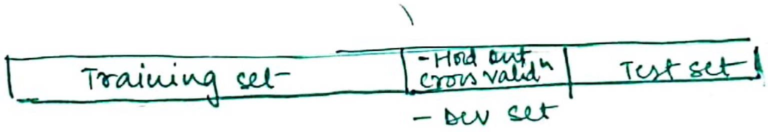
Improving Deep Neural Networks: Hyperparameters tuning, Regularisation, Optimisation:

* Train / Dev / Test sets:

Data 

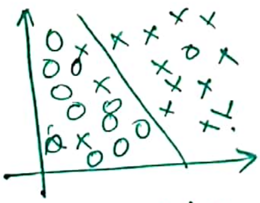
70/30 %
Train test

60/20/20
Train dev Test.

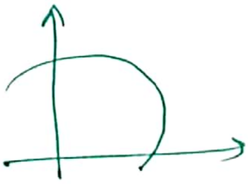
Big data: 98/1/1
Train dev Test.

- Make sure dev and test set come from same ~~resolution~~ distribution.

* Bias / Variance:



high bias.



just right

high variance
overfitting

Train set error 1% → very good
Dev set error 11% → relatively poor } High variance

Train set error 15%
Dev set error 16% } underfitting / High bias

15%
30% } High bias & High variance:

HB HV

under & overfitting
both occur.

0.5%
1% } low bias
low variance.

Human ≈ 0%.

Bayes / optimal error ≈ 0%.

if 15%
High bias

High bias ? (training data performance) $\xrightarrow{Y}$ $\boxed{\text{Bigger N/W}}$ Train larger. NN usage.

$\downarrow$ N.

High var ? (dev set performance) $\xrightarrow{Y}$ More data Regularisation

"Bias variance tradeoff"

→ in new deep learning era, tradeoff has been ~~ero~~ nullified

Regularisation of Neural NW

→* logistic reg :

$$\min_{(w,b)} J(w,b)$$

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

L_2 regularisation $\|w_j\|^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L_1 regularisation $\frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

$\lambda \to$ regularisation parameter.

→* Neural NW : $J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{n} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2.$

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

F : "Frobenius norm"

$dw =$ (from backprop)

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

$$\frac{dJ}{dw^{[l]}} = dw^{[l]}$$

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

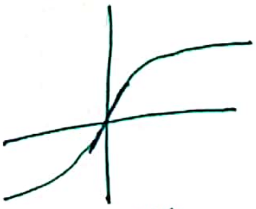
$$w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha(\text{backprop})$$

$$\left(1 - \frac{\alpha\lambda}{m}\right)$$

* Regularisation vs overfitting

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

$$w^{[l]} \approx 0$$



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

$\lambda\uparrow \quad w^{[l]}\downarrow$

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

Every layer ≈ linear
can lead to high bias.

* Dropout regularisation.

keep 0.5 chance of keeping 1 node, 0.5 chance of removing
then train a smaller network.

implementation :

- inverted dropout :
  $l=3$:

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

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

a3 /= keep-prob

Ex: 50 units → 10 unit shut off

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

↳ reduce by 20%.
$/= 0.8$

- Making predictions at x time

$a^{[0]} = X$

no drop out

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

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

⋮

/= keepprob
≠
dropout.

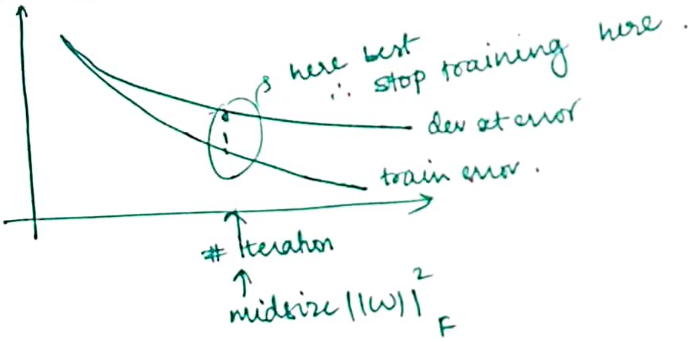
Scanned by CamScanner

Smaller keep-probs makes more powerful drop outs.

→ dropout helps prevent overfitting
→ downside : 'J' not well defined
∴ less calculatable.

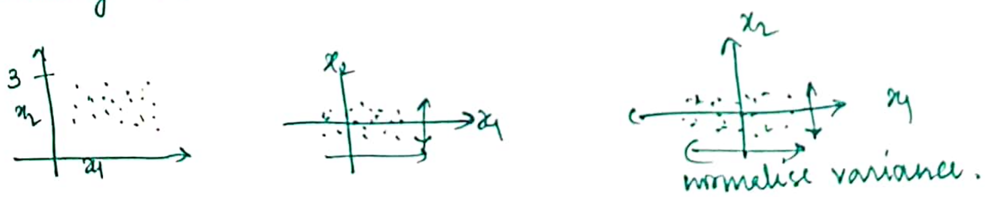
other regularisation techniques:

- Data augmentation
  → make TS more redundant
  → random distortions
  → ...
- Early stopping



Setting up optimisation problem:

Normalising train sets.



Subtract mean:

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

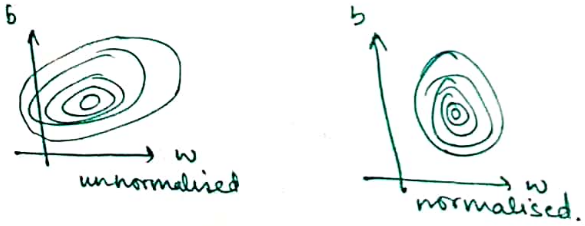
$$x := x - \mu.$$

Normalize var.

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

↳ element wise

$$x /= \sigma^2.$$



* vanishing / Exploding gradients :

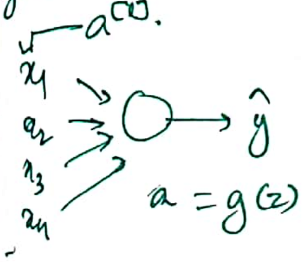
→ when training deep NN, grads. can become very large or very small.

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

$\hat{y} = w^{[L]} w^{[L-1]} \dots \underbrace{w^{[2]} \underbrace{w^{[1]} x}_{z^{[1]}}}$

$\dots \quad a_3 \quad a_2$

* weight initalisation :



$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$.

larger n → smaller w_i

$var(w_i) = \frac{1}{n}$.

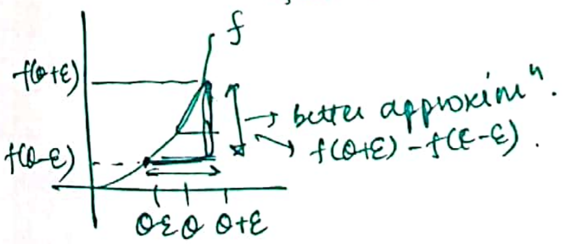
$w^{[l]} =$ np. random. randn (shape) * np.sqrt $\left(\frac{2}{n^{[l-1]}}\right)$ → Relu

tanh : $\sqrt{\frac{1}{n^{[l-1]}}}$

↳ xavier initalisation.

* Numerical approx of gradients :-

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



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

$\frac{(1-0.1)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3$.

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

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

0.01

0.0001

$\frac{f(\theta+\epsilon) - f(\theta)}{\epsilon}$ → less accurate.

* **Gradient checking:**

$w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}$ reshape into big vector θ.

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

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

is $d\theta$ is grad of 'J' ?

grad check:

$J(\theta) = J(\theta_1, \theta_2, \ldots, \theta_n)$

for each i:

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

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

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

$\varepsilon = 10^{-7}$ ✓. correct initialisations.

10^{-5} : check

10^{-3} : worried.

- implementing:
  - only to debug:
    $d\theta_{opp}[i]$ $\quad$ $d\theta$.
  - if algo fails grad-check, look at comp. to find bugs.
    $db^{[l]}$ $\quad$ $dw^{[l]}$
  - Remember regularisation

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

    $d\theta$ = grad of J wrt θ.
  - Doesn't work with dropouts. keepprob = 1.0
  - Run at random initialisation; perhaps after some training.

. If NN seems to have high bias, then
- ↑se no of units in each hidden layer.
- Make NN deeper

0.5% ↑s error
0.8%. dev set error

. → ↑se λ
→ get more training data.

. λ ↑es ⇒ weights → 0

. ↑sing keep-prob param ↓es regularisation effect
↓es train set error.

. Dropout, Data augmentation & L2 regularisn ↓se variance.

. normalise 'x' so that cost funcn faster to initialise.

Optimisation Algos:

* Mini-batch gradient descent:-

$$X = [x^{(1)} \; x^{(2)} \; x^{(3)} \ldots]$$
(n_x, m)

$$Y = [y^{(1)} \; y^{(2)} \; y^{(3)} \ldots]$$
$(1, m)$

if $n = 50,000,000$?

div. to mini-batches.

1000 - each.

$$X = [\underbrace{x^{(1)} \; x^{(2)} \ldots x^{(1000)}}_{\substack{X^{\{1\}} \\ (n_x, 1000)}} \mid \underbrace{x^{(1001)} \ldots x^{(2000)}}_{\substack{X^{\{2\}} \\ (n_x, 1000)}} \mid \ldots]$$

$\ldots \quad X^{\{5000\}}$

$$Y = [\underset{(1,1000)}{Y^{\{1\}}}, \underset{(1,1000)}{Y^{\{2\}}}, Y^{\{3\}}, \ldots]$$

mini batch t : $X^{\{t\}}, Y^{\{t\}}$.

mini batch GD: 1 step of GD using $X^{\{t\}}, Y^{\{t\}}$

for t = 1,..,5000
FWD propagn on $X^{\{t\}}$
$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
$A^{[1]} = g^{[1]}(Z^{[1]})$

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

Backprop to compute gradients wrt $J^{\{t\}}$

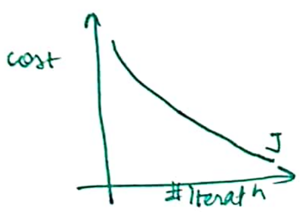
$$w^{[l]} = w^{[l]} - \alpha dw^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha dw^{[l]}$$

}

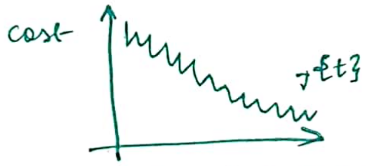
"1 epoch"
↳ pass through Training set.

* MBGD understanding :

Batch GD.



MBGD

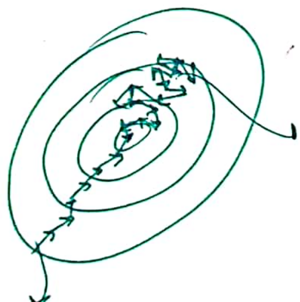


#t (mini batch)
Plot $J^{\{t\}}$ using $X^{\{t\}}, Y^{\{t\}}$.

Chosing size :

If minibatch size = m : Batch GD $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

if mini batch size = 1 : Stochastic GD
Every example is mini batch.
$(X^{\{t\}}, Y^{\{t\}}) = (X^{(1)}, Y^{(1)})$ = mini batch.



practice : 1 – m batch size

Batch GD
Minibatch : m
↓
Too long.

Stochastic GD.
↓
lose speedup from vectorization.

In between
↓
Fastest learning
- Too much vectorizⁿ.
- no waiting.

* If small TS : use batch GD.
(m ≤ 2000).

Typical mb size is 64, 128, ...
2^6 2^7 2^8 2^9

pow of 2 is faster.

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

* Exponentially weighted averages :

$V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

$V_3 = 0.9 V_2 + 0.1 \theta_3$

$\vdots$

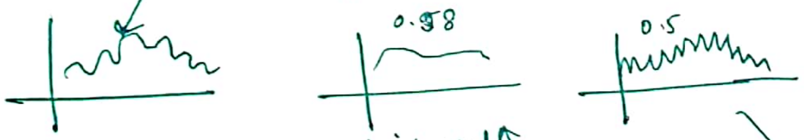
$V_t = 0.9 V_{t-1} + 0.1 \theta_t$.

$\frac{1}{1-\beta}$ days.

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

$\beta = 0.9$: more weight to V_{t-1}

$\frac{1}{1-0.9} \approx 10$ days temp



$\frac{1}{1-0.98} \approx 50$ days

high val
more days
smooth curve.

$\frac{1}{1-0.5} \approx 2$ days.

* Understanding exponentially weighted average.

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

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

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

$\vdots$

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

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

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

$\varepsilon = 1-\beta$.

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $\frac{(1-\varepsilon)^{1/\varepsilon}}{0.9} = \frac{1}{e}$

$0.98^{50} \approx \frac{1}{e}$ (for exponential decay) implementation

$V_0 = 0$

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

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

$V_3 = \beta V_2 + (1-\beta)\theta_3$

...

$V_\theta = 0$

Repeat {

get next θ_t.

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

}

starts very low ∴ Bias error.

$V_1 = 0.9 V_0 + 0.02\theta_1$.

$V_0 = 0$

$V_1 = 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2$

$V_2 = 0.0196\,\theta_1 + 0.02\theta_2$

$\dfrac{V_t}{1-\beta^t} \rightarrow$ Bias correction.

$t = 2$:

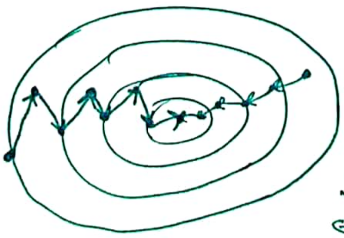
$1-\beta^t = 1-(0.98)^2 = 0.0396$

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

t large → estimate corrected.

* Gradient descent with momentum :

- works faster.



momentum :

on iteration t :

Compute dw db on current mini batch.

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

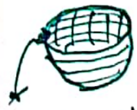
"$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"

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

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

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

vert dirⁿ : average out to zero.
horizontal : quick osc. in horizontal dirⁿ.



A ball ~~reflect~~ rolls down the bow

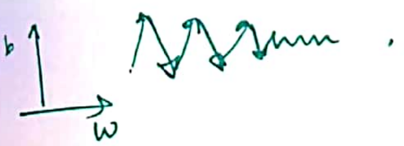
Hyperparam α, β. $\beta = 0.9$.

$avg = \frac{1}{1-\beta}$.

Init :
$V_{dw} = 0$
$V_{db} = 0$

$V_{dw} = \beta V_{dw} + dw$ ←
$V_{db} = \beta V_{db} + db$. ← leave out $(1-\beta)$.

RMS Prop :
- root mean square prop.



on iteration t :
compute dw, db on current mini batch.
$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$ ← element wise. ← relatively small
$S_{db} = \beta S_{db} + (1-\beta) db^2$ ← relatively large.

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}}}$ ↓ bigger $(+\varepsilon)$ → 10^{-8} mostly neglected.

$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$ ↓ smaller. $(+\varepsilon)$ → 10^{-8} mostly neglected.

after rms prop :

Adam optimezion algorithm :

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

on iteration t :
compute dw, db using current MB.
$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$. ← Momentum
$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$ $S_{db} = \beta_2 S_{db} + (1-\beta_2) db$. ← RMSprop
$V^{corrected}_{dw} = V_{dw} / 1-\beta^t$ $V^{corrected}_{db} = V_{db} / 1-\beta_1^t$

Scanned by CamScanner

$S_{dw}^{correct} = S_{dw} / (1-\beta_2^t)$

$S_{db}^{correct} = S_{db} / 1-\beta_2^t$.

$w := w - \alpha \dfrac{V_{dw}^{correct}}{\sqrt{S_{dw}^{corr}} + \varepsilon}$.

$b := b - \alpha \dfrac{V_{db}^{corr}}{\sqrt{S_{db}^{corr}} + \varepsilon}$

~~Hap~~ Hyperparams choice:

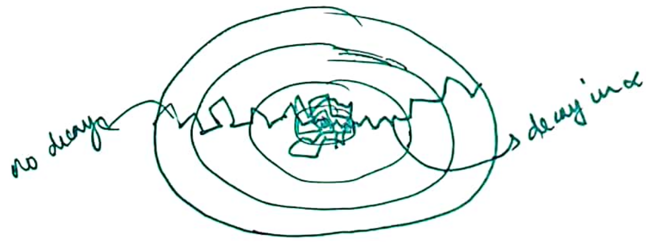
- → α: needs to be tuned
- → β_1: 0.9 (dw)
- → $\beta_2 = 0.999$ (dw^2)
- → ε : 10^{-8}.

Adam: Adaptive moment estim^n

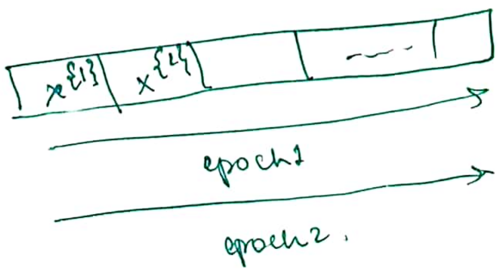
$\beta_1 \to 1^{st}$ moment

$\beta_2 \to 2^{nd}$ moment.

* learning rate Decay:



1 epoch = 1 pass through data.

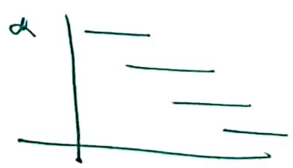


$$\alpha = \frac{1}{1 + \text{decay-rate} * \text{epochnum}} \alpha_0$$

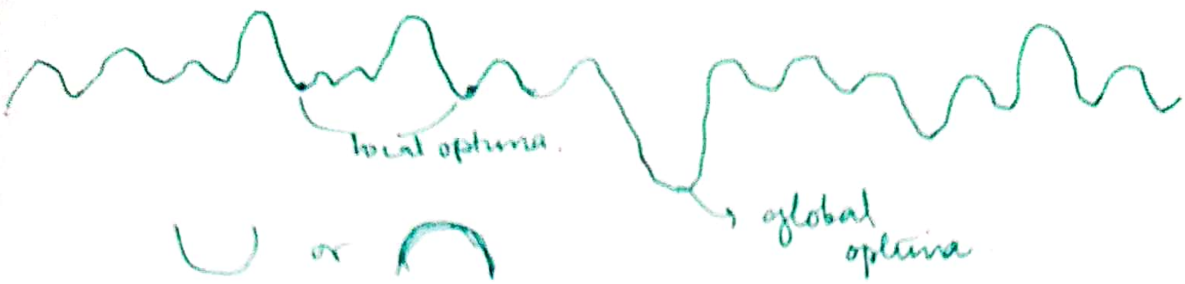
Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4

$\alpha = 0.95^{epochnum} \cdot \alpha_0$ ⇒ exp. decay

$\alpha = \dfrac{k}{\sqrt{epochnum}} \cdot \alpha_0$ or $\dfrac{k}{\sqrt{t}} \cdot \alpha_0$



* the <u>problem of local optima</u>:



saddle pt: pt where derivative = 0.

problem of plateaus:

→ grad ≈ 0 for a long time.

• Plateaus can make learning slow.

Hyperparameter tuning :-

\# Process :

$\alpha \rightarrow$ most imp H.P.

$\left.\begin{array}{l}\beta \sim 0.9 \\ \beta_1, \beta_2, \varepsilon\end{array}\right\} \rightarrow$ significant

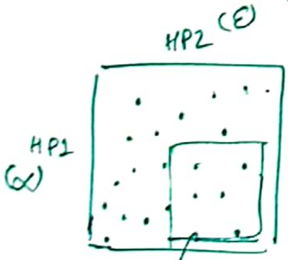
\# layers

\# hidden units $\} \rightarrow$ significant

learning rate decay

mini batch size $\} \rightarrow$ significant.

Try random val : no grid.

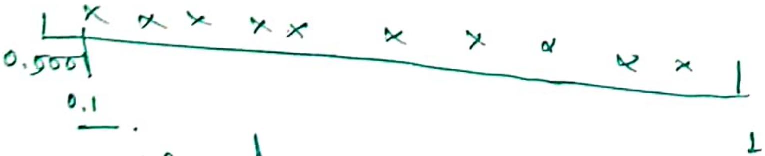


Coarse to fine searches

* using appt scale to pick hyperparams :-

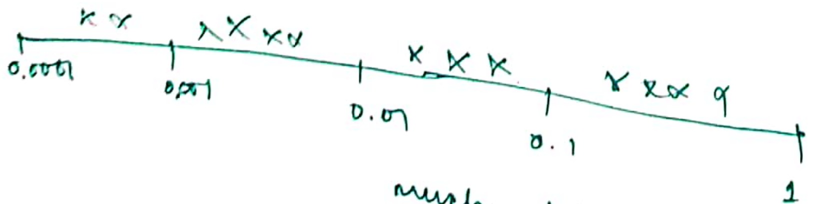
- Picking HP at random

$\alpha = 0.0001, \dots, 1.$



not good.

⇓



much better distbn.

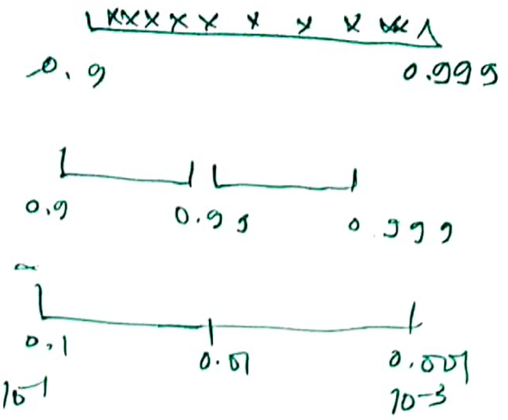
$r = -4 * np.random.rand()$ $\leftarrow$ $r \in [-4, 0]$

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

$\beta = 0.9$ ---- 0.999

$\frac{1}{10}$ $\qquad$ 1000

$1-\beta = 0.1$ ---- 0.001

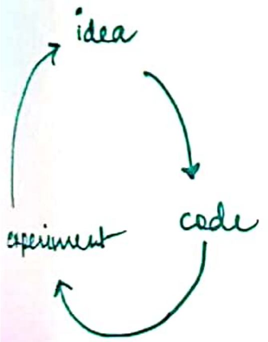


$r \in [-3, -1]$

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

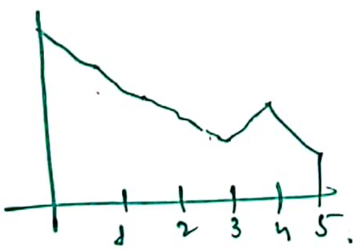
$\beta = 1 - 10^{r}$

Hyperparams tuning in practice (Pandas vs caviar)



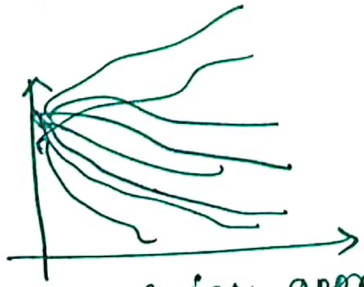
- NLP, Vision, Speech, ..
- Intuitions do get stale.

Babysitting one model: monitoring learning rates one at a time.



pandas approach. (like panda)

Training many model in parallel.



caviar approach (like fish)

Batch normalisation :

* Normalization of activations in a N/W.

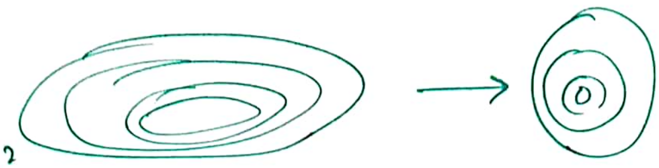
LR:

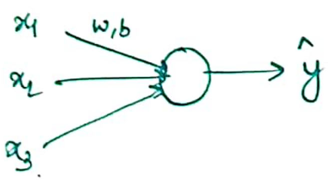
$\mu = \frac{1}{m} \Sigma x^{(i)}$

$X = X - \mu$

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

$X = X/\sigma^2$





normalise $a^{[2]}$ so as to train $w^{[3]}, b^{[3]}$ faster.

implementing :

$$\underbrace{z^{[1]}, \ldots, z^{[m]}}_{z^{[l](i)}}$$

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

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

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

$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$. γ & β : learnable params.

$$\boxed{\begin{array}{l} \text{if} \\ \quad \gamma = \sqrt{\sigma^2 + \varepsilon} \\ \text{then } \beta = \mu. \\ \quad \tilde{z}^{(i)} = z^{(i)} \end{array}}$$

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

Fitting Batch norm into a NN.

~~Batch norm~~

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

params: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$ } $d\beta^{[l]}$

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

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

$b^{[l]}$ $\quad$ $\beta^{[l]}$
$(n^{[l]}, 1)$ $\quad$ $(n^{[l]}, 1)$

implementing GD:

for $t = 1 \dots$ no of MB:

Compute forwardprop on $x\{t\}$

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

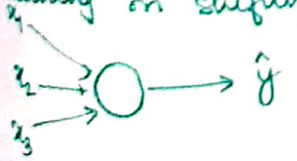
use backprop to compute $dw^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$

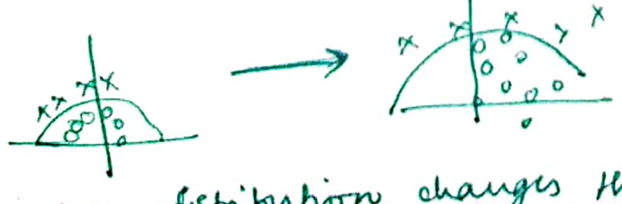
update params
$$\left.\begin{aligned} w^{[l]} &= w^{[l]} - \alpha dw^{[l]} \\ \beta^{[l]} &= \beta^{[l]} - \alpha d\beta^{[l]} \\ \gamma^{[l]} &= \text{---} \end{aligned}\right\}$$

works on momentum, RMS prop, Adam.

* Why Batch norm works?

* learning on shifting i/p distribution





data distribution changes through covariate shift-
if distb^n changes from $X \to Y$, retrain algorithm

- batch norm ↓es the amt by which hidden values shift. Thus, the covariate shift
- batch norm makes sure variance of $z_2^{[2]}$ $z_2^{[2]}$ & $z_2^{[1]}$ remain same.
- makes learning of later layers easier
- Also has a slight regularisation effect. (adds noise to hidden layers)
  ⇓
  unintended side effect.
- After training NN with batch norm, perform needed norm. using μ & σ^2, estimated using exponentially weighted avg across minibatches at test time.

* Batch norm at test time :
⇓
1 minibatch at a time

but at ~~test~~ test time, we need to process examples one at a time.
⇓
estimate μ & σ^2 from a T.S.

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

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

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

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

μ, σ^2 : estimate using exponentially weighted average (across MB)

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

$\downarrow \qquad \downarrow$

$\mu^{\{1\}[l]}$, $\mu^{\{2\}[l]}$, ... $\longrightarrow \mu$

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

$\sigma^{2\{1\}[l]} \qquad \sigma^{2\{2\}[l]}$ . . .

* ε used to avoid division by zero.

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

Multiclass classific^n :

* Softmax Regression

Ex :

C = # no of classes = 4 $(0, \dots, 3)$

layer [L]

$X \rightarrow$ . . . .

$\rightarrow P(\text{other} \mid X)$

$\rightarrow P(\text{cat} \mid X)$

$\rightarrow \hat{y}$

$\rightarrow P(\text{dog} \mid X)$

$\rightarrow P(\text{baby} \mid X)$

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

Act^n funct^n

$$t = e^{z^{[L]}}$$

(4,1)

Scanned by CamScanner

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

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

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

$$\sum_{j=1}^{4} t_j = 176.3$$

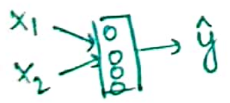
$$a^{[L]} = \frac{t}{176.3} \qquad a^{[L]} = \frac{e^5}{176.3} = 0.842. \quad \ldots$$

$\Downarrow$

Chance of other

4 probabilities that sum to '1'.

Softmax examples:



$$z^{[L]} = w^{[L]}x + b^{[L]}$$
$$a^{[L]} = \hat{y} = g(z^{[L]})$$

Training a softmax classifier:

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

"softmax" "hardmax"

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5 / e^5+e^2+e^{-1}+e^3 \\ e^2 / e^5+e^2+e^{-1}e^3 \\ e^{-1} / e^5+e^2+e^{-1}+e^3 \\ e^3 / e^5+e^2+e^{-1}+e^3 \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Loss function:

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \text{cat.} \qquad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \qquad C=4$$

$$\underbrace{L(\hat{y}, y)}_{\text{small}} = \underbrace{-\sum_{j=1}^{4} y_j \log \hat{y}_j}_{\text{make log } \hat{y}_2 = -\log \hat{y}_2}$$

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

$\hat{y}_{(4,m)}$ ...

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

$= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 & \ldots \\ & \vdots \end{bmatrix}$

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

Backprop:

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

$\rightarrow \frac{dJ}{dz^{[L]}}$

⊕ DL prog. frameworks :

* Tensorflow :

Ex: $J = w^2 - 10w + 25 = (w-5)^2$

min at $w = 5$.

```
import tensorflow as tf

w = tf.Variable(0, dtype = float32)
cost = tf.add(tf.add(w**2 , tf.multiply(-10., w)), 25)
train = tf.train.GradientDescentOptimizer(0.01).minimise(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
sess.run(w)

session.run(train)    # runs 1 step of GD
print(session.run(w))    # 0.1

for i in range(1000):
        session.run(train)
print(session.run(w))    # 4.999   -> working.
```

Scanned by CamScanner

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

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

⋮

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

⋮

session.run (train, feed_dict= {x: coefficients})

⋮

Structuring ML Projects:

Introduction to ML ~~strategy~~ strategy.

* orthogonalisation
  - what to tune in order to acheive one effect, people are clear about this.

tune ~~measure~~ different parameters differently

↑
L→

chain of assumptions in ML:

different tuners for different assumptions independent of each other ⇓ orthogonalisation.

- Fit training set well on cost func^n. → bigger N/W → Better opt. algo
  ↓
  Fit dev set well on cost func → Regularization → Bigger training set
  ↓
  Fit test set well on cost func^n. → bigger dev set
  ↓
  Performs well in real word. → change dev set or cost func^n.

* Single number evaluation metric.

idea → code → exp → idea

	precision	recall
A	95%	90%
B	98%	85%

precision → of examples as cats, what % are cats

recall → what % of actual cats recognised.

F1 Score : "Avg" of precision P & recall R

$\left(\frac{2}{\frac{1}{P}+\frac{1}{R}}\right.$ "Harmonic mean")