2 Improving deep neural networks

Andrew Ng

Note by:
Hossein Mamaghanian
amirhm@gmail.com

Improving NNs.

choice of hyper parameters are always highly iterative process

(# layers, learning rates, activation functions, ...)

Train : Model is trained on this

Dev sets / hold out / Developement sets : verify the choice of hyper parameters.

Test : check performance on unseen test sets,

→ before 70/30% ratio was reasonable

but currently with milions of data normally smaller fraction are dedicated for dev or test (98%, 1%, 1%) exp.

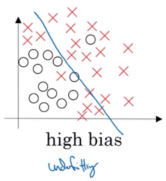
Mismatched train / test distributions,

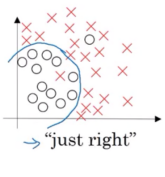
→ Make sure dev / and test sets come from the same distribution,

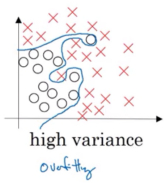
→ Not having test set might be ok, (means only dev set could be fine). Just had to be careful not to overfit on the dev set.

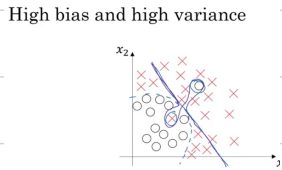
→ Bias / Variance trade off

→ example









worst of all

But in reality its not possible to plot the boundries for decision

Train set error	10%	15%	15%	0.5%
Dev set error	110%	16%	30%	1%
	high variance	high bias	high bias & variance	low bias low variance
		⊛		

→ This is with assumption that human error is ∅ (atleast there is solution) or optimal (Bays) error is zero. and Train, Dev are from same distribution,

→ if Bays error is 15% for example, 2nd column then is not bad actually(Train error 15%, Dev 16%)

→ such case is for example for highly noisy, blurry data which even human can not fully classify correctly.

Basic Recipe for machine learning:

① does it have high bias? (training data) → Bigger Net
Train longer
architecture search.

(at least we should be able to overfit, considering the Bays error is ∅).

② high variance? → More Data
Regularization to reduce overfit
architecture search

* Since before all the tools either were reducing Bias or Variance (But damaging other) it used to be called Bias/Variance trade off.
But in Big Data era and dl, we have such tools like having More Data will reduce Variance But will not affect Bias, Or Bigger Network (considering regularization is done) will help ↓Bias and not ↑ variance

Training Bigger Network never hurts (if regularization are done) just more computation.

"Regularization"

→ for reducing overfitting.

Imagine the loss function,

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

ℓ_2-norm regularization.

→ Most common ℓ_2-norm

ℓ_1-norm regularization $\frac{\lambda}{m}\sum_{i=1}^{n_x}|w|$ → w will end up being sparse

(May be useful for compressing)

→ in reality this doesn't end up really compressing model,

λ : called regularization Parameter and we use dev train to find right value.

→ (another hyper parameter

→ In NN

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

"forbinuus norm"

BP will change: $dw^{[l]} = (\text{from BP}) + \frac{\lambda}{m} w^{[l]}$

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

→ weight updates Becomes

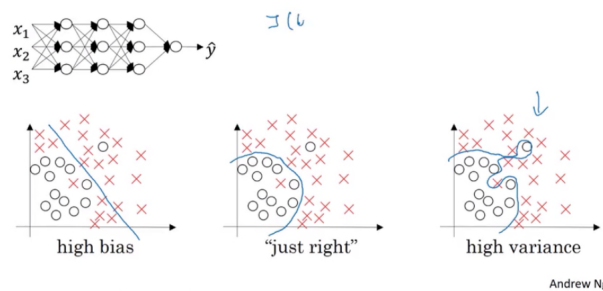
$$w^{[l]} = w^{[l]} - \alpha\left[(\text{backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

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

This means that our weights w is always shrinking, for this, l_2-norm also called "weight decay" regularization.

Why regularization helps with over fitting?

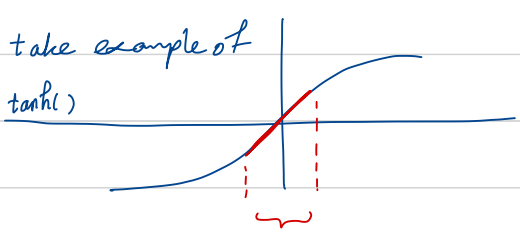
How does regularization prevent overfitting?



Andrew Ng

① . Intuition : regularization by penalizing the weights, tries to reduce impact of hidden units, thus simplifying the network if needed.

② . take example of tanh( )



$\lambda\uparrow$ $w^{[l]}\downarrow$ then $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

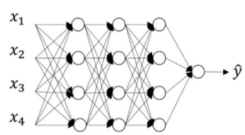
↳ then every layer ⇝ linear

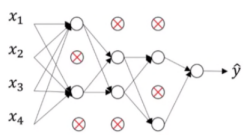
During the iteration, when we plot the $J(a, Y)$, we should consider that now we should add the $(+\frac{\lambda}{2m}\|w\|_F^2)$ term as well otherwise it may seems that it is not decreasing monotonically.

"Dropout" → "as a regularization technique"

Dropout is another technique for regularization which helps prevents over fitting

→ in each iteration, randomly some % of hidden units set to zero (turned off)





Implementing dropout ("Inverted dropout")

Illustrate with layer $l=3$. keep-prob $= 0.8$ (0.2)

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

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

→ `a3 /= keep_prob`

50 units. ⇝ 10 units shut off

$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$

↑ reduced by 20%.

$/= 0.8$

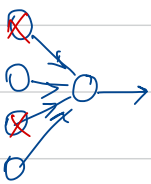
in implementation, it's important that if drop outs is used in each iteration a should be scaled to keep the same expected value.

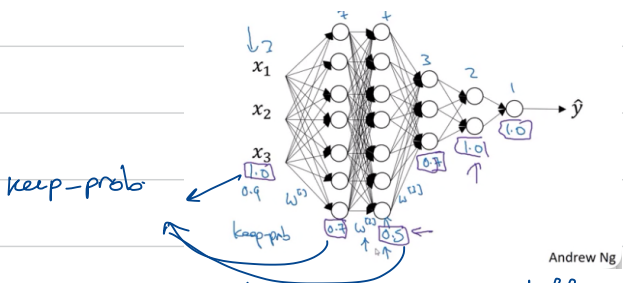
in each iteration the random dropout should regenerated not to keep the same!

dropout obviously is not implemented on the test time. (if done, it should don't change alot but it will be possibly random). or could be done few times and averaged the results (not good idea still).

"why dropout works as regularization ?" (intuition: can't rely on any feature, so have to spred the weights).

by knocking randomly the units, it force not to rely just on some and distribute the weight → shrinking weight : similar to l_2-norm





one essential parameter is keep-prob. and it can be different for each layer.

for example for bigger W we can set smaller keep-prob (0.5)

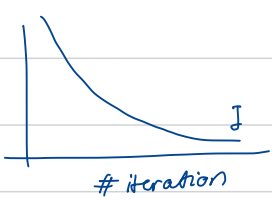
and 1 for input / output layers.

bad: More hyper parameters.

→ More is used in "cv" which data is huge but not enough. very frequely used.

bad: cost function J is not well defined any more

→ set keep-prob to 1 and check J is monotonically decreasing and then turn on drop out.



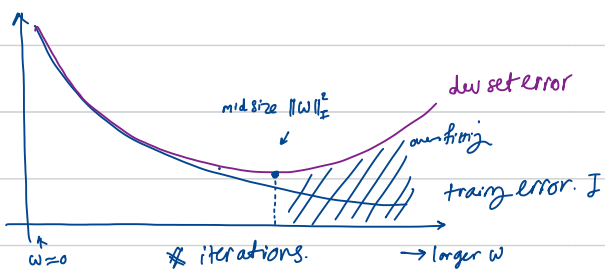
plotting J is like debuging ..

→ other techniques:

* Data augmentation
  1. fliping.
  2. croping
  3. zooming.

for digits we can add also random rotation and distortions aswell.

* early stopping:



1. optimize cost function J (gradient desent, adam, ...)
2. Not overfit : use regularization

→ these two step should (better) to be look two diffent task. First focus on 1 to chose the related hp, optimization and so, later 2. (orthogonalization) (one task at a time).

bad: early stoping break this rule (rather both are affected). alternative is to use l_2 which only focus to solve the overfitting (More h-p choice)

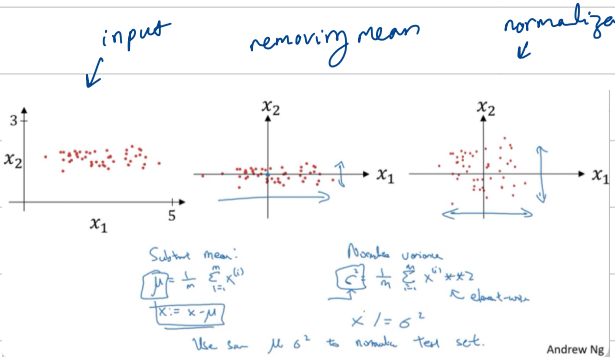
good: by raning one gradient decent process we will check all the possible options for w.

better to use l_2-norm (with diffent λ) if possible.

"Setting up optimization problem"

※ Normalizing inputs (speed up training)!

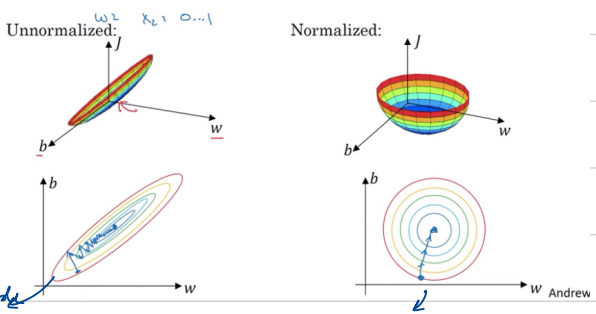
→ test set should also use the same μ, σ^2



why?

→ cost function could be very elongated.

more symetric



gradient decent has more trouble to reach local minimum

better trajectory.

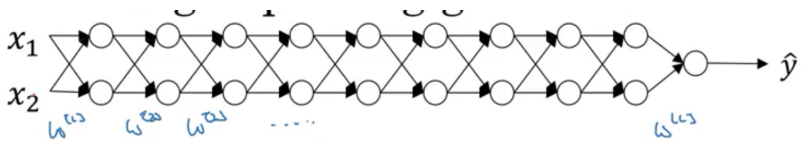
※ at least it never hurts!

Vanishing or exploding gradient.

↳ gradient becomes very small → very big.

both makes training difficult!

take example of very deep NN. (take $b=0$ and linear activation).



$\hat{y} = W^{[L]} W^{[L-1]} \cdots W^{[2]} W^{[1]} X$

→ imagine W are same $= \begin{bmatrix} a_0 & 0 \\ 0 & b_0 \end{bmatrix}$

$= W^{[L]} \begin{bmatrix} a_0 & \\ & b_0 \end{bmatrix}^{L-1} X$

then if $a_0, b_0 \rightarrow > 1 \rightarrow$ activation will explode

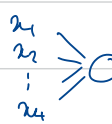
$< 1 \rightarrow$ vanishes

→ same arguments are true for gradients as well.

→ **Partial Solution**

→ random initialization of weights. if $g^{[l]} =$ Relu

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



$n \uparrow \rightarrow w_i$ should be smaller

↳ $\text{Var}(w_i) = 2/n$ (for Relu).

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

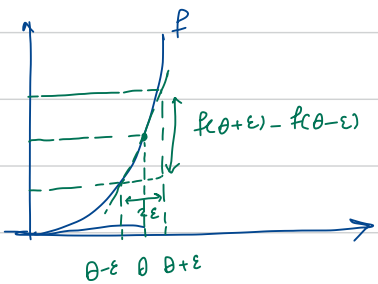
if tanh Xavier initialization $\sqrt{\frac{1}{n^{[l-1]}}}$

$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

Numerical approximation of gradients.

To double check if gd implementation is correct.

How to check derivative computation.?



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

$= 3.0001$ (calculated)

$\theta = 1$

$\varepsilon = 0.01$

$= 3 \rightarrow$ (real)

approximation error = 0.0001

(if done just 1 side error = 0.03)

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

→ error $O(\varepsilon^2)$

if we use $\lim \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$

error → $O(\varepsilon)$

→ Now this numerical gradient calculation could be used to verify if our implementation of Backprop is correct.

→ Implementation *

Take all the $W^{[1]}, b^{[1]}$ and reshape into a big vector θ.

Now the cost $J(W^{[1]}, b^{[1]}, \dots W^{[l]}, b^{[l]}) = J(\theta)$

Now all the $dW^{[1]}, db^{[1]}$ .... into a big vector $d\theta$

Gradient checking:

for each i:

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

$$\approx d\theta[i] = \frac{dJ}{d\theta^i}$$

check if they are close enough

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

for $\varepsilon = 10^{-7}$ ~ should be $\leq 10^{-7}$

$< 10^{-5}$ Maybe fine but should be checked

$> 10^{-3}$ problematic

check which value is creating such diff & and tracke the problem.

Implementation Notes!

→ grad check only for debug not in training (its very slow)

→ if grad check fails, look at components which is problematic.

↳ For example $db^{[l]}$ are very diffents but not $dW^{(l)}$, ...

→ if regularization is used sure remember to use it

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

$d\theta$ = grad of J w.r.t. θ.

→ grad check doesn't work with dropout. (obviously)

implement without dropout (keep_prob = 1.0) and check

use grad check and if pass use drop-out.

→ Run grad check at random initialization and after few train as well

if $w, b \simeq 0$ is maybe looks correct but not.

"Optimization algorithms"

Batch vs mini-batch gradient decent.

Normally size of training sets are huge, if we want to process all once and then ran just one step of gradient decent, it is not at all efficient.
Training set are divided in smaller mini_training set (mini batches).

$$X = [\underbrace{x^{(1)}\ x^{(2)}\ \cdots}_{X^{\{1\}}}\ |\ \underbrace{x^{(1001)}\ \cdots}_{X^{\{2\}}}\ |\ x^{(2000)}\ \cdots\ ]$$

Similar for the Y as well,
In each iteration one mini-batch is given and gradient decent is applied

$(i) \rightarrow$ training example index

$[\,] \rightarrow$ layer number

$\{\} \rightarrow$ mini_batch index

Batch g.d: if all the training set is used in each step of g.d.
mini-batch : if g.d ran on mini-batches.

For $t = 1, \ldots, 5000$

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

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

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

For $t = 1, \dots, 5000$

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

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

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

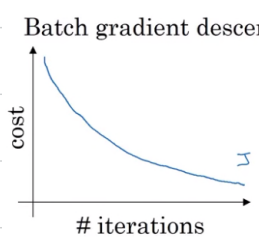
Back-prop to compute gradient w.r.t $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

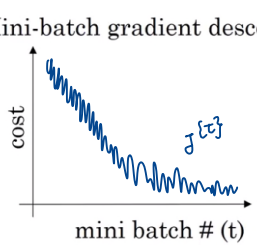
$W^{[l]} := W^{[l]} - \alpha \, dW^{[l]}$, $b^{[l]}$ → update

→ One full path over full traing set : "1 epoch"

in batch gd in each epoch 1 gd step is done while in mini-batch gd, 5000 (ex.)

Normally many epoch has to be done. (always min-batch is used obviously).



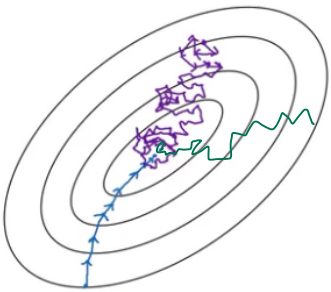


this noisier trends is that due to the fact that some mini-batchs one may be harder or easier than others.

Andrew Ng

{ mini batch size = m , Batch gradient decent (Just 1 min-batch)

" " = 1 , stochastic " " (evy example is mini-batch).



batch gd trajectory (Too long)

stochastic gradient decent trajectory

alway very noisy and will never reach to exact local minimum (will oscillate around)

(could be improved by using smaller learning rate)

advantag: gd step in each example of t.s.

→ Main disadvantage: loosing speedup from vectorization.

* Something between should be selected, (not to small / not to big).

→ Faster due to vectorization

→ faster progress on gd.

guide lines:

smaller traing set : batch gradient decent. ($m \leq 2000$)

otherwise: mini-batch size: (64, 128, ... 512)

* Make sure mini-batch fits in cpu/gpu memory. (otherwise will be slow).

mini-batch size is another hyper-parameter.

Expotentially weighted averages (Moving average).

faster then batch or mini-batch optimization.

applying a MA over data

with parameter β (≤ 1)

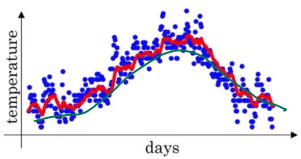
$V_t = \beta V_{t-1} + (1-\beta)\theta_t$

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

- $\beta = 0.9$: $\approx$ 10 days' temperature
- $\beta = 0.98$: $\approx$ 50 days

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

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



Exponentially weighted averages

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

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

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

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

...

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

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

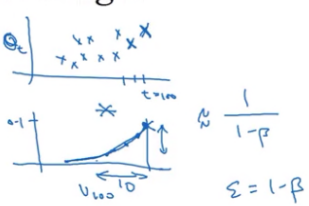
$= 0.1\theta_{100} + 0.1\times 0.9\,\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + \cdots$

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$

$(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$

$0.9^{?}$

$\varepsilon = 0.02 \rightarrow 0.98^{50} \approx \frac{1}{e}$



$\approx \frac{1}{1-\beta}$

$\varepsilon = 1-\beta$

$0.1\theta_{98} + 0.9 v_{97}$

Andrew Ng

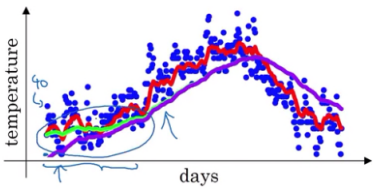
it is roughly we could say that its is averaging (weighted) over $\simeq \frac{1}{1-\beta}$ past samples.

→ Implementation.

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

in comparison with really averaging hens the advantages of speed / Memory compared to real averaging.

"Bias Correction"



$\beta = 0.98$

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

$V_0 = 0$

$V_1 = 0.98 V_0 + 0.02\,\theta_1$

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

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

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

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

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

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

Andrew Ng

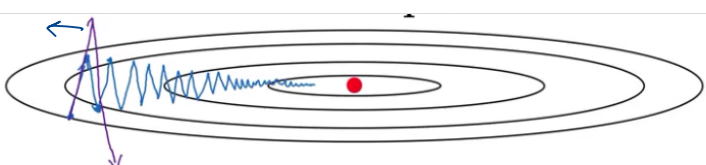
if β is close to 1 and $V_0 = 0$

Normally purple line is what we get, we apply bias correction term to reach better estimate of green line

if $V_0 = 0$ it takes more sample to reach better estimates.

adding bias correcting term $\frac{1}{1-\beta^t}$ (t = sample number)

→ "gradient decent with Momentum"



* blue line is the trajectory of gd (mini or batch)

bigger learning rate (overshoot)

what we want slower gradient on vertical axis

bigger " " horizontal "

"gradient decent with Momentum"

→ on each iteration t

compute dw, db on current mini-batch. (or batch).

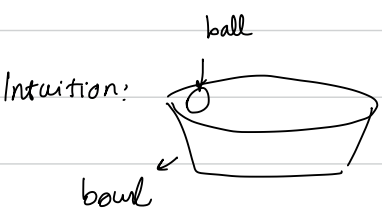
$V_{dw} = \beta V_{dw} + (1-\beta) dw$

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

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

This smooth out steps of gradient decent.

Intuition:



dw or db (acceleration)

V_{dw} or V_{db} (velocity)

β - friction

Helps get the momentum in correct direction.

Implementation details

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

In practice we dont usally add bias correction term $\frac{V_{dw}}{1-\beta}$ (it just need to to reach, why bother?)

V_{db}, V_{dw} are initialize so

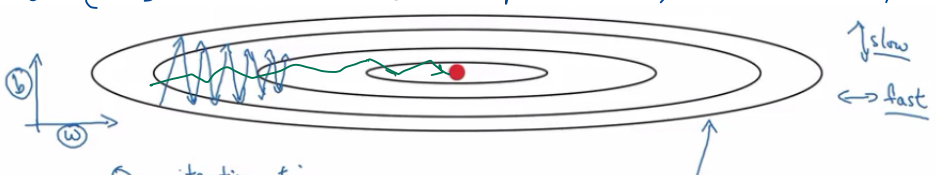
Hyperparameters: α, β $\beta = 0.9$

some even omits the $(1-\beta)$ in update : $V_{dw} = \beta V_{dw} + dw$ like integrating with α, . if used mean the α should be modified properly (better to keep with original version).

* almost always work better than g.d. without momentum.

→ RMS prop

we saw the problem of g.d when the steps are not right, for example we want gradients steps to be small in b, and larger for w (This could be for set of parameters, not necessarily all).



On iteration t:

Compute dw, db on current mini-batch

element-wise

$S_{dw} = \beta S_{dw} + (1-\beta) dW^2$ ← small → element wise power 2.

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

Similar to exponentially averaging (momentum)

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$ ←

$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$ ←

ε: for Numerical stability

S_{dw} is small then boost the steps for w

→ is big then reduces the steps for update of b.

This results in faster learning without diverging. The green line will be new trajectory. β is new hyper parameter, do not mix with the one from momentum (shown by β_2)

for Numerically stability we add ε to prevent dividing to $\emptyset$

→ Adam Optimization (Adaptive moment estimation).

→ gradient decent with momentum + RMSprop

Adam optimization algorithm

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

On iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ ← "momentum" β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db$ ← "RMSprop" β_2

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

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

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

Andrew Ng

Moment step:

RMSprop:

Bias correction is used originally.

α : need to be tuned

β_1 : 0.9 (dw)

β_2 : 0.999 (dw^2)

ε : 10^{-8}

Normally these default values are used.

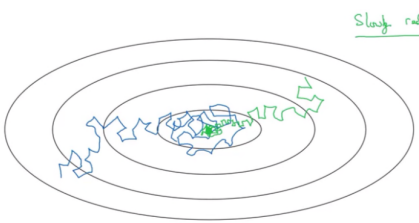
"learning rate decay"

Instead of using a fixed value for α, we reduce the α as we iterate to get closer to optimal point.

at earlier steps it is better to have $\uparrow\alpha$ to go faster but as it get closer $\downarrow\alpha$

$$\alpha = \frac{1}{1 + \text{decay rate} * \text{epoch\_num}} \alpha_0$$

→ New hyperparameter



exponential decay. $\alpha = 0.95^{\text{epoch num}} \cdot \alpha_0$

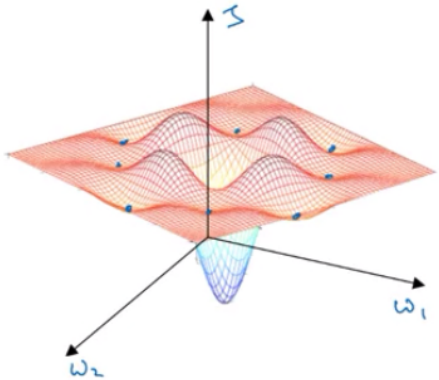
$$\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \alpha_0$$

discrete staircase decay

Manual decay : if it is just one model it is possible just to look at the error rate and adjust the α as it goes

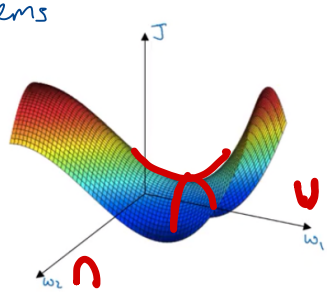
"local optima"

Previously, the intuition about the optimization problem is considered like this, One global optima with many local optima with zero gradient which could act like a trap.

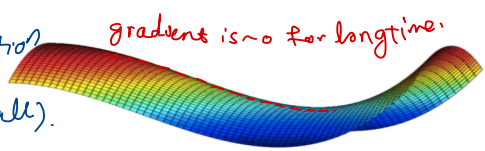


[This is correct for smaller Problems like this 2D visualization)

But for high dimentional problem, most of local optima points are actually saddle points, for high dim. Problems at each point each param should be either U or ∩ the chance all be U to be like the 2D shape is low (2^{-n}, n: # Parameters)



In higher dim. problem plateaus are Problem → (slower) the opt.) since for a large iteration the gradient is close to zero (very small).



→ local optimas are not problem is ON, rather plateaus are problem in slowing the optimization

→ momentum, RMS prop or Adam work well to solve this

"hyper Parameter tunning process"

→ We saw that there are so many hyper parameter to tune always in DNN

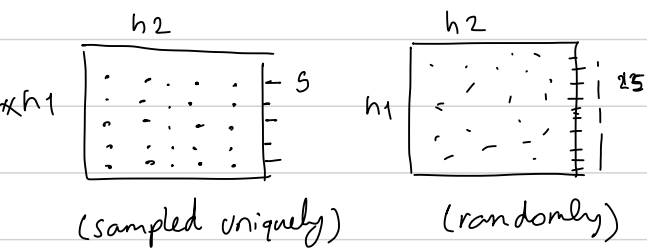
- α → the most important hyper parameter to tune.
- β
- $\beta_1, \beta_2, \varepsilon$ (for adam) — 0.9, 0.999, 10^{-8}
- # layers
- # hidden units
- learning rate decay
- mini-batch size

Most important

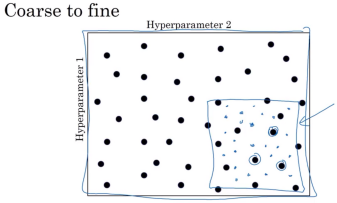
second priority

3rd priority.

→ In earlier version it was common to do a grid over hyper parameters and check. but now we more tend to sample randomly over the hyper parameter of interest.



→ if we do randomly at same number of training we have checked more possibily for each (search more efficenly over h.p. space).



while searching for best value of hp. one approch could be to go in a coarse to fine grid (start randomly more coarse and zoom in region which looks better).

→ Using an appropriate scale to pick h.p.

→ sampling uniformly random over h.p. space doesn't always an efficient way.

for example for # layer [1 ---- 4]
$n^{[l]}$ [50 ---- 100]

is not bad but for example consider α

[α : 0.0001 ---- 1]

if we sample uniformly 90% are [0.1 1] and only 10% [0.0001 0.1].

for such h.p. it is more reasonable to sample in log scale.

```
r = -4 * np.random.rand()
n = 10^r
```

h.p for exponentially weighted averages.

β : 0.9 ---- 0.999 (uniform random does not make sense).

$1-\beta$: 0.1 ---- 0.001 [10^{-1} 10^{-3}] $r \in [-3\ -1]$ → $1-\beta = 10^r$ → $\beta = 1-10^r$

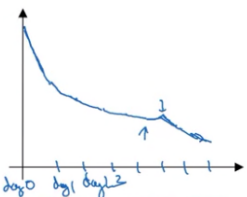
→ This helps to sample efficiently in ranges which are important.

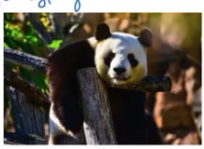
β : 0.9000 → 0.9005] roughly same average over 10

β : 0.9990 → 0.9995] chang from 1000 to 2000 huge change.

Re-test hyperparameters occasionally

Babysitting one model

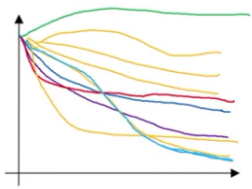




Panda ⇐

When there are alot of data but relatively less computation power. Take one model and baby sit model by observing error (dev) and change h.p. accordingly.

Training many models in parallel



Caviar ⇐

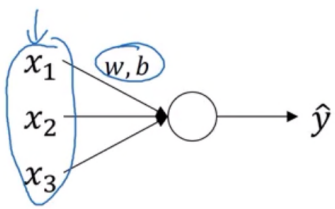
Andrew Ng

When there one enough computation power or less data we can train multiple model in parallel and chose the best one!

Batch Normalization.

→
- search of h.p easier
- robust for range of h.p
- more easy to train even more deep Networks.

Previously, we used the idea of normalizing inputs, to get more uniformed learning curves and to speedup algorithms like gradient_decent to optimize.

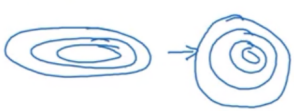


$\mu = \frac{1}{m} \sum_i x^{(i)}$

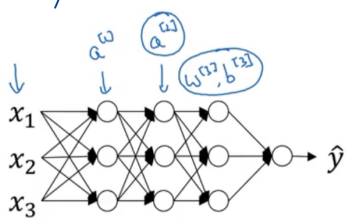
$X = X - \mu$ ← elect_wise

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

$X = X / \sigma^2$



Same idea in batch_normalization could be used for intermidiate layers as well. (There is a debate weather to use Z or A for batch_Norm, but more we use Z (before activation).



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

Normalize $z^{[2]}$

Andrew Ng

Implementation

given some intermediate values in Network :

$z^{(1)}, \dots, z^{(n)}$ — all from layer[l]

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

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

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

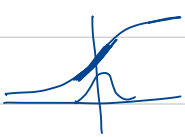
ε : for numerical stability

$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$ (γ, β learnable p. of model).

Now use $\tilde{z}^{[l]}$ instead of $z^{[l]}$

→ This helps to control the mean and variance of values.

$\gamma = \sqrt{\sigma^2 + \varepsilon}$
$\beta = \mu$
} reverse the operation.
$\tilde{z} = z$

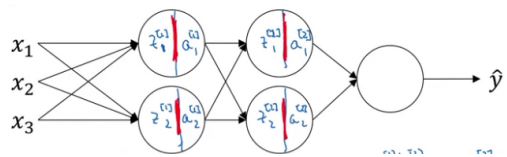
for example if we are using sigmoid activation, 
we dont want to distribute all around the linear part and we use γ, β to use more of non linearity of sig.

→ by Normalizing the intermediate layers b.n. helps speedup optimization similar to normalizing the inputs.

"Adding b.n. to a Network"

→ adding the B.N is as follow

in every layer z (output before activation) is normalized.



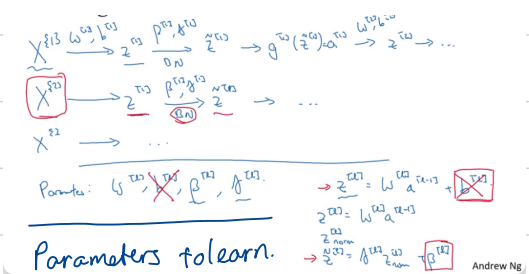
$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{B.N}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[\text{B.N}]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]}$

Parameter to be updated $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots$

and new b.n ones $\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots$ → these are also updated

similar to w, b $\quad \beta^{[l]} \leq \beta^{[l]} - \alpha d\beta^{[l]}$

batch normalization works with mini-batches as well.



→ Now, having b is really not necessary since in b.n step we are re-adjusting the bias. we can remove b

Implementing gradient descent

for $t = 1 \ldots$ num Mini Batches

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 $w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$, $\beta^{[l]} := \beta^{[l]} - \alpha\, d\beta^{[l]}$, $\gamma^{[l]} := \ldots$

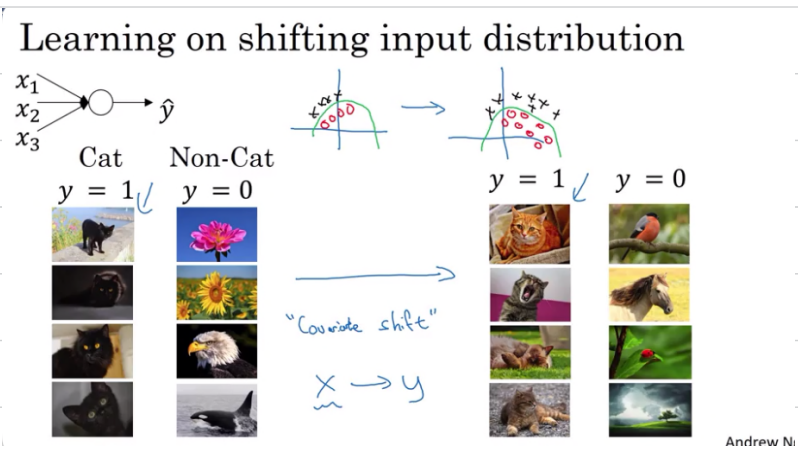
Works w/ momentum, RMSprop, Adam.

B.N work with / momentum, RMSprop, Adam, ..

→ Why B.N works? How speed ups optimization?

*→ Similar to normalization of input could speed up learning (all values are similar scale and learning outcom is more uniform).

*2: It make the weights in deeper layers more robust to changes in weights in earlier layers of N.N.



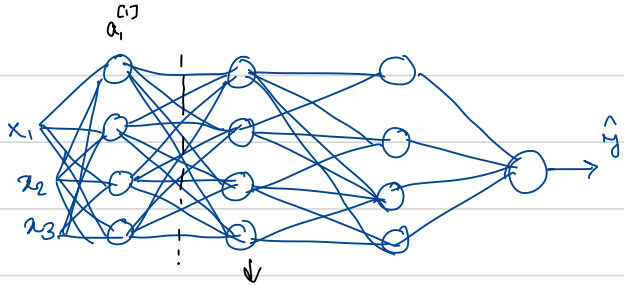
for example if we have trained our net on left data (black cats), Now will have problem with new data of changed distribution (other color).

Andrew N

like a model learned on left could not generalize to right even though it is same function.

→ This is called "covariate shift" : if you learned a maping $X \rightarrow Y$ if the di's of data changed, You should re-learn the mapping.

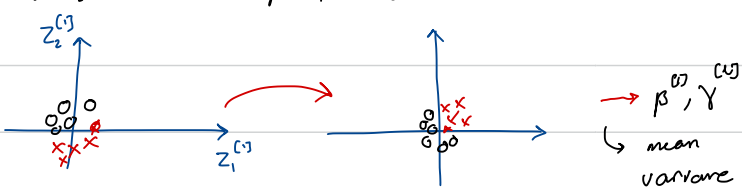
imagine a NN



from a prospective of this hidden layer, it receives $a^{[1]}$ as input and tries to map it to output.

but indeed since weight in previous layers are changing then, distribution of $a^{[1]}$ are also chanj.

b.n ensure that no matter how weight are changj at least it keeps the distribution same.



it limits the amount of effects of changing weights in previous layers, could chany distribution of out

→ (weakins the dependency of layers learning).

Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect. (regularization effect is not that much, better to be used with dropout. smaller b.s has more regularization but dont use b.n as regularization.

→ drop out has multiplicitive noise, b.n. has additive noise

applying in B.N at test time:

→ B.N is done as follow on forward path on training:

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

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

$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \quad (*)$$

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

μ, σ are calculated on mini-batch.

but on test time we may not have a mini-batch, we tend to estimate μ, σ over mini-batch one time on traing time.

→ μ, σ^2 : estimate using exponentially weighted average (across mini-batch)

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

↓

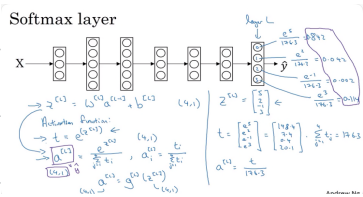
$\mu^{\{1\}[l]}$ $\mu^{\{2\}[l]}$ $\mu^{\{3\}[l]}$ → average $\mu^{[l]}$

→ σ^{2}

then at test time instead of using formula in $(*)$ use:

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

Softmax layer



in multi class, classification, we could use an output vector representing each class

→ in this case we use softmax layer at output.

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

Activation function:

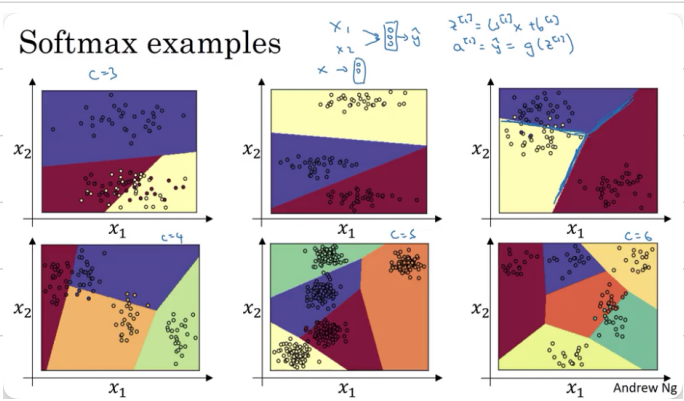
→ $t = e^{(z^{[l]})}$

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{j=1}^{4} t_i}$$

$$z^{[l]} = \begin{bmatrix} 5 \\ -2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \qquad \sum_j t_i = 176.3$$

$$a^{[l]} = \frac{t}{176.3} = \begin{bmatrix} 0.84 \\ 0.04 \\ 0.002 \\ 0.114 \end{bmatrix}$$

→ decision boundaries softmax represents:



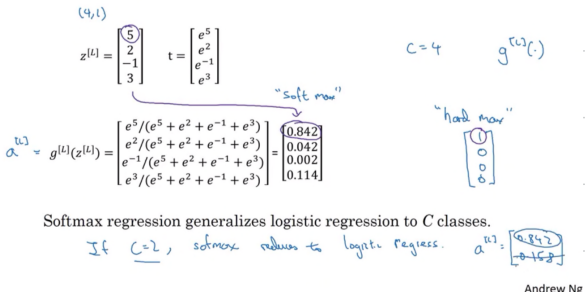
a NN with no hidden layer

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

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

Multi-class classification

Understanding softmax



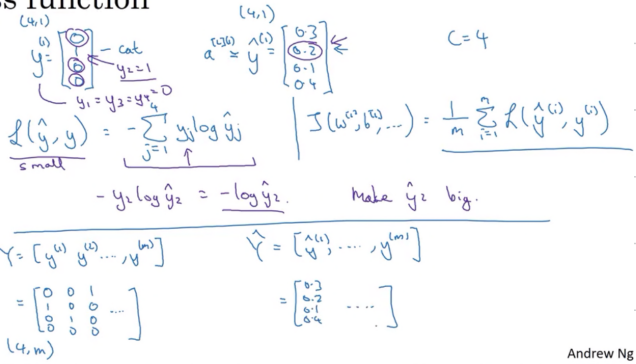
Softmax regression generalizes logistic regression to C classes.

If C=2, softmax reduces to logistic regress. $a^{[L]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$

Andrew Ng

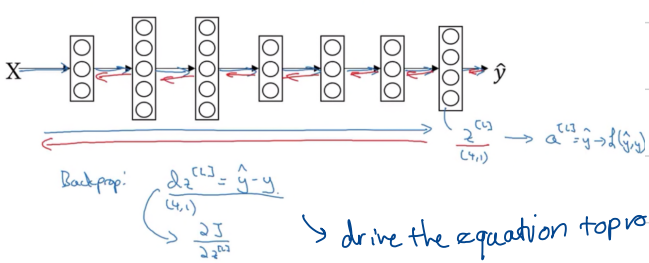
Loss function

Maximum likelihood loss function is used



Andrew Ng

Gradient descent with softmax



↳ drive the equation to prove!

Andrew Ng