weights during random weight initialization.



$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$

for z not to blow up & not become too small, larger the n is, smaller the w should be.

So, it can be done that $var(w) = \frac{1}{n}$ {n = # input features into each neuron}

If we are using Relu activn fn then $var(w) = \frac{2}{n}$ works better.

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

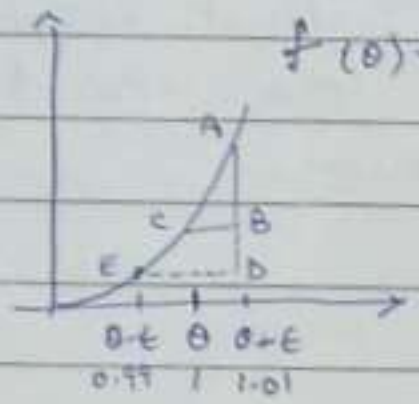
$\rightarrow np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$ {in case of tanh activn fn, called xavier initializn.}

Note $n^{[l-1]}$ bcoz that is the number of units being fed into the layer l.

$* np.sqrt\left(\frac{2}{n^{[l-1]} + n^{[l]}}\right)$ {another alternative}

15/8/18

→ Numerical approximation of gradients:



$\frac{AD}{DE}$ over $\frac{AB}{BE}$ is used as it is a much better approximn to the derivative of θ.

Gradient checking is done to make sure that implementn of backprop is correct. So, for it we need to numerically approx the gradients first.

Now, $\frac{AD}{DE} = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$

$$\frac{1.01^3 - (0.99)^3}{2(0.01)} = 3.0001$$

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

approx error: 0.0001

when only one side was considered, error = 0.03

So, the two sided difference helped getting close to $g(\theta)$ and tells that $g(\theta)$ is a pretty good approximn of derivative of $f(\theta)$.

this runs twice as slow as compared to the one-sided

define.

Note $f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$

↳ order of error for this is $O(\epsilon^2)$ and ϵ we know is very very small.

but $\lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta)}{\epsilon}$, error : $O(\epsilon)$.

So, by taking a two sided difference, we can numerically verify whether or not a fn g, $g(\theta)$ that someone else gives is a correct implementation of $f'(\theta)$. This can be used to verify the correctness of backpropagation implementation.

- Gradient checking : to verify " " " " for a NN

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

Concatenate.

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

Now, reshape $dw^{[1]}, db^{[1]} \ldots dw^{[L]}, db^{[L]}$ into a big vector $d\theta$ of same dimension as θ.

Q. Is $d\theta$ the slope or gradient of the cost fn $J(\theta)$?

Ans for each i :

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

$$\approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i} \text{ i.e (partial derivative of } J \text{ w.r.t } \theta_i)$$

So, we need to check if $d\theta_{approx} \approx d\theta$ to see if $d\theta$ is the slope. To see this we can compute Euclidean distance b/w the two vectors.

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ ——— ①

(for normalize?)

↳ Euclidean lengths

~~We take~~ $\varepsilon = 10^{-7}$

if ① $\approx 10^{-7}$, that's great as error is quite low.

If it comes out to be more like 10^{-5} or 10^{-3}, then there must be a bug somewhere and individual components of θ need to be checked if there is a specific value of i for which $d\theta_i$ is very different from $d\theta^{[i]}_{approx}$ and then that computa' error need to be tracked. If after debug guij, error reduces to 10^{-7}, then probably you have a correct implementation. So, it must pass with a small grad check value.

Imp notes on gradient checking:

① Use grad check only for debugging, not training as the computa' of gradcheck is very slow and so only during debugging its checked if $d\theta^{[i]}$ is close to $d\theta^{[i]}_{approx}$ and after that it is turned off.

② If algorithm fails grad check, look at individual components to try to identify bug.

③ Remember regularization term while doing grad check.

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

then $d\theta$ = gradient of J w.r.t θ including the regularize' term.

④ Gradcheck doesn't work with dropout bcoz in dropout there isn't an easy to compute $J(\theta)$ that dropout is doing gradient decent on because of its property of randomly dropping parameters in every iterations so implement Grad check without dropout i.e keep-prob = 1.0.

and then turn on dropout hoping its implementa^n to be correct. grad check can be used with dropout if the pattern of dropout is fixed, but usually its recommended to turn off dropout while grad check.

⑤ Run at random initializa^n ; perhaps again after some training as then w and b have some tnde to wander away from 0 which tends to have correct implementa^n of backprop, larger values of w and b give more inaccurate implementa^n.

Note You do no apply dropout and do not keep the $\frac{1}{keep\text{-}prob}$ factor in the calculations used in training.

⊙ Week 2 :

18/8/18

→ Mini batch gradient decent: Splitting the traning set into little baby training sets give mini-batches this helps get a faster algo as gradient decent starts to make some progress even before it finishes processing the entire traning set, but just 1 mini batch.

So, $X = [\underbrace{x^{(1)} x^{(2)} x^{(3)} \dots x^{(1000)}}_{X^{\{1\}}} | \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}} | \dots | \underbrace{x^{(m)}}_{X^{\{5000\}}}]$ $(n \times m)$

So, now we are having 5000 minibatches if $m = 5$ million. The same thing is done for Y as well.

So, Mini batch t : $X^{\{t\}}, Y^{\{t\}}$

1000 training examples with the corresponding i/p o/p features/ pairs

Note $Z^{\{t\}}$: t^{th} mini batch , $Z^{(i)}$: i^{th} traning e.g , $Z^{[l]}$: l^{th} layer

code for mini batch gradient decent:

repeat { // to take multiple passes through the training set.

  for t = 1, ..., 5000 { {considering 5K mini batches}

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

    $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$ // as we are processing only the mini batch t at a time.

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

    : // vectorized implementation all over processing 1K examples at a time

    $A^{[L]} = g^{[L]}(Z^{[L]})$ // rather than 5 million examples.

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

    ↓ examples from the minibatch t. ↓ regularized term.

    Backprop to compute gradients w.r.t $J^{\{t\}}$ {using $(X^{\{t\}}, Y^{\{t\}})$}

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

  }

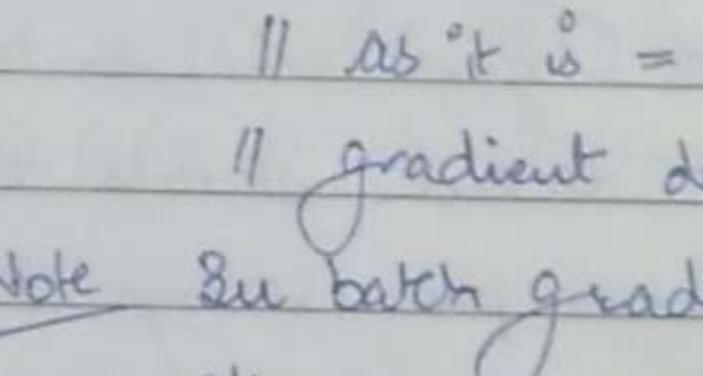
  // this code is also called doing 1 epoch of training

  // as it is = 1 pass through the training set using mini batch gradient decent.

Note: In batch gradient decent, a single pass through the training allows us to take only one gradient decent step while with mini batch gradient decent, single pass through training set, allows to take 5K (as in above e.g.) gradient decent steps. With large training set, mini-batch runs much faster than the batch gradient decent.

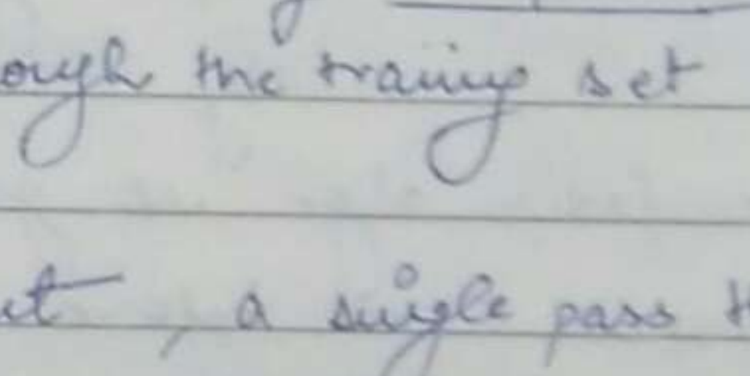
Batch gradient decent

(i)



{goes through complete dataset on every iteration}

(ii)



Plot $J^{\{t\}}$ computed using $X^{\{t\}}, Y^{\{t\}}$.

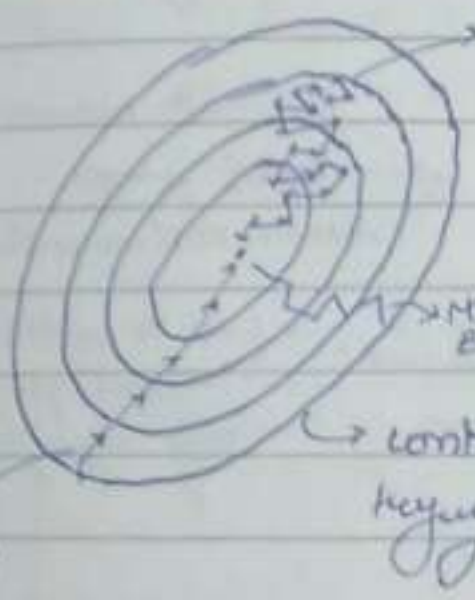
{it will go down but is noisy}

the (2) is noisy bcoz may be the batch $x^{\{1\}}, y^{\{1\}}$ is just the easiest of every mini batch, so cost might be little lower but maybe just by chance, $x^{\{2\}}, y^{\{2\}}$ is just a harder mini batch, as a result of which the cost will be a little higher.

Parameter: size of mini batch

- may dwindle from m (= batch gradient decent) to 1 (stochastic gradient decent), In batch, there is just 1 batch while in stochastic, every example is a batch.

Now we will see, what these two extremes do on optimizing the cost fn.



→ contours that we are trying to minimize.

- Batch GD takes greater steps, has low noise. It takes too long per iteration.
- SGD takes very small steps and is highly noisy but on an average will take us on a good dir?. SGD never converges to minimum, it keeps on oscillating nearby but never stays there.

Noise in SGD can be overcome by using a small learning rate but the huge disadvantage is that we lose all the speedup that can be gained through vectorization as we are processing a single example at a time. Its good that we are learning after every example but the time becomes a major issue.

BGD works fine with smaller datasets, both learning wise and time wise.

So, midway is best i.e t somewhere b/w 1 and m, bcoz we do get a lot of vectorization which would be much faster than processing the examples one at a time. Also, progress is

made without needing to wait till the processing of entire training set. In our previous eg. we had 5000 gradient steps per iteration or epoch.

- Guidelines to choose the mini batch size:

i) In case of small training set, use BGD. Eg. when m ≤ 2K)

ii) " " " big " " , typical mini batch sizes would be anything from 64 upto may be 512 (typical values) bcoz the way computer is laid down, sometimes computa~ are faster when size is in power of 2. So, in previous example try using 1024 instead of 1000. This mini-batch size that is decided should fit the CPU, GPU memory otherwise performance will fall off a diff.

* Optimize algorithm faster than GD ← GD with momentum
RMS prop
Adam optimization

- Exponentially weighted moving averages in statistics: (to be understood before going to any algo)

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

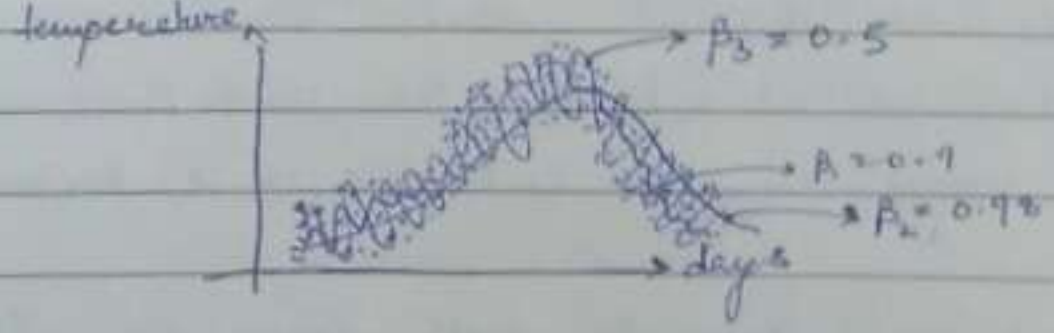
θ_t → temp at time t

$\beta_1 = 0.9$: ≈ averaging 10 days temp.

$\beta_2 = 0.98$: ≈ " 50 days temp

$\beta_3 = 0.5$ ≈ " 2 " "

V_t approximately averages over $\approx \frac{1}{1-\beta}$ days' temperature.



Plot for β_2 is much smoother bcoz now we are averaging over more days of temperature but the curve has shifted further to right bcoz you've now averaging over a much larger window of temperatures and so it adapts more slowly when the temp changes.

↓

exponentially weighted moving avg. formula

It is low when β is large, much weight is given on the previous temperature values and only $(1-\beta)$ weight is given on the current temp. value. $\{\beta v_{t-1} + (1-\beta)\theta_t\}$
When $\beta = 0.5$, graph is much more noisy as only 2 days temp is being averaged over and so graph is more susceptible to outliers and adapts much more quickly as the temp. changes.

If
$$\begin{cases} v_{100} = 0.9\, v_{99} + 0.1\, \theta_{100} \\ v_{99} = 0.9\, v_{98} + 0.1\, \theta_{99} \\ v_{98} = 0.9\, v_{97} + 0.1\, \theta_{98} \\ \vdots \end{cases}$$

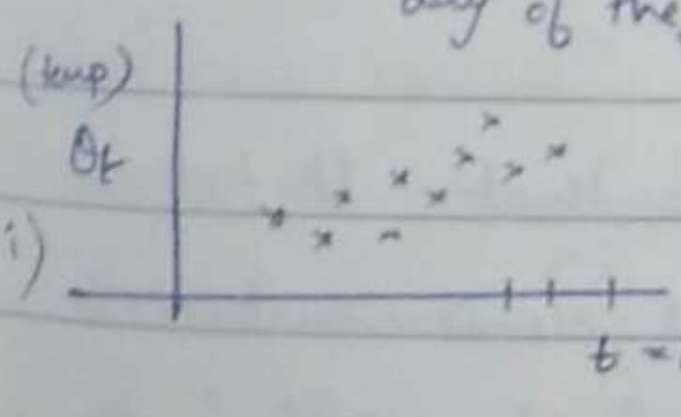
$$v_{100} = 0.1\,\theta_{100} + 0.9\, v_{99}$$
$$= 0.1\,\theta_{100} + 0.9\,(0.9\, v_{98} + 0.1\,\theta_{99})$$
$$\vdots$$

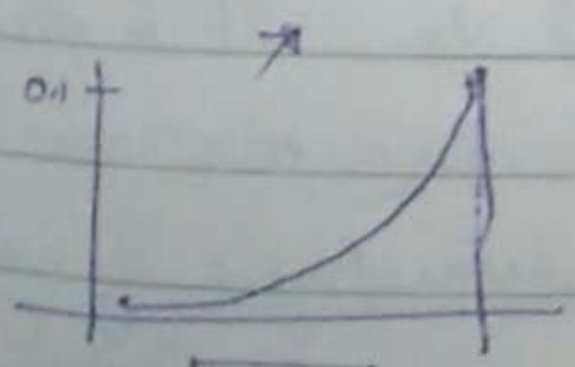
and so on we get,

$$\boxed{v_{100} = 0.1\,\theta_{100} + 0.1 \times 0.9\,\theta_{99} + 0.1\,(0.9)^2\,\theta_{98} + \dots .}$$

→ calculated on the 100th day of the year.

weighted average or sum of θ_{100}.
↓
current day's temp.





(exponentially decaying function)

$\boxed{v_{100}}$ = element wise product b/w functions i) and ii) and sum it up.
i.e daily temp (θ_t) multiplied with the exponentially decaying fn (going from 0.1 to $0.1(0.9)$ to $0.1(0.9)^2$ to approx 0)

Note: All the coeff. in v_{100} sum up to approximately 1.

Now, we need to know, how many days are being averaged over. well, $(0.9)^{10} \approx 0.35 \approx \frac{1}{e}$

$(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e} \approx 0.35$ (ε in this case = 0.1)

this means it takes about 10 days for the weight of exponential f^N to decay to around $1/3^{rd}$ of the peak. Hence, when β, $\beta = 0.9$ we say that it is as if we are computing an exponentially weighted average that focuses on just the last 10 days temperature, bcoz its after 10 days that the weight decays to less than about a third of the weight of the current day. while when $\beta = 0.98$, $(0.98)^{50} \approx 0.35$, so we say 50 day avg. in case of $\beta = 0.98$. From here we say that we are averaging over $\frac{1}{1-\beta}$ days $\{i.e. \frac{1}{1-\beta}\}$

- implementing exponentially weighted averages:

$V_0 = 0$	$V_0 = 0$ {on computer}.
$V_1 = \beta V_0 + (1-\beta)\theta_1$	Repeat {
$V_2 = \beta V_1 + (1-\beta)\theta_2$	Get next θ_t
$V_3 = \beta V_2 + (1-\beta)\theta_3$	$V_0 = \beta V_0 + (1-\beta)\theta_t$ // (update)
...	

advantage of this exponentially weighted avg. is that it takes very little memory. we need to keep just 1 row number in computer memory, and then keep on overwriting V_0. It is for sure not the best method to calculate avg. like the one conventionally used $\left(\frac{\theta_1 + \cdots \theta_t}{t}\right)$ but the disadvantage of using the conventional is that it requires more memory as we need to keep and store all 10 days or 50 days temperatures and is computationally more expensive.

19/8/18

- Bias correction in exponentially weighted averages:

In the initial days, estimate of the temp. is not very good as

$V_0 = 0$
$V_1 = 0.1 V_0 + 0.02\,\theta_1 = 0.02\,\theta_1$

$v_2 = 0.98 v_1 + 0.02 \theta_2$

$v_2 = 0.0196 \theta_1 + 0.02 \theta_2$ // v_2 comes out to be much less // than θ_1 or θ_2, hence not a // very good estimate for the // first two days of the year.

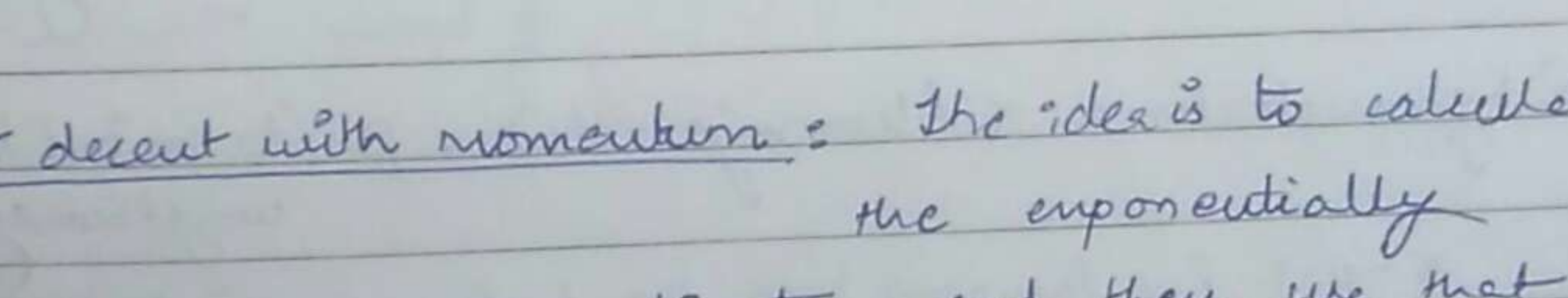
So, an correc is made for this bias to remove.

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

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

$$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$ // bias is removed.

When t is large, $1-\beta^t \approx 1$ and hence the bias correc makes no big difference.



This bias correc can help us get a better estimate early on when the moving wighted avg. is warming up and is in its initial phases.

① Gradient decent with momentum : The idea is to calculate the exponentially weighted average of your gradients, and then use that gradient to update the weights instead.

A very large learning rate might end up overshooting or

diverging and so the need to prevent the oscillations from getting too big forces to use a learning rate that itself not too large.

We want the learning to be slower on vertical axis to avoid those oscillations but on the horizontal axis faster learning is desired.

Momentum:

Initialize $V_{dw} = 0$, $V_{db} = 0$ {same dimensions as w & b respectively.}

On iteration t: (mini batch number)

Compute dw, db on current mini batch

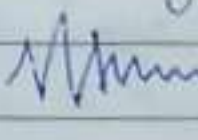
$V_{dw} = \beta V_{dw} + (1-\beta) dw$ // computing moving average of

$V_{db} = \beta V_{db} + (1-\beta) db$ // the derivatives for w you

$w = w - \alpha V_{dw}$, $b = b - \alpha V_{db}$ // are getting thus smoothing // out the gradient descent // steps.

'friction' (to stop from speeding) — velocity — acceleration

this way, GD with momentum ends up eventually just taking steps that are much smaller oscillations in the vertical direction but are more directed to just moving quickly in the horizontal direction, and so this allows our algorithm to take a more straightforward path.



2 hyperparameters = α (learning rate), β (to control the exponentially weighted avg.)

$\beta = 0.9$ has for now been the best value, but keep on trying others as well if in case some other value proves to be more robust.

Not much effort is put for removing the bias estimate as the GD learns after a little warming up automatically.

Momentum GD works better than the straight GD.

② RMS prop (Root mean square prop)

We saw that using GD might end up in huge oscillation in the vertical direction even while it's trying to make progress in horizontal direction.

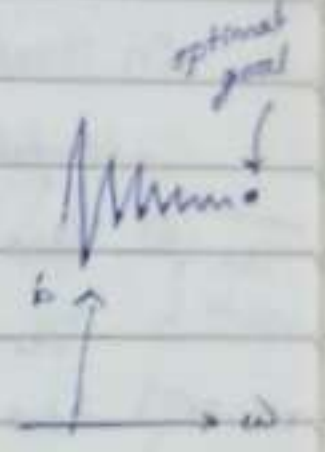
On iteration t :

compute dw, db on current mini batch :

$$S_{dw} = \beta S_{dw} + (1-\beta)\, dw^2 \quad \hookrightarrow \text{(element wise square)}$$

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

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

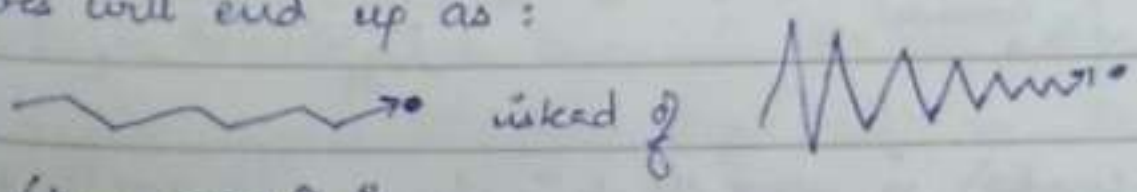


Slope is very large in the b direction, relatively large db and a relatively small dw. Updates in the vertical dir^n are divided by much larger no. $\sqrt{S_{db}}$ to damp the oscillation in that dir^n whereas updates in horizontal dir^n are divided by a smaller number.

We want learning to be fast in w dir^n and damp all the oscillation in b dir^n.

to avoid the denominator being zero.

So, net impact of using RMS prop is that updates will end up as :



instead of

(less oscillation in vertical dir^n).

and so a larger α can be used without diverging in vertical dir^n, and helping in learning fast.

Here we are taking this as 2D, but in practical applica^n, the dir^n in which we want to damp the oscillation may be a sum set of many w's like w_1, w_2, w_3 etc, and the horizontal

dw may be w_5, w_6, w_7 etc and so on bcoz in practice it is multidimensional. dw and db are both high dimensional

(3) Adam Optimizaⁿ Algorithm : (=Momentum + RMS prop)

$V_{dw} = 0,\ S_{dw} = 0,\ V_{db} = S_{db} = 0$ // Initializeⁿ

On iterⁿ t:

Compute dw, db using current mini-batch

momentum like update:
$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\, dw \quad \text{// using } \beta_1 \text{ instead of } \beta \text{ to}$$
$$V_{db} = \beta_1 V_{db} + (1-\beta_1)\, db \quad \text{// differentiate from earlier}$$

rms prop like update:
$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2 \quad \text{// usage.}$$
$$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2$$

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

(after bias correcⁿ)

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

$$w = w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon}, \quad b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Adam optimaⁿ (a combo) is very effective for many different NN & a very wide variety of architecture.

- Hyperparameter choice: α : needs to be tuned
  β_1 : 0.9 (dw)
  β_2 : 0.99 (dw^2)
  $\varepsilon = 10^{-8}$ (doesn't affect much)

β_1 and β_2 can also be tuned but are generally not and are used with these robust values only like 0.9 and 0.99 for β_1 and β_2 respectively.

Adam optimizeˆ can be used in batch and min-batch GD both.

Adam : Adaptive moment estimation. bcoz ↴

β_1 is computing the mean of the derivatives. This is called the first moment. β_2 is used to compute the exponentially weighted avg. of the squares and that's called second moment.

- learning rate decay :- To slowly reduce the learning rate over time can help speed up the learning algorithm.

As we iterate, the steps are a little bit noisy and tend towards the minimum but won't exactly converge. It will go on being around the minimum point but would not stop at it bcoz of using some fixed value of α and there is some noise in the mini batches. So, reducing α at later stages like when it reaches close to minimum helps oscillations in a tighter region around this minimum rather than wandering far away, so α needs to be reduced bcoz in initial stages of learning, one might afford bigger steps but as learning approaches convergence, having a slower learning rate allows to take smaller steps.

1epoch = 1 pass through the data

$$\alpha = \frac{1}{1+(\text{decay rate} * \text{epoch number})} * \alpha_0$$

formula for learning rate decay

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

if $\alpha_0 = 0.2$
decay rate = 1

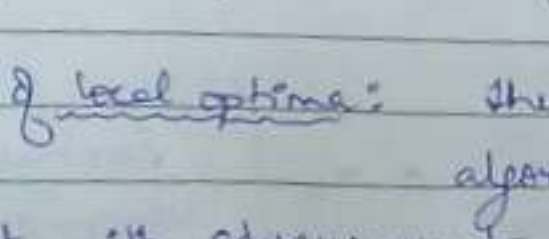
any constant less than 1.

$$\alpha = 0.95^{\text{epoch number}} \cdot \alpha_0$$ (exponential decay of learning rate.)

Or

$$\alpha = \frac{k}{\sqrt{\text{epoch number}}} \alpha_0$$ or $$\alpha = \frac{k}{\sqrt{t}} \cdot \alpha_0$$ (t → epoch number)

Or



learning rate decaying in discrete steps.

(discrete staircase)

Or

Manual decay : If we are training just one model at a time and if your model takes many hours or even days to train, then just watch the model getting trained over a large no. of days and then manually say, it looks like α has slowed down, I am going to ↓ α a little bit. So, its turning α by hand hour by hour or day by day. As long as we are training a small no. of models, its feasible.

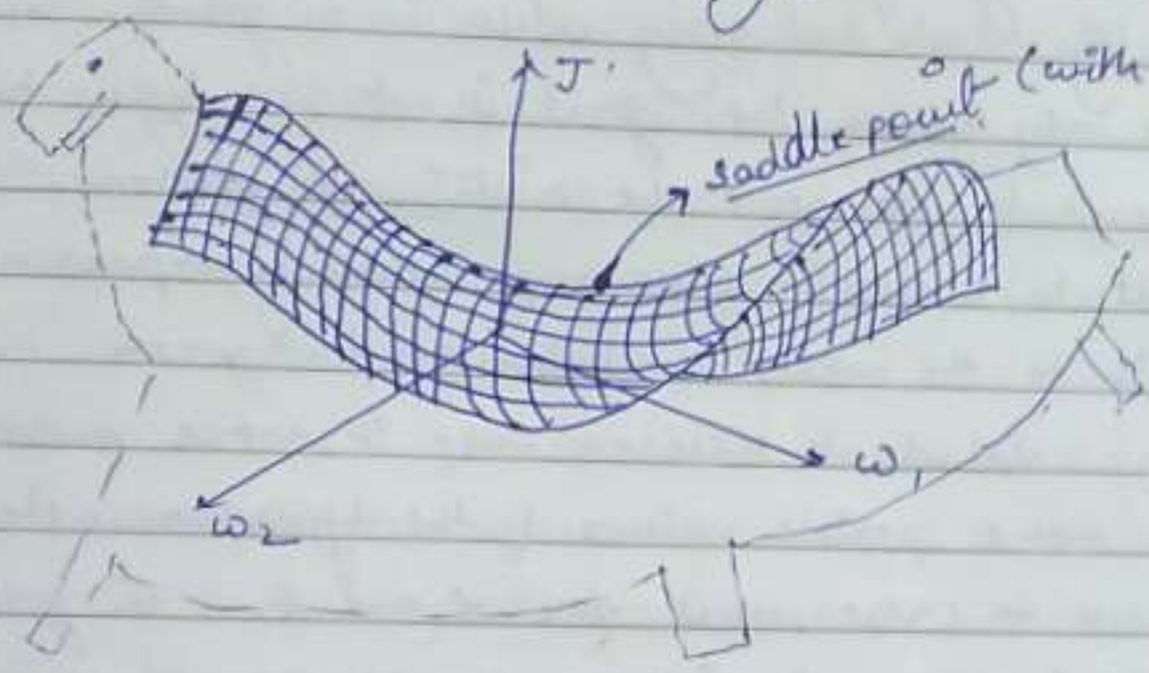
Note It is sometimes preferred to have just 1 tuned α for learning. Though LRD (decay) helps but is less often used.

- Problem of local optima: There is a problem with optimization algorithms getting stuck in a bad optima but with advancement in understanding of deep learning, concept of local minima has also changed.

Earlier we visualized

i.e a 2-D plot to see local optima but in practice, such low dimensional

problems don't exist and in high dimensional space, it is much more likely to run into a saddle point rather than a global or local optima (where gradient is zero in all the dimensions) which is very hard to occur. Gradient might be zero for some dimensions at a point but not for all and such a point is called saddle point, where the curve function is bending



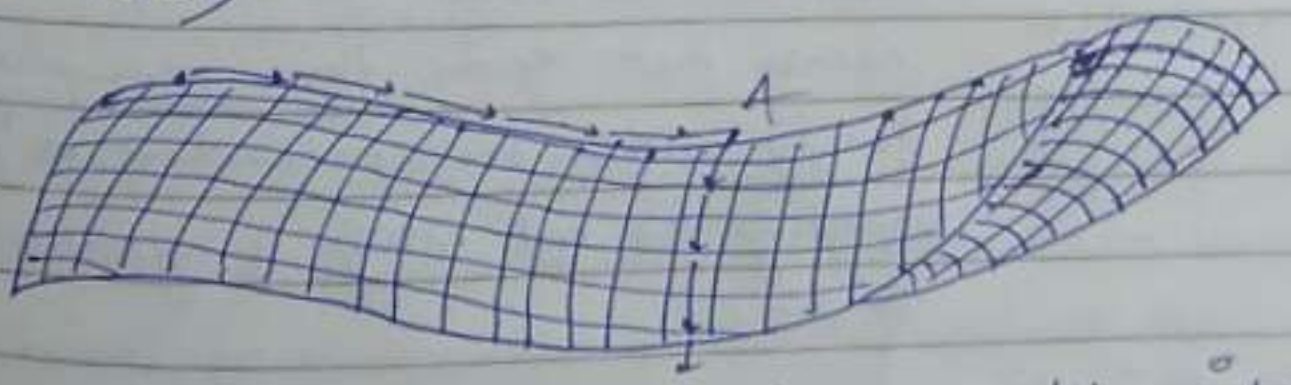
called a saddle point bcoz it can be put on the back of the horse.

So, our assumptions on low Dimensional space do not always generalize well in high Dimensional space in deep learning.

Q. If local optima is not an issue, then where does the problem lie?

Ans It turns out that plateaus can really slow down learning.

(a region where the derivative is close to zero for a long time.)



So, the gradient might make it long way till point A and then fall off the plateau. So, at these points the optimization actually help speed up the process and are of help.

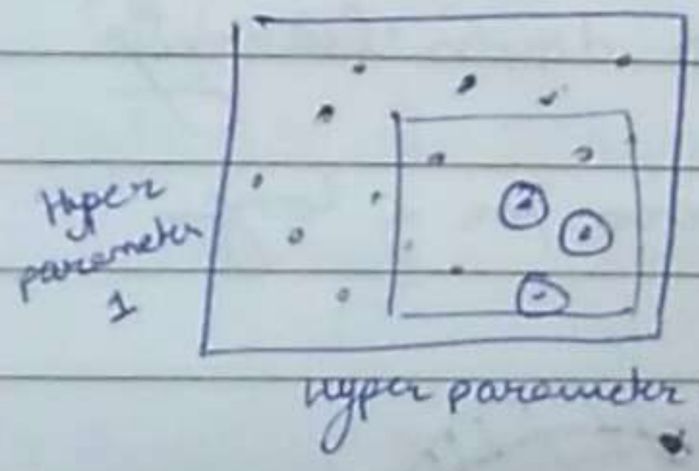
⊙ Week 3 : [HYPERPARAMETER tuning and much more]

α > β, #hidden units, minibatch size > $\beta_1, \beta_2, \varepsilon$

in top order of the hyperparameter importance
(not a hard and fast order though)

Traditional way of finding the right value of hyperparameter was to try each and every value in a grid but that is feasible only when the grid is not too big but now its recommended to choose the points at random, bcoz it is difficult to know in advance which hyperparameters are going to be the most important for your problem. So, selecting values at random make it more probable to choose more possible values of the hyperparameters. e.g. if I have to chose value for α and ε, in grid method if grid is 5x5, I will go across 25 values with only 5 different values of α and wasting too much energy on doing ε as it is not affecting the results much, while in random selec' this won't happen.

Another common practise is to use a coarse to fine sampling scheme.

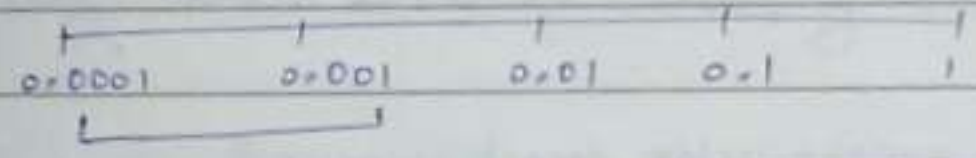


If I find that points in a particular region are doing well for the model, then we will zoom in and select the other points from nearby that region thus sampling more density within the space where we are suspecting the best setting of hyperparameters to lie.

Note Sampling at random doesn't mean sampling uniformly at random over the range of valid values instead its important to pick the appropriate scale on which to explore the hyperparameters

eg suppose we need to choose the no. of hidden units in layer l.
And we find $n^{[l]}$ = 50 to 100 to be the possible range,
Then random selec^n in b/w 50 to 100 can be reasonable.
If I was to chose no. of layers which I think to be okay b/w 2 to 4 then uniform sampling b/w 2 to 4 would have been reasonable. In these cases, sampling at random works BUT

If I am trying to tune α and suspect 0.0001 to 1 to be the possible range, at that time uniform random selec^n doesn't work, instead it seems more reasonable to search for hyperparameters on a log scale. Bcoz on a linear scale b/w 0.0001 to 0.1, 90% of the uniformly randomly chosen values will lie b/w 0.1 to 1 thus eliminating the scope for values b/w 0.0001 to 0.1.
But on a log scale this problem is solved as:



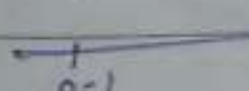
More resources are now available for searching b/w 0.0001 and 0.1

Implementa^n : $r = -4 * np.random.rand()$ $\quad r \in [-4, 0]$ $\quad [a, b]$

20/8/18 $\quad \alpha = 10^r$ $\quad \alpha \in [10^{-4}, 1]$

Now r can be set uniformly at random b/w a and b.
Hence, appropriate scale (log in above case) is necessary for hyperparameter tuning.

Another difficult thing is selecting hyperparameters for exponentially weighted averages. So, let's say β is suspected to be b/w 0.9 to 0.999 where 0.9 means avging over the last 10 days while 0.999 is like averaging over the last 1000 days/values.
So, now instead of sampling β, we will sample $1-\beta$ ie from 0.001 to 0.1 which has same resources as range of β.



Q. Why is it a bad idea to sample on linear scale?

Ans When β is close to 1, the sensitivity of the results we get changes even with very small changes to β. If β goes from 0.9 to 0.9005, it's no big deal, this is hardly any change in the results but if β goes from 0.999 to 0.9995, this will have a huge impact on exactly what the algo is doing. In both of the cases, it's averaging roughly over 10 values but- in 1st it has gone from an exponentially weighted average over about 1000 examples to now the last 2000 examples

Also, we have $\frac{1}{1-\beta}$ which is very sensitive to small changes in β when β is close to 1.

So, the sampling process helps to sample more densely in the region when β is close to 1 or when $1-\beta$ is close to zero, thus helping in exploring the spaces of possible outcomes more efficiently.

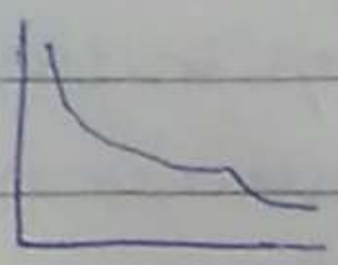
Q How to organize hyperparameter search process?

Ans Retest and re-evaluate your hyperparameters at least one every several months.

two ways to organize hyperparameter search process

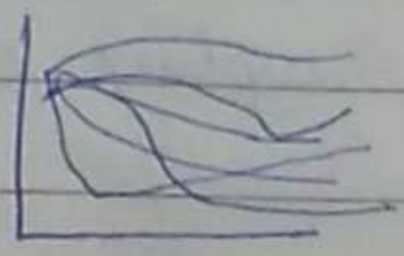
- Babysit one model (when there is lot of data but less of computational resources) — this approach is when only one model can be run,
- training many models in parallel.

In babysitting approach, the hyperparameters are tuned every day even when the model is getting trained, while in the other approach, the models are left to run on their own with some setting of the hyperparameters for days or weeks. Many models are run in it and the best one is chosen at the end.



Babysitting method

(Panda approach)



parallely models running approach.

(Linear approach)

Panda approach bcoz they also give one or 2 baby at a time and then pay too much attention to ensure their survival.
Caviar approach bcoz reptiles give millions of eggs and don't pay much attention afterwards hoping that atleast one of them or a bunch will survive.

Q. It's the computational resources that decide the kind of approach to be followed.

* Batch Normalization : (makes hyperparameter search problem much easier.) (also makes our NN more robust.)
(also helps train very deep NN)

(as seen before) in logistic regression

Q. for any hidden layer can we normalize the values of $a^{[l]}$ (activa) so as to train $w^{[l]}$, $b^{[l]}$ faster ? It is bcoz $a^{[2]}$ is the input to the next layer and will affect w and b.
Ans. Technically we normalize $z^{[2]}$ and not $a^{[2]}$ by default.

→ Implementing Batch Norm :
Given some intermediate values in the NN like $z^{[2](1)}$, so given these values, we compute the mean as follows :

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

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

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

↳ for numerical stability just in case $\sigma^2 = 0$ ✓

Normalizing the i/p features x can help learning in a NN. Batch prop extends this to even hidden layers.

Now, these z's have mean 0 and standard unit variance.

→ This makes $z^{(i)}$'s have some fixed mean & variance.

→ tilde

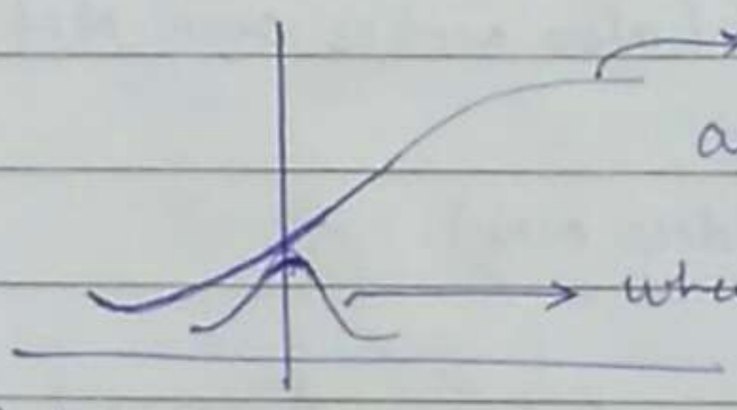
$$\tilde{z}^{(i)[l]} = \gamma z^{(i)[l]}_{norm} + \beta$$

(γ and β are learnable parameters of the model)

→ This is done bcoz we don't want the hidden units to always have mean 0 and variance 1, may be it makes sense for hidden units to have a different distribution unlike the i/p features.

Note If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, $\tilde{z}^{(i)} = z^{(i)}$

Now new values $\tilde{z}^{(i)}$ will be used instead of $z^{(i)}$ for training purposes and later computa? in NN.

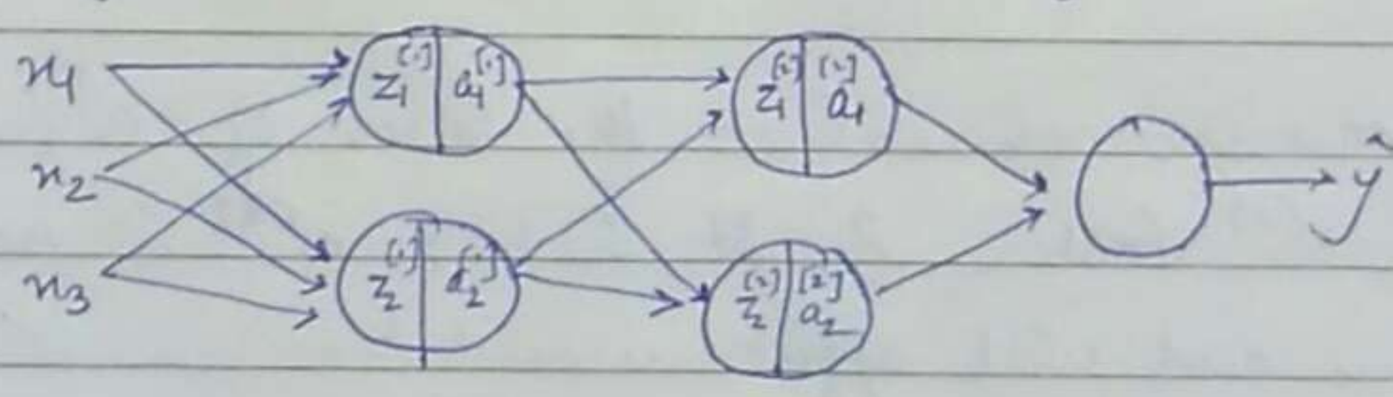


→ when μ and σ^2 have fixed values, activation (sigmoid) has greater range / variance.

→ when $\mu = 0$ and $\sigma^2 = 1$, the range reduces and is concised to such limits which is undesirable.

21/8/18

→ Fitting batch norm into the training of a deep network:



$$X \xrightarrow{w^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[\text{Batch Norm}]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} Z^{[2]}$$

$$\beta^{[2]}, \gamma^{[2]} \text{ (BN)} \rightarrow \tilde{Z}^{[2]} \rightarrow a^{[2]} \cdots$$

So, we are using the mean & normalized value $\tilde{z}$ instead of the regular Z. Here β is different from one used in momentum.

So, the parameters we have are: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots w^{[L]}, b^{[L]}, \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots \beta^{[L]}, \gamma^{[L]}$

This batch norm is provided in programing frameworks and can be applied with just 1 line of code as provided in tensorflow or other frameworks.

$$\beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}$$ (after using some optimize algorithms).

In case of mini batches, the batch normalization step is done mini batch wise only, considering that much data only for computation.

→ Implementing GD using Batch Norm:

for t = 1 to no of mini batches

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

In each hidden layer use Batch Norm to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

// This ensures that the value Z end up with some normalized // mean and variance.

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

Update the parameters. $w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$

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

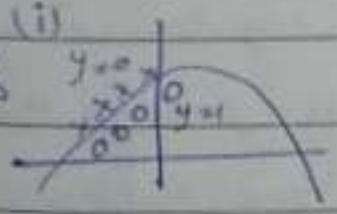
$\gamma^{[l]} = \gamma^{[l]} - \alpha\, d\gamma^{[l]}$

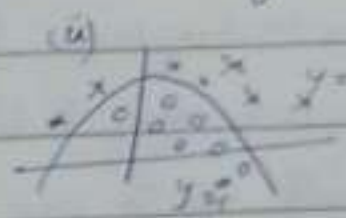
// This also works with GD with momentum or RMSprop // or adam. These algos can also be used to update the // parameters that are added by the Batch Norm.

Q Why does Batch Norm work?

Ans By normalizing all the input features x to take on a similar range of value can speed up learning. Batch Norm is a similar thing which extends it to hidden layers and not just the i/p features.

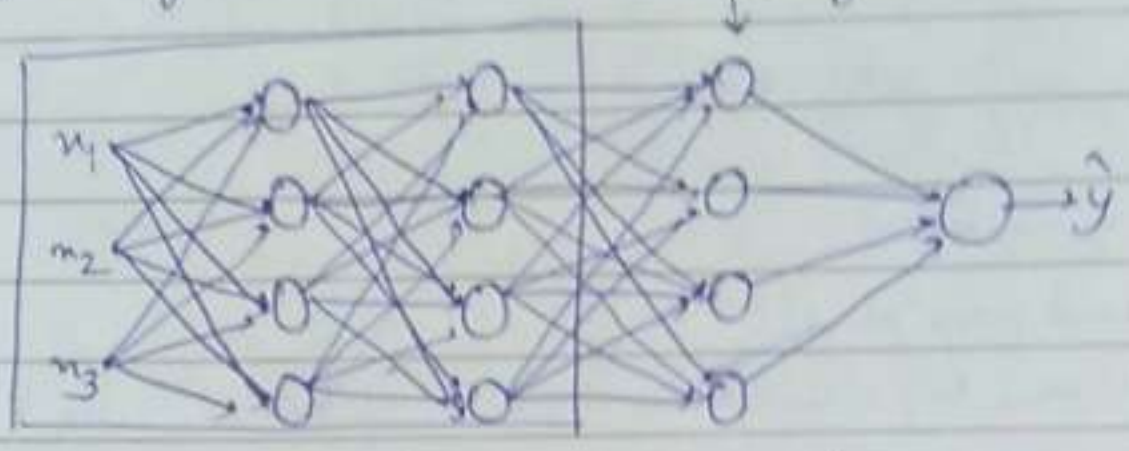
Also, batch norm makes the weights in the deeper layers of the network more robust than the ones in earlier layer.

If data is (i) 

and other data is (ii) 

though the fN is same for both the data but model trained on (i) will not be able to identify the fN for (ii). This shift of data is called 'covariate shift'. So, there is a need to retrain the model if data distribution changes and the situation worsens if

even the function shifts. So, how does this problem of covariate shift apply to a NN?



Let's look from the point of view of 3rd hidden layer. This layer is getting some values as $a_1^{[2]}, a_2^{[2]}, a_3^{[2]}, a_4^{[2]}$. Now the job of 3rd layer to map these as to $\hat{y}$. The network is adapting the values of $w^{[3]}, b^{[3]}, w^{[4]}, b^{[3]}$ continuously and so as these parameters change, $a^{[3]}$ will also change, so from 3rd hidden layer's perspective, these values are changing all the time and is suffering from covariant shift. So, Batch Norm reduces the amount that the distribution of these hidden unit values shifts around. Even if the i/p changes, the later layers stand firm and don't get affected too much.

Also, batch norm has a slight regularize effect.

- 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 some noise to each hidden layer's activation, bcoz the estimates of the mean and standard deviation are noisy.
- So, similar to dropout, batch norm has some regularize effect bcoz by adding some noise to the hidden units, it's forcing the downstream hidden units not to rely too much on any one hidden unit.
- Batch norm can be used together with dropouts bcoz the regularize effect is not too impactful.
- Using larger mini batch size, reduces noise and hence reduces regularization.

Date
Page

Note Don't use batch norm for regularization but as a way to normalize the hidden units and ∴ speed up learning, and regularize is almost an unintended side effect.

Q. Batch norm works on minibatches, but at test time there are single examples at a time to process. There might not be minibatches every time. So, what should be done to make sure your predict make sense?

Ans

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

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

} averaged over the minibatch size.

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

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

But at test time, taking mean and variance of one example makes no sense.
So, during training time, μ and σ^2 are computed on an entire mini batch of say 64 or some other size. Then to calculate it during test time on one example at a time, just estimate μ and σ^2 from the training set through exponential weighted avg., also called running average (across mini batches).

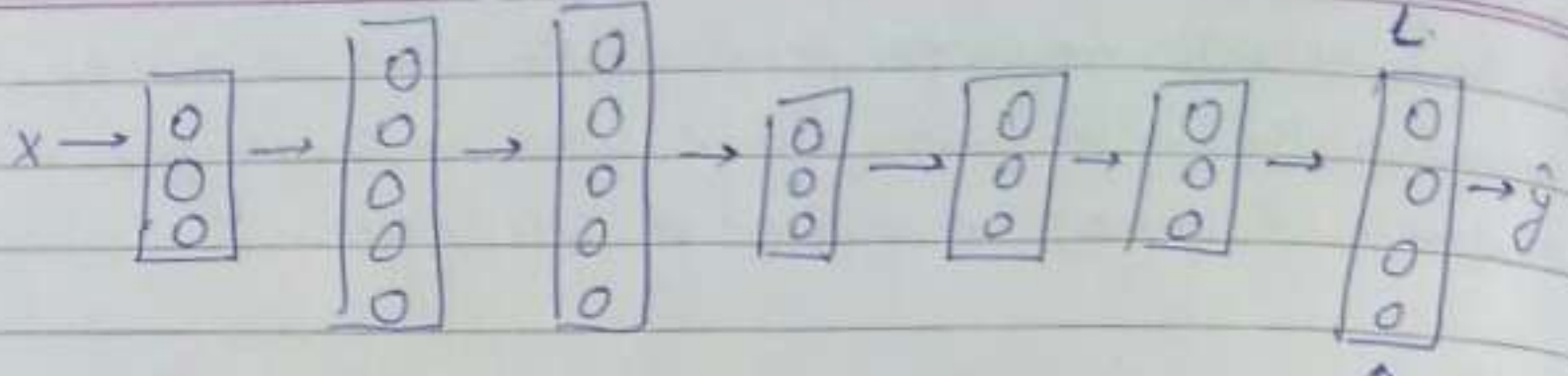
Any way to estimate μ and σ^2 for the hidden unit values z should work fine at test.

Multiclass classification:

Softmax Regression - (generalize of logistic regression to C classes.

C = # of classes

Suppose C = 4 for now, So the NN would be:



↑ (no. of units in o/p layer

The layer L is there to tell the probability of each of the 4 classes. Also the 4 nos. in output ŷ should sum to 1 as they are probabilities. To do this, softmax layer is used.

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

$\{z^{[L]} = (4,1)\text{-}\}$

1 dimension of $z^{[L]}$ vector and t vector.

Activation function:

$t = e^{z^{[L]}}$ (element wise exponentiation)

↓

temporary variable.

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

(4,1)

e.g. if $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$, $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$, $\sum_{j=1}^{4} t_j = 176.3$

$\boxed{\hat{y} = a^{[L]}} = \frac{t}{176.3}$, so, $a_1^{[L]} = \frac{e^5}{176.3} = 0.842$

$a_3^{[L]} = \frac{e^{-1}}{176.3} = 0.002$ $a_2^{[L]} = \frac{e^2}{176.3} = 0.042$

$a_4^{[L]} = \frac{e^3}{176.3} = 0.114$

So, if X image was input and for which $z^{[L]}$ came to be $\begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$, the probability of belonging to each of the classes is 0.842, 0.042, 0.002 and 0.114 respectively. $\hat{y} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$

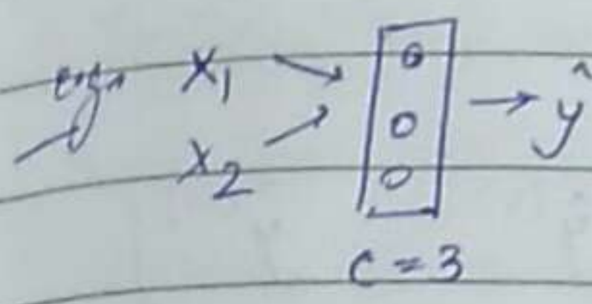
$a^{[L]} = g^{[L]}(z^{[L]})$

↓

softmax activa°

The unusual thing about this particular activa° fn is that it takes i/p as 4×1 vector

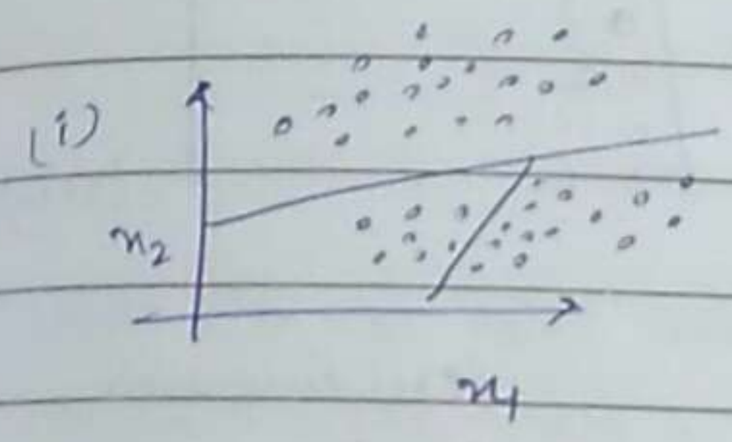
and o/p a 4×1 vector. Previously our activa^n f^n used to take in a single row value i/p like the sigmoid and the relu which i/p a real number and o/p a real number. Softmax is unique because it needs to normalize across the different possible outputs and it both inputs and outputs a vector.



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

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

↳ Softmax activa^n



= generalize^n of logistic regression with linear decision boundaries & handy more than two classes ($C > 2$).

Softmax classifier can classify data into multiple classes with linear decision boundaries as in (i) when there is no hidden layer. If there is a much deeper n/w used, then the Softmax can learn even more complex non-linear decision boundaries to separate multiple different classes.

- Training a softmax classifier : 22/8/18

Note Softmax is called so because its soft unlike the 'hardmax' which computes t to be $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ instead of $\begin{bmatrix}1484\\7.4\\0.4\\20.1\end{bmatrix}$ from using the z vector.

the highest value in z vector element is mapped to 1 and the rest are mapped to 0.

When $C = 2$, Softmax reduces to logistic regression bcoz then it computes $\begin{bmatrix}0.8\\0.2\end{bmatrix}$ for the two classes which is effectively equivalent to calculating just 0.8 (single value that LR calculates) bcoz the other value has to be 1−0.8 without any further calculn.

e.g. Suppose $y = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ and NN computes $\hat{y}$ as $\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$ which is not a good computation

$$L(\hat{y}, y) = -\sum_{j=1} y_j \log \hat{y}_j = -y_2 \log \hat{y}_2 = -\log \hat{y}_2 \quad \text{(for above case)}$$

Now, for reducing the loss $L^{(2)}$, $\hat{y}_2$ should be big and because it's a probability it should be less than 1.

So, L is the loss on a single training e.g., now the cost J on the entire training set

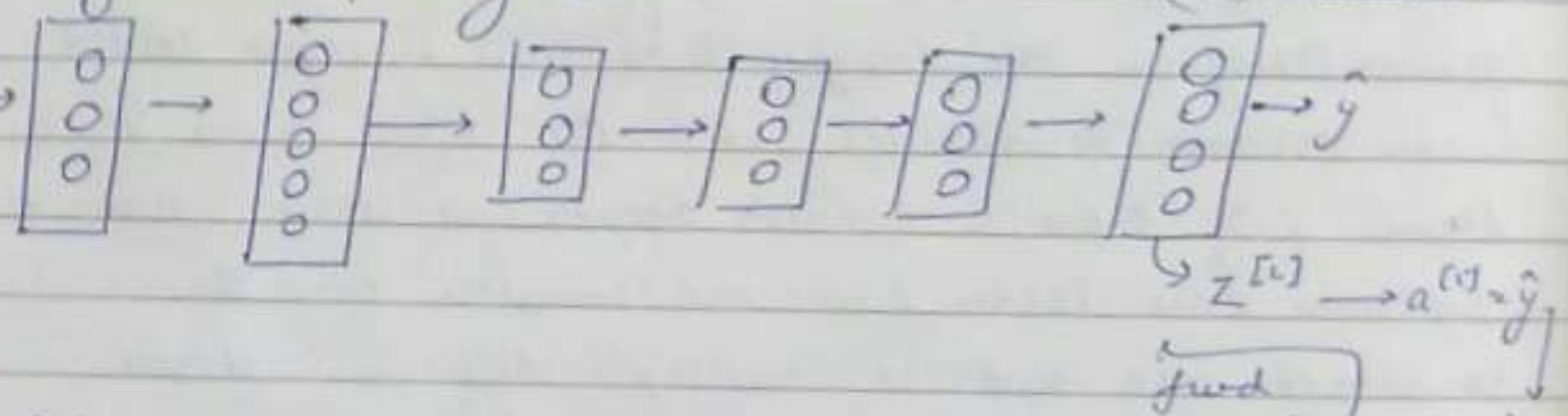
$$J(w^{[1]}, b^{[1]}, \ldots) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$. GD is then used to minimize this J.

$$Y = [y^{(1)}\ y^{(2)} \ldots y^{(m)}] = \begin{bmatrix} 0 & 1 & 0 & \\ 1 & 0 & 0 & \\ 0 & 0 & 0 & \ldots \\ 0 & 0 & 1 & \end{bmatrix}_{(4 \times m)}$$

$$\hat{Y} = [\hat{y}^{(1)}\ \hat{y}^{(2)} \ldots \hat{y}^{(m)}] \cdot \begin{bmatrix} 0.3 & & \\ 0.2 & \ldots & \\ 0.1 & & \\ 0.4 & & \end{bmatrix}_{(4 \times m)}$$

[if C=4 as in previous example]

→ GD on softmax output layer:



Now to implement back prop or GD, the key eq^n needed for initialization:

backprop: $dz^{(1)} = \hat{y} - y$ (4,1) dimensional

$$\boxed{dz^{[1]} = \frac{\partial J}{\partial z^{[1]}}}$$ (using this back prop can be implemented) to compute all the derivatives

Deep learning Programming frameworks:

These frameworks help us implement complex deep learning models more efficiently. Writing everything and everything from scratch is not practical and very efficient as using frameworks is. Some frameworks that make it easy to implement NN are:

① Caffe / Caffe 2 ③ DL4J ⑤ Lasagne ⑦ Tensorflow ⑨ Torch
② CNTK ④ Keras ⑥ mxnet ⑧ Theano ⑩ Caffe-2

- Choosing deep learning frameworks:
  - ease of programming (development and deployment)
  - running speed (some frameworks allow efficient run on large dataset)
  - truly open (open source with good governance)

→ TensorFlow :

Basic structure of tensorflow program —

Suppose we want to minimize $J(w) = w^2 - 10w + 25$ (cost)

$(w-5)^2$ and $w=5$ gives minimum values.

But let's see how we can implement something in Tensorflow to minimize this.

Code:

```
1  import numpy as np
2  import tensorflow as tf
3  w = tf.Variable(0, dtype = tf.float32)   // w is the parameter to be optimized.
4  cost = tf.add(tf.add(w**2, tf.multiply(-10, w)), 25)
5  train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
                                              (0.01 learning rate)
6  init = tf.global_variables_initializer()
7  session = tf.session()   // to start tensorflow session
8  session.run(init)        // to initialize global variables.
9  sess.run(w)              // tensorflow is evaluative variable
                            // to run the learning algorithm.
```

// train is our learning algo with GD Optimizer to minimize J.

// value for session.run(w) = 0 bcoz we haven't initialized w yet.

// run anything yet. and so the initialized value is returned for w.

// in line 5, some other optimizer can also be used.

```
10. session.run(train)   // will run 1 step of GD.
11. print(session.run(w))   // value for w = 0.1 now.
12. for i in range(1000):   // running 1000 iterations of GD now.
13.     session.run(train)
14. print(session.run(w))   // value for w = 4.999 as we would
                            // expect bcoz J was (w-5)^2.
```

// So tensorflow automatically knows how to take derivatives of add
// and multiply as well as other f^N which is why we had to
// implement only fwd prop and tf can itself figure out how to do
// the back prop or GD computation.

Also, cost = w**2 - 10*w + 25 can also be written in tensorflow
here once w is declared a tensorflow variable, +, *, -
operations are overloaded.

Note The previous example had tensorflow minimizing a fixed function
of w but one of the f^N we want to minimize is the f^N of our
training set which changes during training. So, to get training
data into a tensorflow program:

```
x = tf.placeholder(tf.float32, [3,1])        // x having 3 values
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
// so now x has become sort of like data that controls the coeff.
// of this quadratic fn.
// placeholder tells tf that x is something for which values will be
// provided later.

coefficients = np.array([[1.], [-10.], [25.]])
session.run(train, feed_dict={x: coefficients})
```

The in between steps remain the same as before code.

// So the placeholder allows us to get the training data into the J
// conveniently. feed_dict sets the coeff of the eq^n as x. this w
// it can optimize the given cost f^N by getting trained
// with changing data in just a few lines of code.

Note The cost f^N is the heart of the tf program and tf has built
in functions to compute the backward f^N for backprop and
compute derivatives for us. Thats why backprop need not be
explicitly implemented.