Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

– Andrew Ng

Applied NN is a highly iterative process.

↙

#layers, #hidden units, learning rates, activation fun's

Data

Ratio → 60%. | 20%. | 20%. or 80%. | 10%. | 10%.

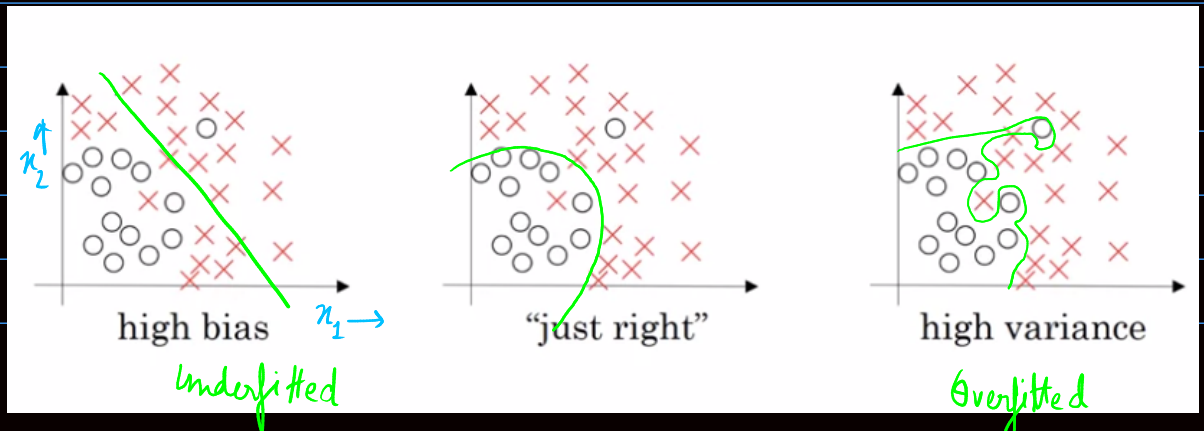
Tip. * For very big data set make this split smaller. 10,000 images are enough for dev & test set.

Mismatched Train/Test Distribution

* Make sure the dev and test sets come from the same distribution

* Not having a test set might be okay. (Only Dev set)

BIAS / VARIANCE



Cat classification

① Train Set Error = 1%. Dev Set Error = 11%. → High variance. or Overfitted

② = 15%. = 16% → High Bias or Underfitted.

③ = 15%. = 30%. → H.B & H.V

④ = 0.5%. = 1%. → L.B & H.V

Optimal or Bayer's Error = 0%. {Human Error}

If Optimal Error is 15%. → ② ⇒ L.B & H.V

BASIC RECIPE FOR ML {Bias - Variance Tradeoff}

① High Bias ? (training data performance) —Y→ ① Bigger n/w ② Train Longer. ③ Find other n/w search.

↓N

High variance ? (data set performance) —Y→ ① More data ② Regularization ③ NN architecture search

↓

DONE

Regularization → Prevents Overfitting

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

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

(Regularization Parameter; In practice can be omitted)

L_2 Reg $\|w\|^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ ← Used much more often

L_1 Reg $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w_x| = \frac{\lambda}{2m}\|w\|_1$ { w will be sparse ie w vector will have a lot of zeros which will compress the model }

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

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

↑ FROBENIUS NORM (L2 Regularization or Weight Decay)

Gradient Descent

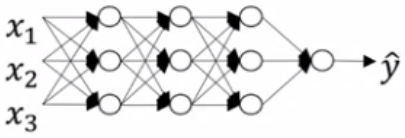
$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{2m}w^{[l]}$$

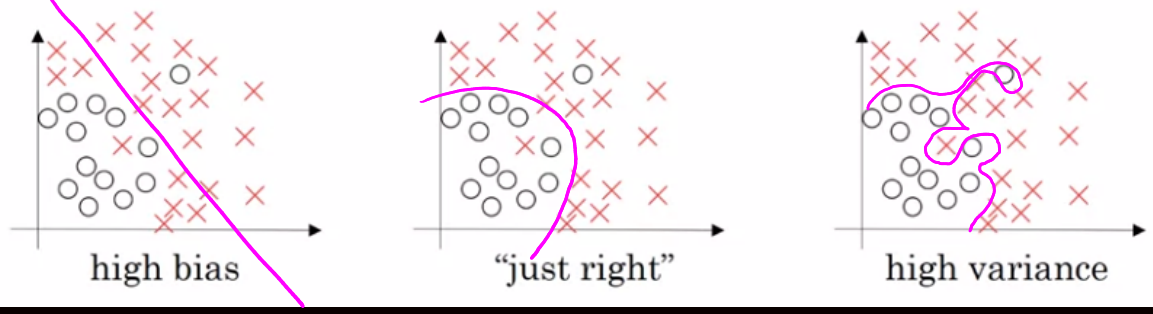
$$w^{[l]} = w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m}w^{[l]}\right]$$

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

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

How does regularization prevent overfitting?



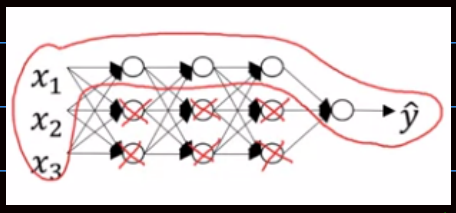


penalizes the wt. matrices from being too large

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

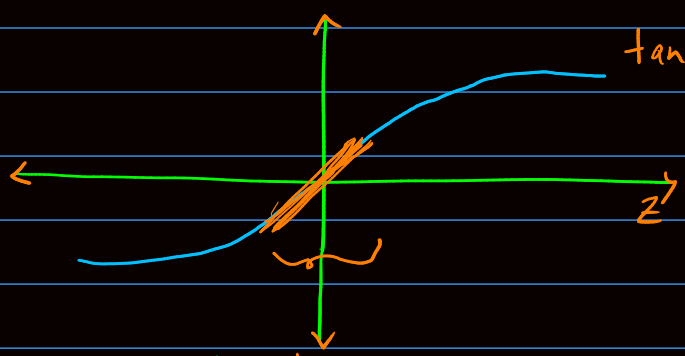
① If λ very large, $w^{[l]} \approx 0$

reducing the impact of wt. in hidden units



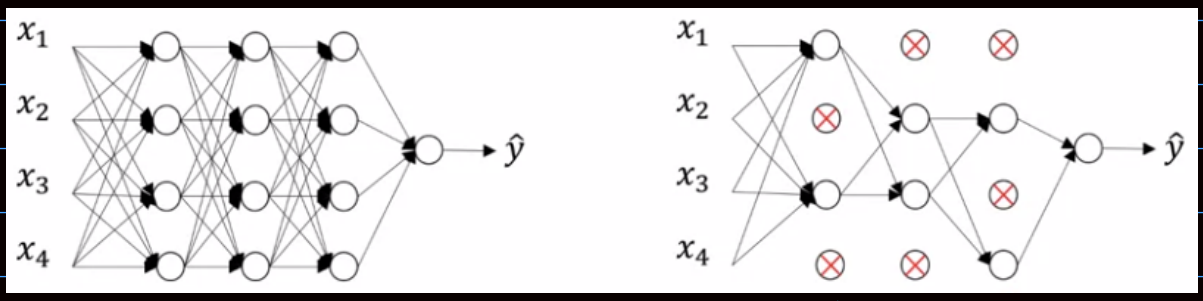
like a logistic Reg Unit

②



If $\lambda\uparrow$, $w^{[l]}\downarrow$, z is $\therefore$ the $g(z)$ will end up in the linear regime which is bad.

DROPOUT REGULARIZATION (* Mostly used in C.V Problems)



Implementation

At layer 3

keep_prob = 0.8

Generates Boolean Value → 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 }

$$Z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$$

↑ reduced by 20%.

Example:

In [46]:
```
keep_prob = 0.8
a3 = np.random.rand(50, 1)
print(a3[:5])
d3 = np.random.rand(a3.shape[0],a3.shape[1])<keep_prob
print(d3[:5])
a3 = np.multiply(a3,d3)
a3/= 0.8
print(a3[:5])
```

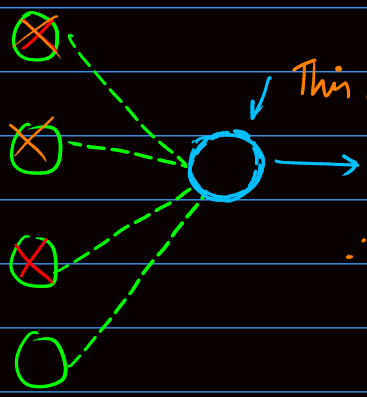
```
[[0.31604719]
 [0.94878238]
 [0.32860215]
 [0.18180209]
 [0.3065882 ]]
[[False]
 [ True]
 [ True]
 [ True]
 [ True]]
[[0.        ]
 [1.18597798]
 [0.41075269]
 [0.22725262]
 [0.38323525]]
```

Making Predictions at Test Time

* Don't use dropout {Add noise to our prediction & also we don't want anything random at test time}

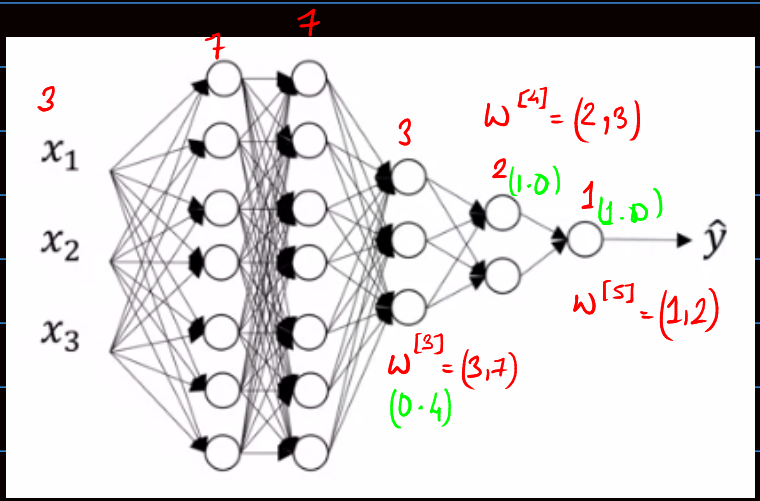
Why does Dropout Work?

Intuition: Can't rely on any one feature, so have to spread out weights → shrink wts. (L2)



This o/p can't rely on specific inputs only as on each iteration some units gets eliminated.
∴ Weights are spread out & small

* Vary keep_prob in each layer



* keep_prob values

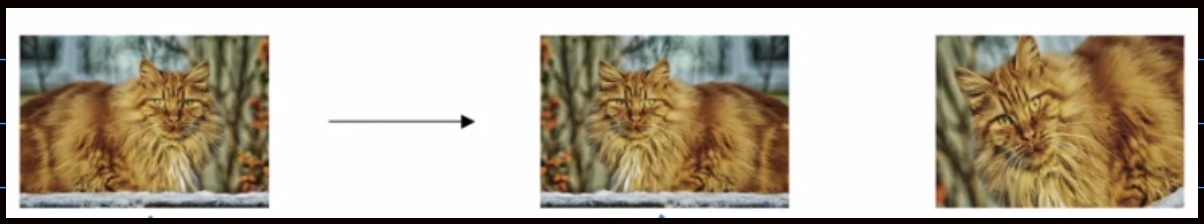
* Don't generally add dropout to the i/p layer {keep it close to 1}

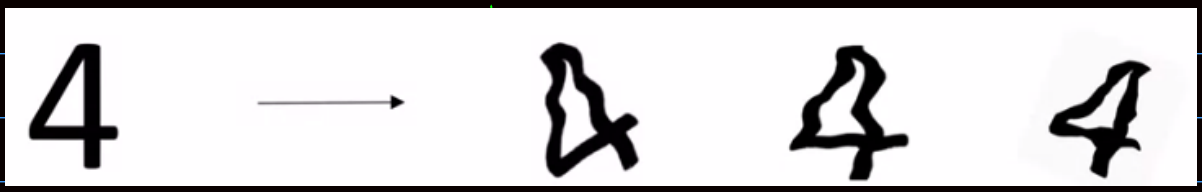
* Drawback → Not well defined J fun^n.

Other Regularization Techniques

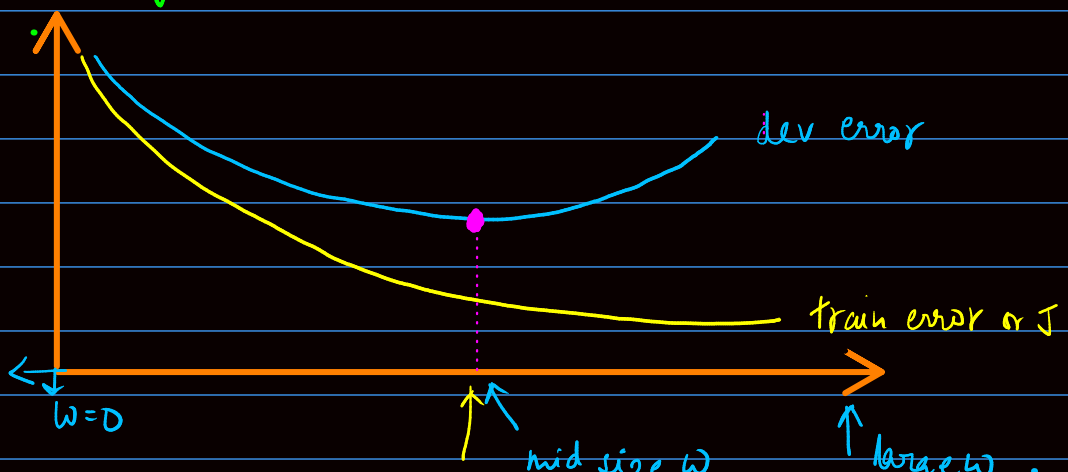
① Data Augmentation

Example:





② Early Stopping



NN does well till here acc. to dev error

$\|W\|_F^2$ = small norm

Any ML Problem (ORTHOGONALIZATION)

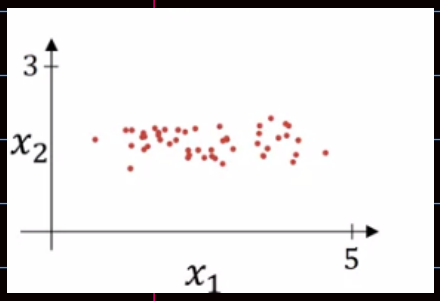
One task at a time

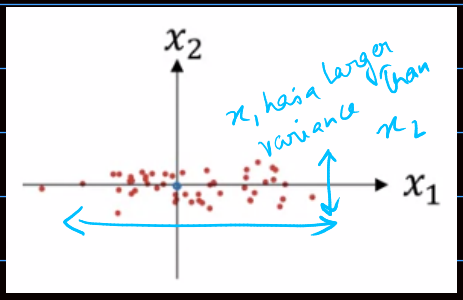
1. Optimize Cost fun^n J. { Gradient Descent... }] J(w,b)
2. Not Overfit { Regularization }] Seperate task

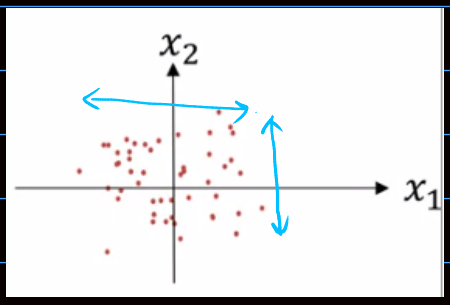
* But with Early Stopping I have to take care of both task simultaneously

Instead L2 Regularization is better though it requires much more expensive computation

Normalizing Training Sets







$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Subtract μ

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

$$X := X - \mu$$

Normalize Variance

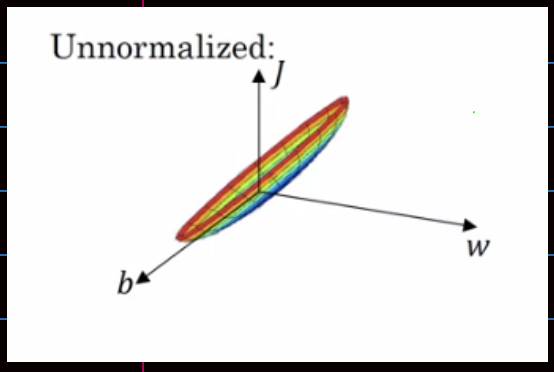
$$\sigma = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} **2$$

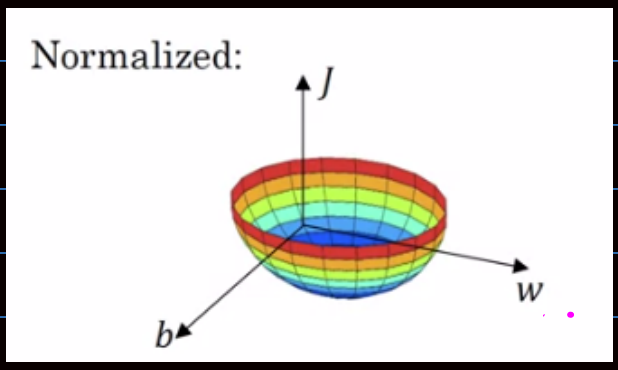
** element wise

$$X /= \sigma$$

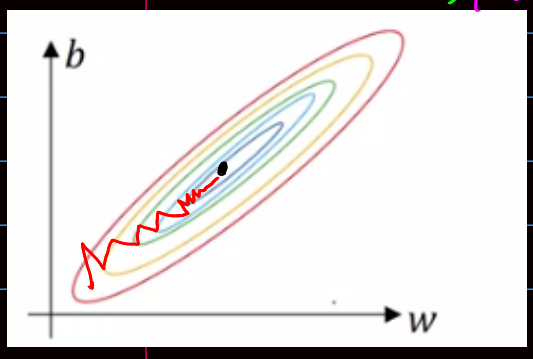
* Use the same μ & σ for normalizing test set.

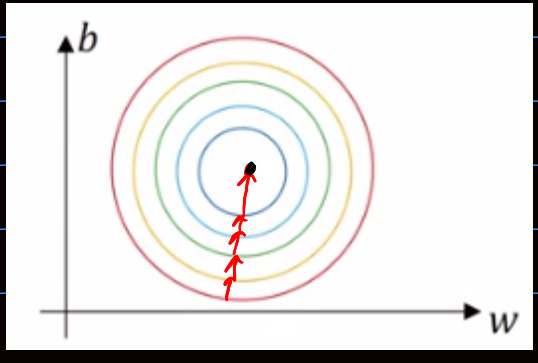
* $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$



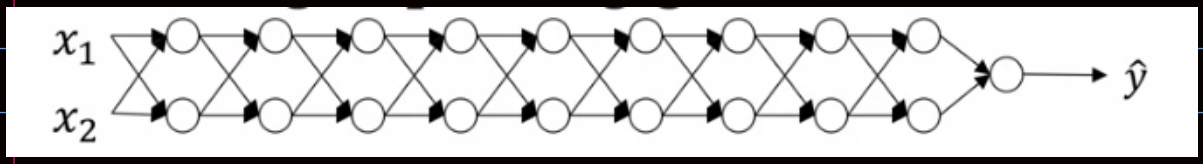


If i/p's are $x_1 = (1, \ldots 1000)$
$x_2 = (0, \ldots 1)$
} For similar ranges not much of an issue }





VANISHING & EXPLODING GRADIENTS



$W^{[1]}$ $W^{[2]}$ ——— $W^{[L]}$

lets used $g(x) = \text{identity}$, $b^{[l]} = 0$

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

$a^{[3]}$ $a^{[2]}$ $a^{[1]}$

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

$a^{[1]} = z^{[1]}$

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

$= 1.5^{(L-1)} x$ → very large.

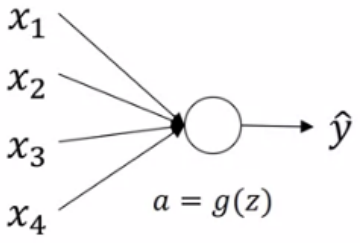
If $\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} = (0.5)^{L-1} x$ → very small

$W^{[l]} > I$ ⟶ exp. ↑ gradient descent & activation

$W^{[l]} < I$ ⟶ " ↓ " " " "

* Careful Weight Initialization can solve this problem upto some extent

Single neuron example



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

larger n ⟶ smaller w_i

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

∴ In code;

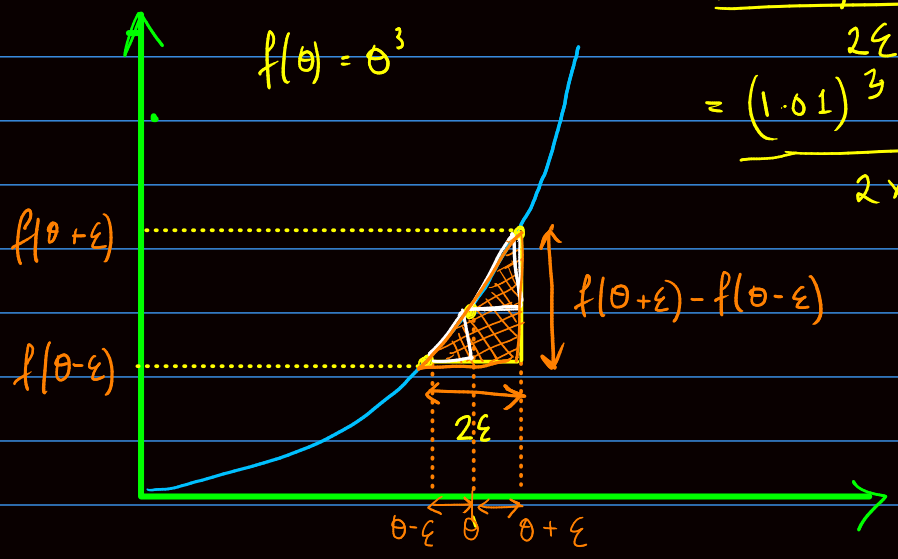
tanh. (Xavier's Initialization)

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

If ReLU is used as activation,

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

Numerical Approximation of gradients



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

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

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

$\varepsilon = 0.01$

approx error = 0.0001 { two sided diff way i.e considering two bigger Δ with both sides from θ

One sided error = 0.03

---

order of.

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$$ — $O(\varepsilon^2)$ — 0.01, 0.0001 — more accurate

$$\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$ — error: $O(\varepsilon)$ — 0.01 — less accurate

GRADIENT CHECKING {Debugging technique}

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

concat

Step 2: Take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

Concat

$$J(\theta) = J(\theta_1, \theta_2, \theta_3 \ldots)$$

for each i:

$$d\theta_{approx}^{[i]} = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \varepsilon, \ldots) - J(\theta_1, \theta_2, \theta_3 \ldots \theta_i - \varepsilon)}{2\varepsilon}$$

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

↓

Check $d\theta_{approx} \overset{?}{\approx} d\theta$

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

=

- 10^{-7} → great
- 10^{-5} → Not bad but have a look again
- 10^{-3} → must be some error somewhere

* Don't use in training — only to debug {very slow operation}

* If algorithm fails grad check, look at the components to identify bug

* Remember to include the regularization term if used.

* Doesn't work with dropout. {Turn off dropout when using grad check}

* Run grad check at random initialization perhaps again after some training {As sometime backprop works better when $W, b \approx 0$}

WEEK 2

Batch vs mini-batch gradient descent

$$X = \left[x^{(1)} \; x^{(2)} \; \ldots\ldots\ldots \; x^{10^6} \right]_{(n_x \times m)}$$

$$X = \left[y^{(1)} \; y^{(2)} \; \ldots\ldots\ldots \; y^{10^6} \right]$$

In such big data we split the data into mini batches of 1000 (say)

$$X = \left[\underbrace{X^{(1)} \; X^{(2)} \; \ldots\ldots \; X^{1000}}_{X^{\{1\}}_{(n_x, 1000)}} \;\middle|\; \underbrace{X^{1001} \; \ldots\ldots \; X^{2000}}_{X^{\{2\}}} \;\middle|\; \ldots\ldots \;\middle|\; \underbrace{\ldots\ldots \; X^{1m}}_{X^{\{5000\}}} \right]$$

$$Y = \left[\underbrace{y^{(1)} \; y^{(2)} \; \ldots\ldots \; y^{(1000)}}_{Y^{\{1\}}_{(1, 1000)}} \;\middle|\; \ldots\ldots\ldots \;\middle|\; \underbrace{\ldots\ldots \; y^{1m}}_{Y^{\{5000\}}} \right]$$

for $t = 1 \ldots$ range :

{

One step of gradient descent

[Forward Prop on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b$$

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

$$\vdots$$

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

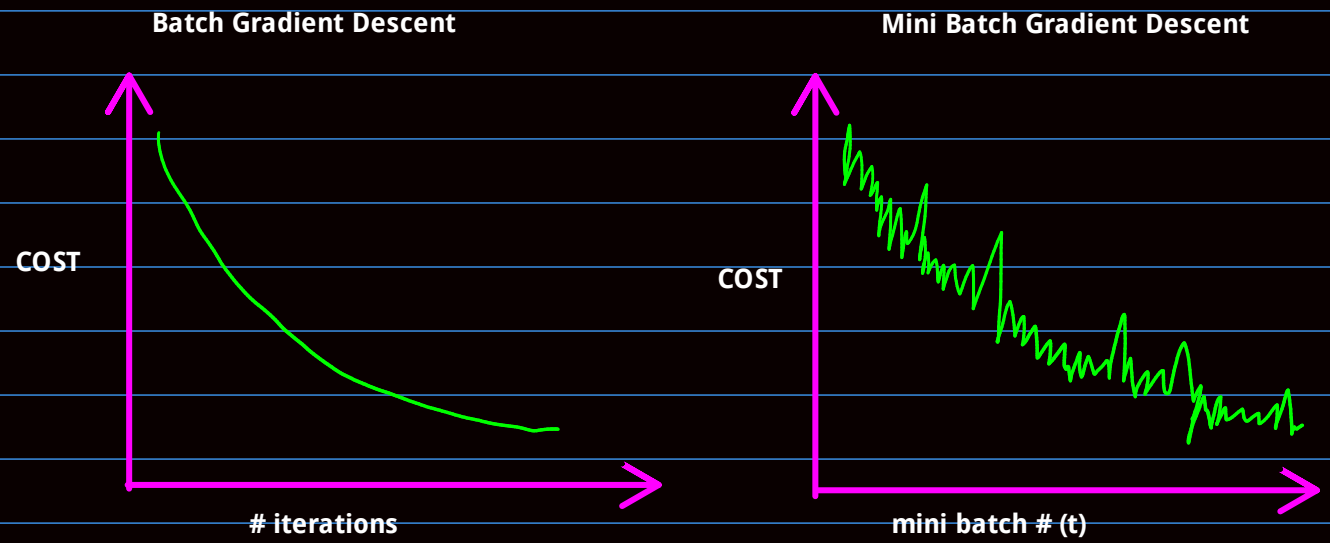
} Over 1000 examples

Compute cost $= \frac{1}{1000} \sum_{i=1}^{1000} L(\hat{y}^{(i)}, y^{(i)}) := J^{\{t\}}$

Back Prop w.r.t $J^{\{t\}}$

$$W^{\{t\}} = W^{\{t\}} - \alpha \, dW^{\{t\}}, \quad b^{\{t\}} = b^{\{t\}} - \alpha \, db^{\{t\}}$$

}]

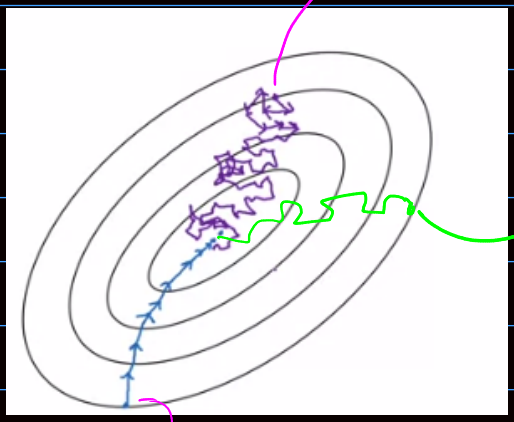


Plot $J^{\{t\}}$ captured using $X^{\{t\}}, Y^{\{t\}}$

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

(M.B)

- If M.B size = 1 : Stochastic Gradient Descent
  Each batch is one training example.



Stochastic Gradient Descent
↳ Drawback : Loss all speeding up due to vectorization

Mini
Best: Batch Gradient Descent
(Batch size: not too big or small)
- Fastest Learning
- Make Progress & no waiting for the entire dataset

Batch Gradient Descent (Batch size = m)
Drawback : Too long per iteration

* If small data set : Batch Gradient Descent
  ($m \leq 2000$)

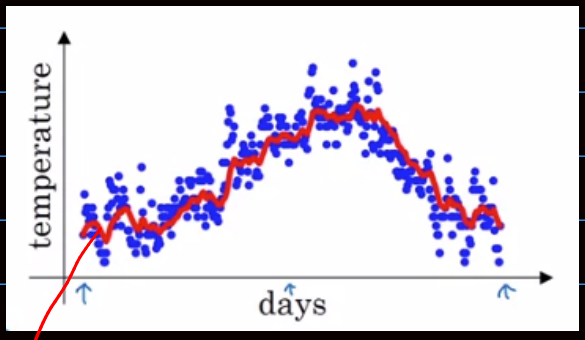
* Typical Mini Batch Size: 64, 128, 256, 512

* Make sure minibatch size in CPU/GPU Memory

EXPONENTIAL WEIGHTING AVERAGES

Temperature Plot

$\theta_1 = 40^\circ F$
$\theta_2 = 40^\circ F$
$\theta_3 = 45^\circ F$
:
:
$\theta_{180} = 60^\circ F$
$\theta_{181} = 56^\circ F$



Moving average Plot

Let $V_0 = 0$

$V_1 = 0.9 V_0 + 0.1\theta_1$

$V_2 = 0.9 V_1 + 0.1\theta_2$

$V_3 = 0.9 V_2 + 0.1\theta_3$

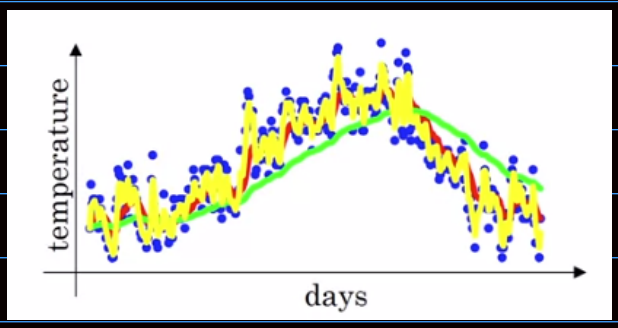
$$V_t = 0.9 V_{t-1} + 0.1\theta_t$$

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

When $\beta = 0.9$ ~~~

$\beta = 0.98$ ~~~

$\beta = 0.5$ ~~~



Example

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

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

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

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

...

$V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9 v_{97}))$

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

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

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

$0.98^{?}$

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

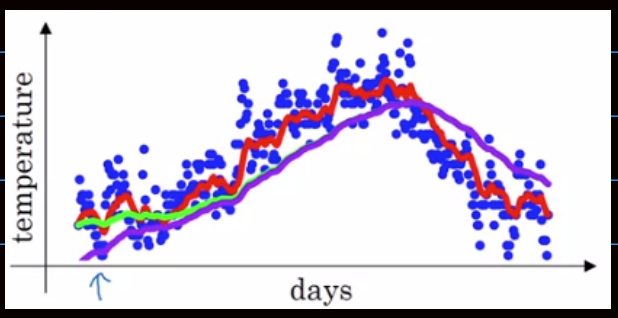
$\approx \frac{1}{1-\beta}$

$\varepsilon = 1-\beta$

Andrew Ng

Bias Correction

When $\beta = 0.98$ we should get the green line but we get the purple line which starts off really low.



$V_0 = 0$

$V_1 = 0.98 V_0 + 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2 = 0.09(0.98 V_0 + 0.02\theta_1) + 0.02\theta_2$

$= \underline{0.0196\theta_1} + \underline{0.02\theta_2}$

much small

Rather,

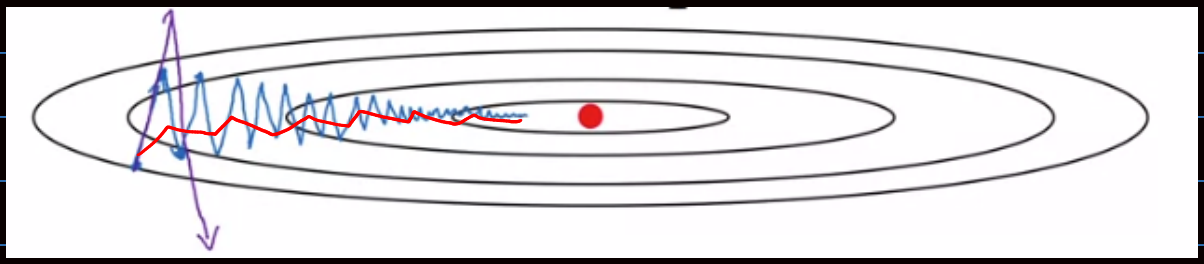
Correction $\frac{V_t}{1-\beta^t}$

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

$\frac{V_2}{0.039} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

When t is large $1-\beta^t \approx 1$, so the purple & green line becomes one

GRADIENT DESCENT WITH MOMENTUM



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

On iteration t;

Compute dw, db on current mini batch

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

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

Slow learning ↕ faster learning ↔

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

* In practise bias correction is not really necessary

Another analogy,

$V_{dw} = \beta V_{dw} + (1-\beta)\, dw$ → provides acceleration to the ball.

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

friction — provides velocity

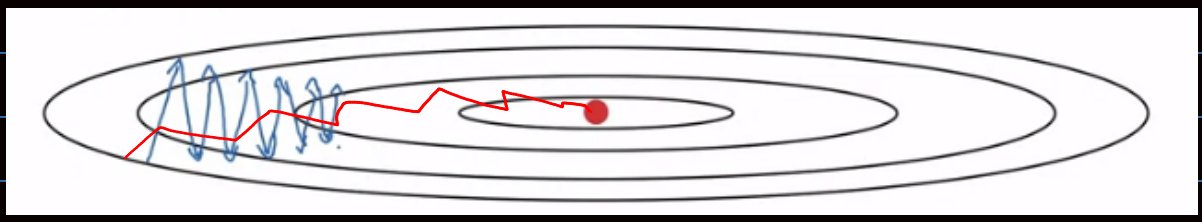
So here we have 2 hyperparameter α, β

Common $\beta = 0.9$ ⟶ Iterating over the last 10 gradients

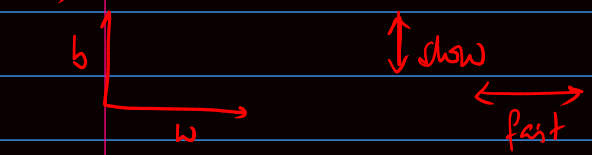
* Sometimes in practice $(1-\beta)$ term is omitted from the equation.

$V_{dw} = \beta V_{dw} + dw$ → But not much preffered

Root Mean Square Prop. (RMP Prop)



let,

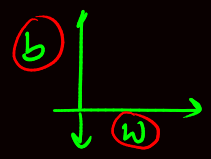


On iteration t:

Compute dw, db on current mini batch

$S_{dw} = \beta S_{dw} + (1-\beta)\, dw^2$ ← element wise
↳ small

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



✓ correct intuition

$$w = w - \frac{\alpha\, dw}{\sqrt{S_{dw}}} \qquad b = b - \frac{\alpha\, db}{\sqrt{S_{db}}}$$

S_{dw} small → $\frac{dw}{\sqrt{S_{dw}}}$ large; b large → $\frac{db}{\sqrt{S_{db}}}$ small

* $\sqrt{S_{dw} + \varepsilon}$ or $\sqrt{S_{db} + \varepsilon}$ is used where $\varepsilon = 10^{-8}$

\+ Adaptive Momentum

ADAM OPTIMIZATION (Momentum & RMS Prop Combination)

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

On iteration t:

Compute dw, db using current mini batch

Momentum → $V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw^2$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

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

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

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

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

Hyperparameters:

α = needs to be tune

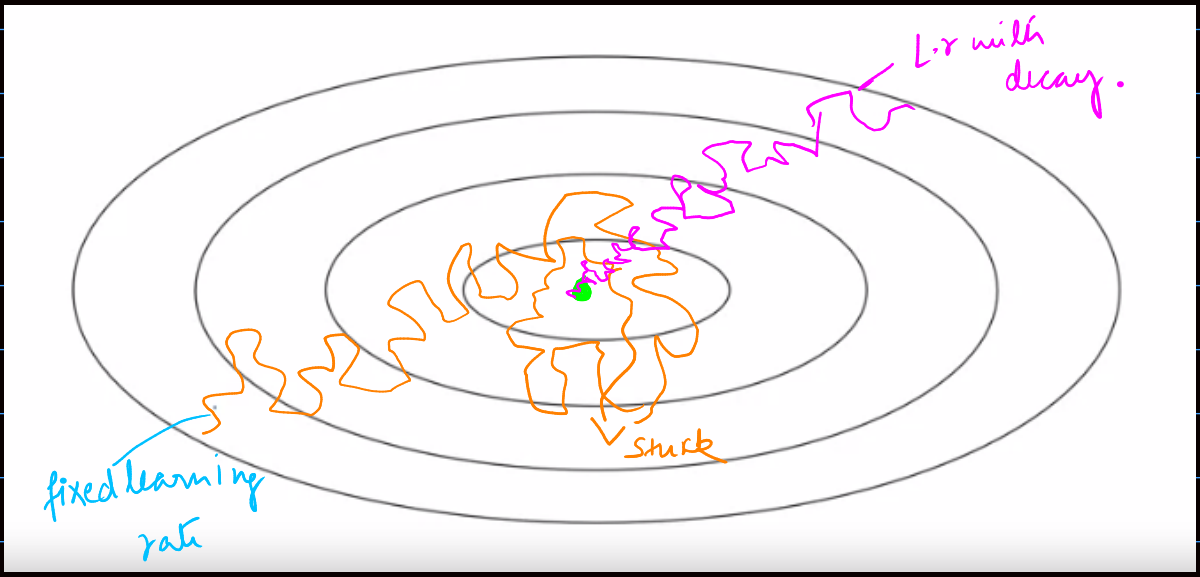
$\beta_1 = 0.9$ (dw)

$\beta_2 = 0.999$ (dw^2)

$\varepsilon = 10^{-8}$

} Usually not tweaked.

Learning Rate Decay



1 epoch → 1 pass through the entire data

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

hyperparameter

Let $\alpha_0 = 0.2$, decay-rate = 1.

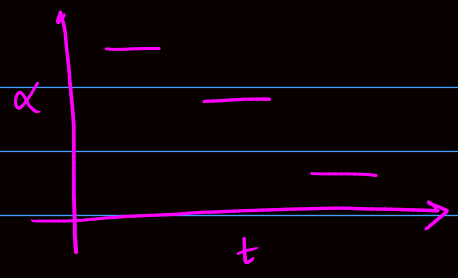
Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04

Other l.r. decay

Exponential Decay : $\alpha = 0.95^{\text{epochnum}} \alpha_0$

or $\alpha = \frac{k}{\sqrt{\text{epochnum}}} \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$

or

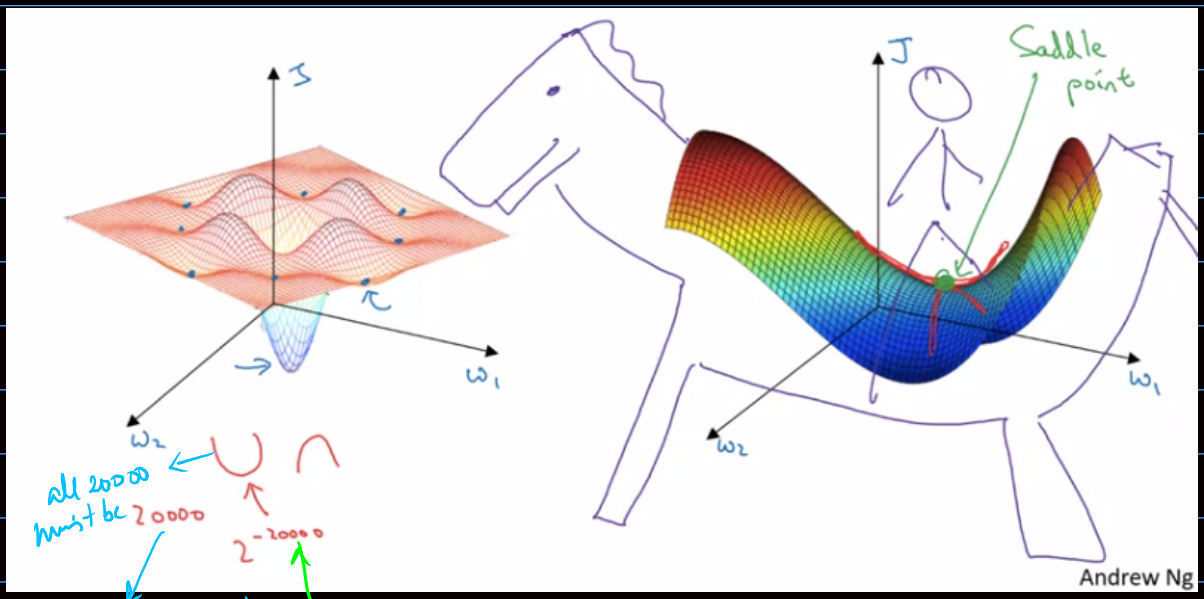


discrete stepwise decay

or

* Manual Decay

LOCAL OPTIMA IN NEURAL NETWORKS



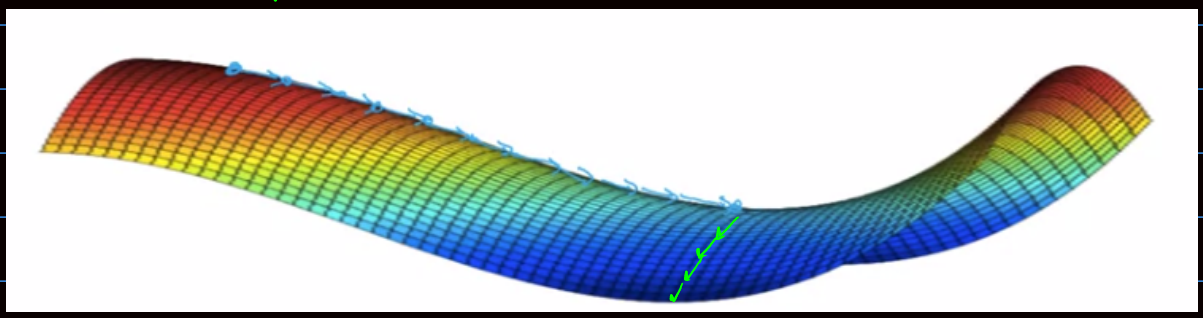
all 20000 must be zero

2^{-20000}

dimensions

This is very unlikely to find.

Problem of Plateau



- Unlikely to get stuck in a bad optima.
- Plateaus can make learning very slow (Use Adam, RMS Prop)

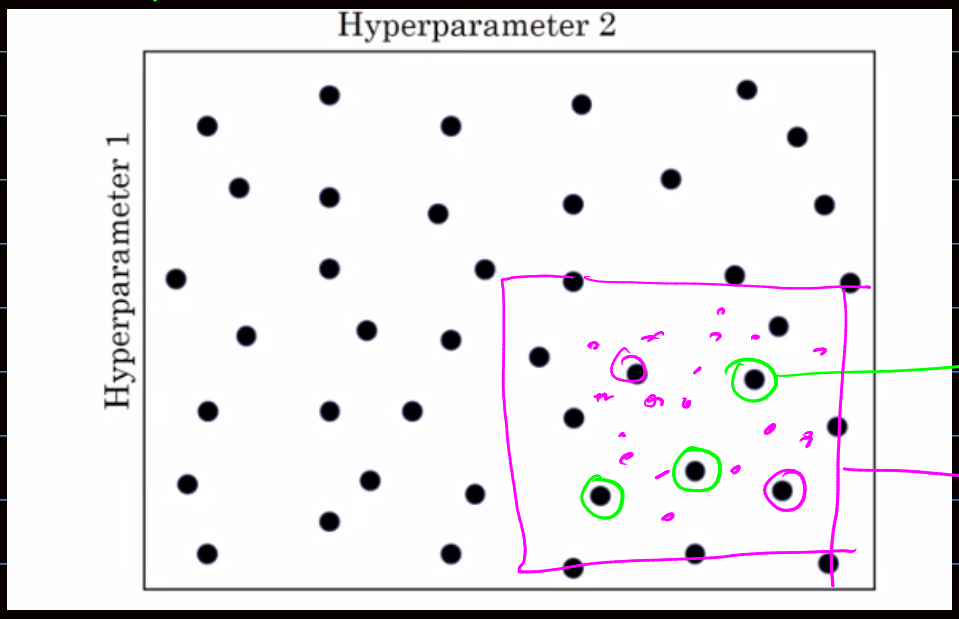
WEEK 3

Tuning Process

In order. {Tip}

$\alpha \longrightarrow \beta$ | hidden units | mini batch size $\longrightarrow$ #layers | learning rate decay $\longrightarrow \beta_1, \beta_2, \varepsilon$

* Try Random Values : Don't follow a pattern strictly
* Course to fine



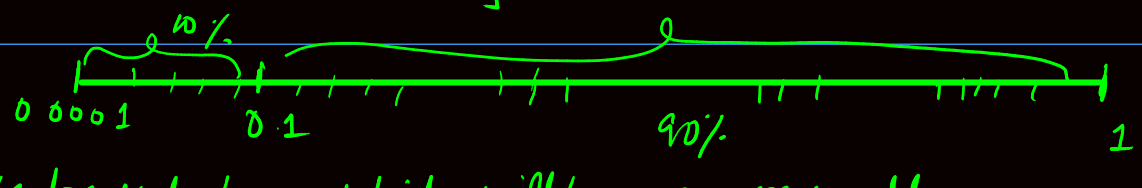
Using an appropriate scale to pick hyperparameters

For eg

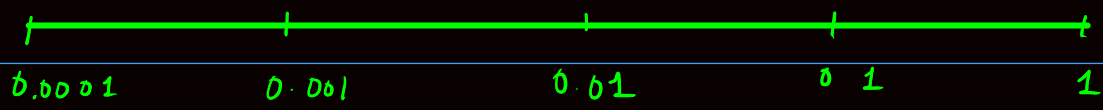
① $n^{[l]} = [50, 100]$

② #layers $= [2, 6]$

③ $\alpha = [0.0001, 1]$



∴ Use log scale here, which will be more reasonable

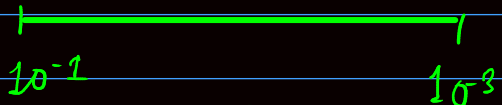


④ $\beta = 0.9 \ldots\ldots 0.999$

↓ last

10 Avg over 1000 values

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



Dont use linear scale

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

$1-\beta = 10^r$

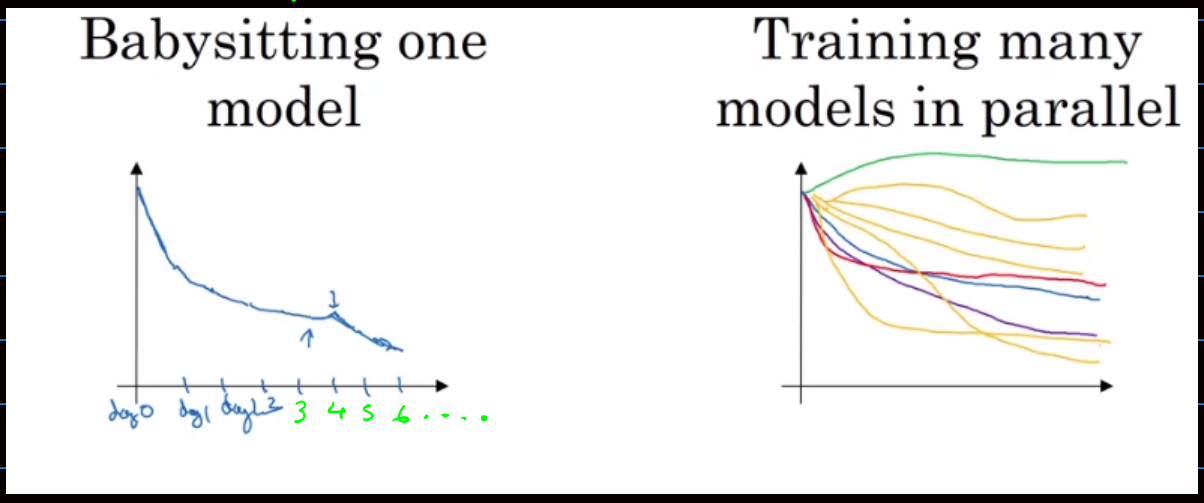
$\beta = 1 - 10^r$

* To organize hyperparameters

Intuitions

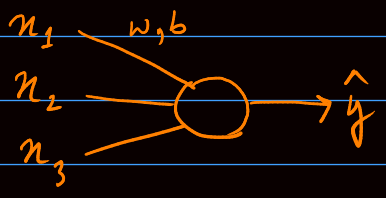
* Hyperparameter do get stale overtime so re-evaluate them occasionally

Approach



Choose the approach according to the computational expenses

BATCH NORMALIZATION

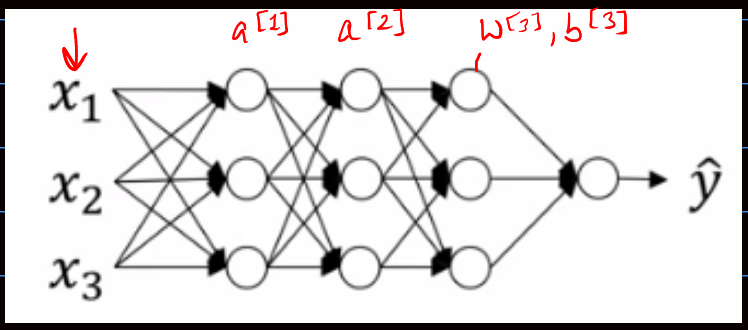


$$\mu = \frac{1}{m}\sum_i x^{(i)}$$

$$X = X - \mu$$

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

$$X = X/\sigma^2$$



In bigger model we normalize after every step

* Normalizing $z^{[2]}$ or $a^{[2]}$ to train $w^{[3]}$ & $b^{[3]}$ faster is a debate but $z^{[2]}$ is used more often

* Given some intermediate values in NN $z^{(1)}, z^{(2)} \ldots z^{(m)}$

$z^{[l](i)}$

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

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

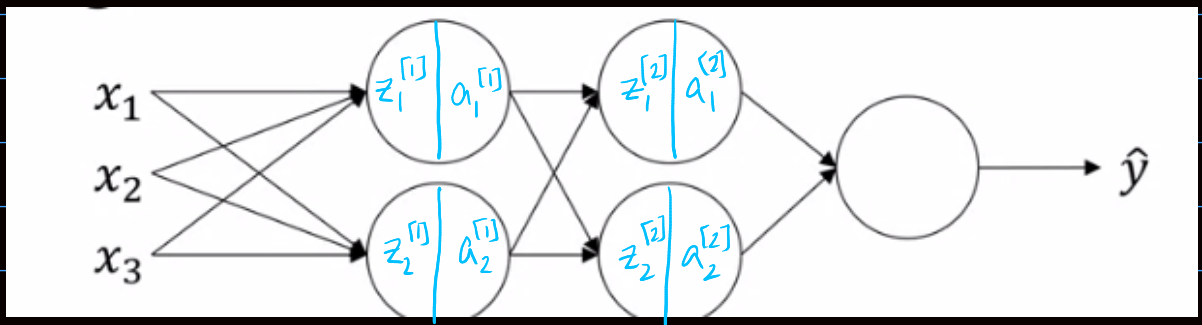
$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \approx \text{Here mean} \approx 0 \text{ \& Var} = 1$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$ { If we want diff mean & var }

(γ, β: learnable params of the model.)

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

So, in batchnorm use $\tilde{Z}^{[l](i)}$ instead of $Z^{[l](i)}$



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

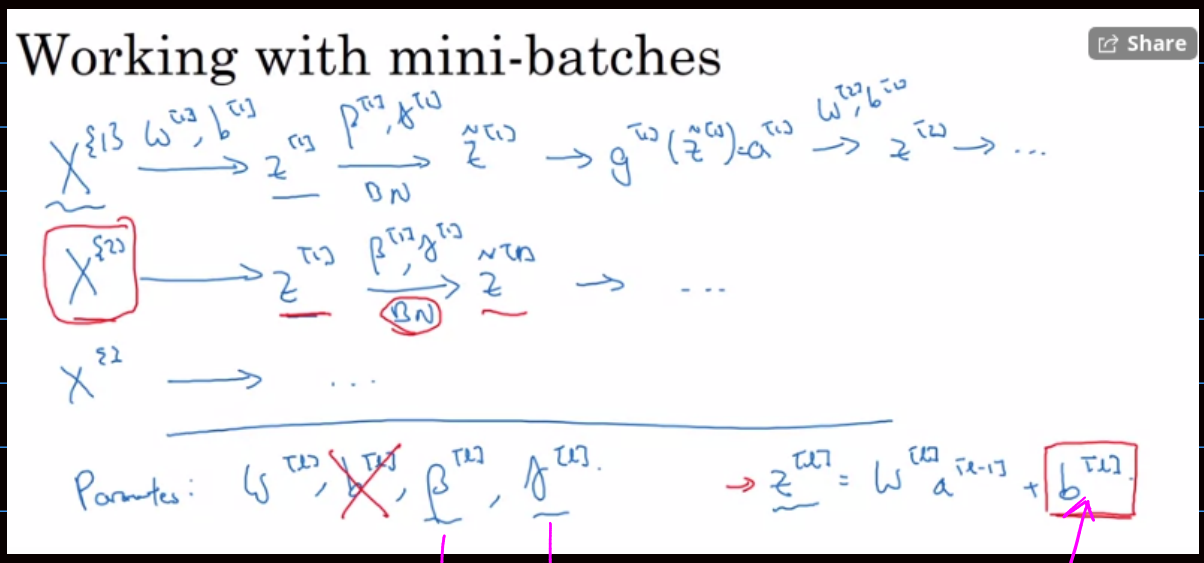
$W^{[2]}, b^{[2]}$

$\ldots\ldots a^{[2]} \leftarrow \tilde{Z}^{[2]} \xleftarrow[\beta^{[2]}, \gamma^{[2]}]{B.N} Z^{[2]}$

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \ldots\ldots$

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

B-N Paper → X Momentum β



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

This gets cancelled in the mean subtraction step

$\therefore$ $b^{[l]}$ is not required if BN is used.

Implementing gradient descent.

for $t = 1 \ldots$ num of mini batches :

{ Compute Forward Prop on $X^{\{t\}}$

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

Use back Prop to compute.

$dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

Update

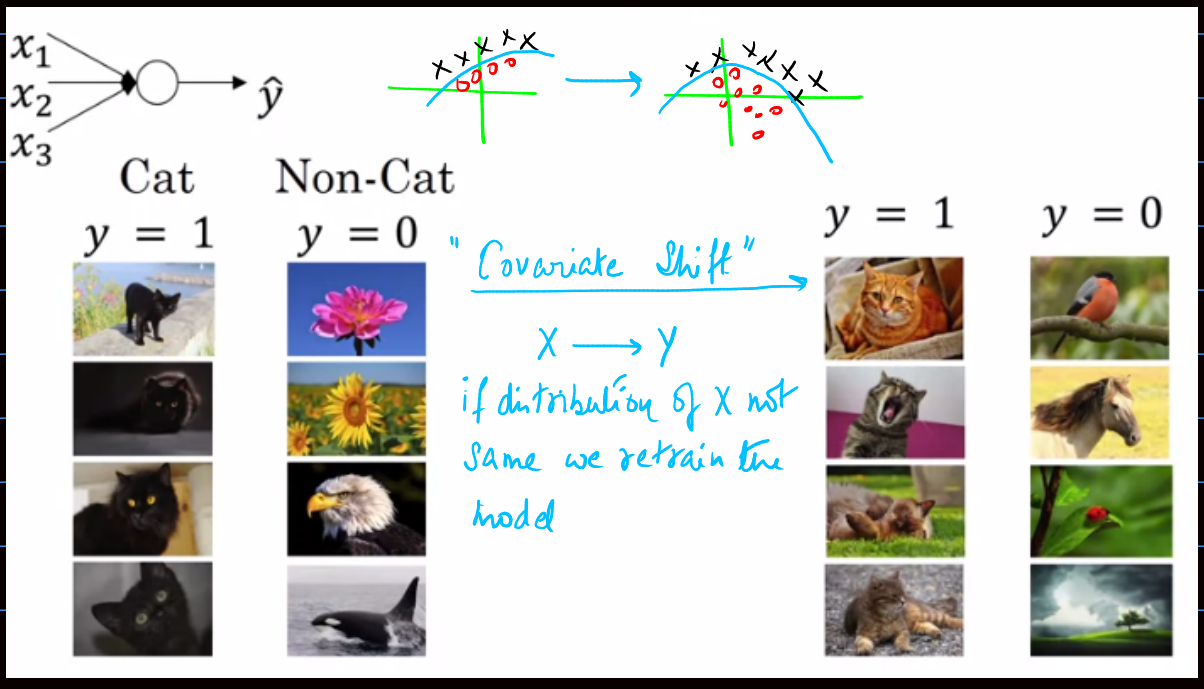
$$W^{[l]} = W^{[l]} - \alpha\, dw^{[l]}$$
$$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$$
$$\gamma^{[l]} = \gamma^{[l]} - \alpha\, d\gamma^{[l]}$$

Works with Momentum / Adam / RMS Prop

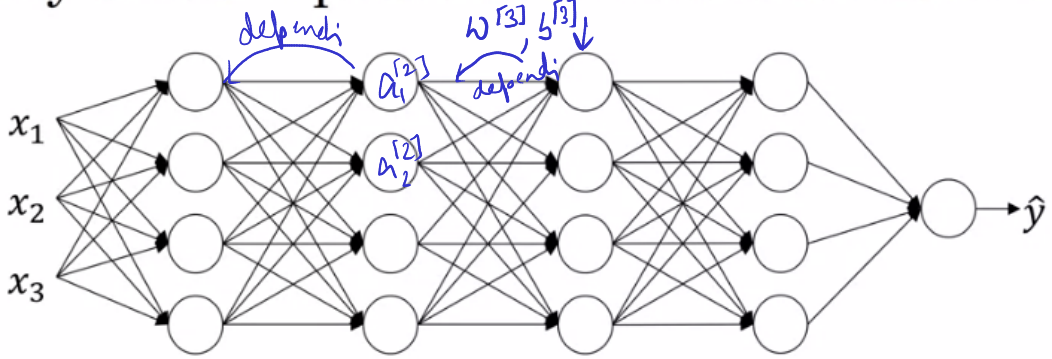
}

Why Batch Norm works?

Learning on shifting input distribution

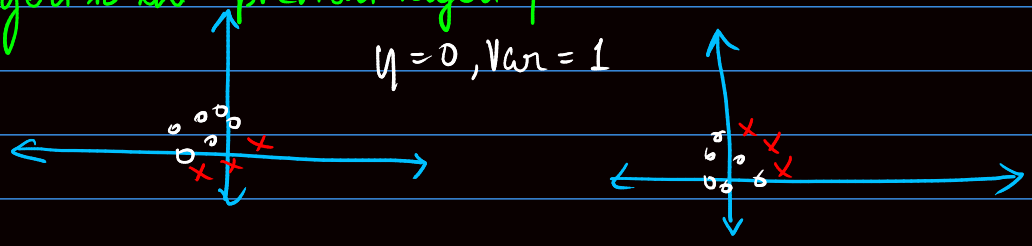


Why this is a problem with neural networks?



Each layer deep into the network is directly dependent on the parameter changes in the previous layer

* So, at least to make the distribution uniform with same mean & variance B.N is used, & also amount of adaptability of the later layers to the previous layers ↑



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. (Both multiplicative & additive noise)
- This has a slight regularization effect.

Use bigger mini batch size we reduce the regularization effect

Batch Norm at test time:

Regular B.N on Train set →

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

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

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

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

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

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

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

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

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

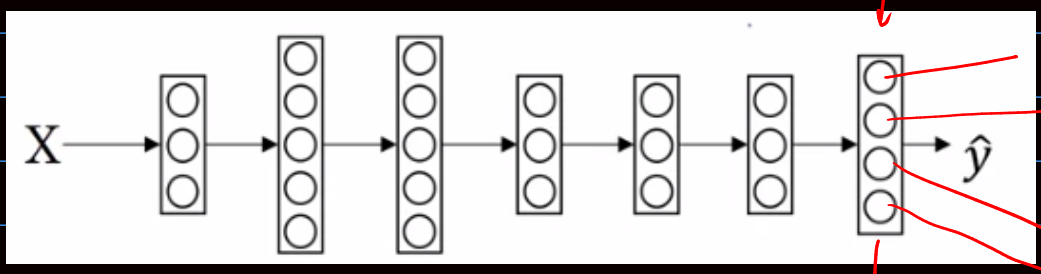
* ML Frameworks will do this automatically.

SOFTMAX REGRESSION

Recognizing cats, dogs, and baby chicks

3 1 2 0 3 2 0 1

C = #classes = 4 (0 ... 3)



$\hat{y} = (4,1)$

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

Activation:

$$t = e^{z^{[l]}} \quad \{ \text{element wise exponentiation}$$

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{i=1}^{4} t_i} \quad (4,1) \qquad a_i^{[l]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

Eg

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

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

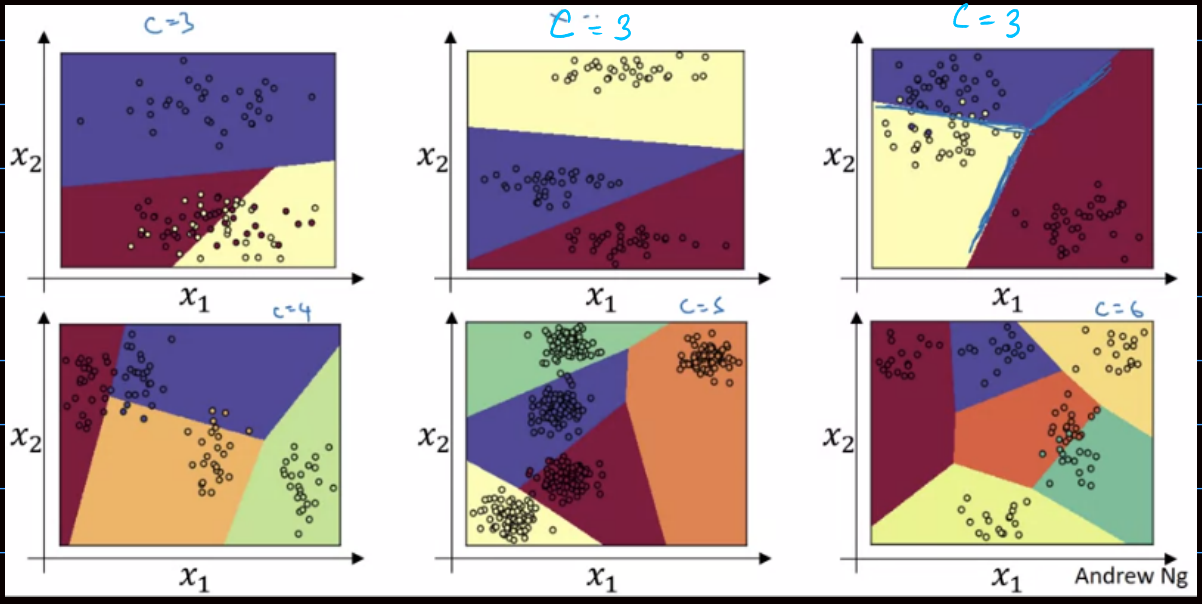
$$a^{[1]} = \frac{148.4}{176.3} = 0.842 \qquad a^{[2]} = \frac{7.4}{176.3} = 0.042$$

$$a^{[3]} = \frac{0.4}{176.3} = 0.002 \qquad a^{[4]} = \frac{20.1}{176.3} = 0.114$$

Softmax Activation:

$$a^{[l]} = \frac{t}{\sum_{i=1}^{4} t_i}$$

$$a^{[l]} = g^{[l]}\left(z^{[l]}\right)$$



Training a softmax classifier.

(4,1)

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

Softmax

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

"hard" max

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Softmax regression generalizes logistic regression to C classes

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

Loss fun^n

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$ Cat

but, $a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$

$y_1 = y_3 = y_4 = 0 \; ; \; y_2 = 1$

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

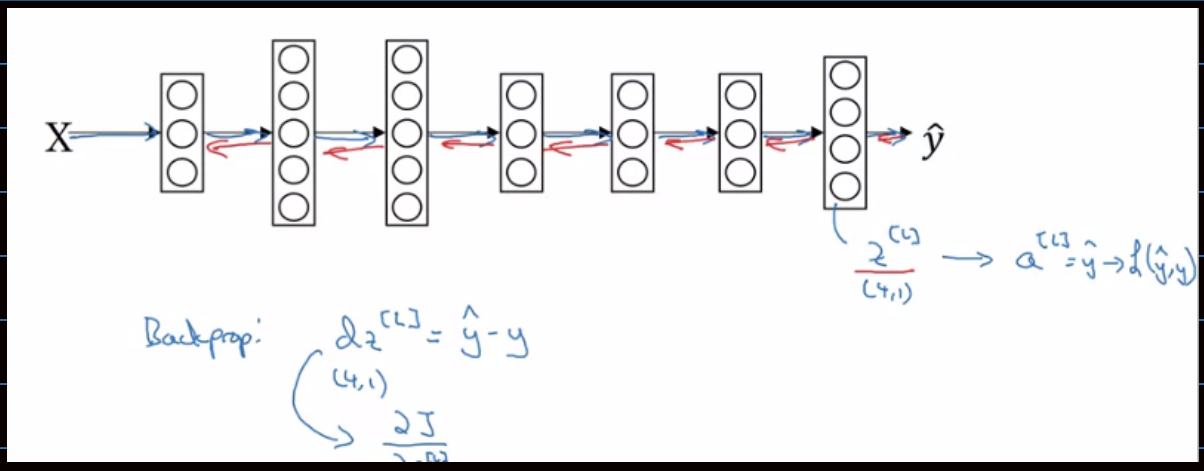
$$= -y_2 \log \hat{y}_2 = -1 \log \hat{y}_2 \quad \therefore \hat{y}_2 \uparrow$$

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

$$Y = [y_1^{(1)} \; y_2^{(2)} \; y^{(3)}, \ldots]$$

$$= \begin{bmatrix} 0 & 0 & 1 & \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \cdots \\ 0 & 0 & 0 & \end{bmatrix}$$

Gradient Descent



Deep learning frameworks

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Choosing deep learning frameworks

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