Optimize NN, ※2

Week 1

Train / dev / test set

Data | 60 % | 20 % | 20 %

Train set - Dev set - Cross validation set | Test set

make sure your dev and test set come from same distribution

Bias / Variance

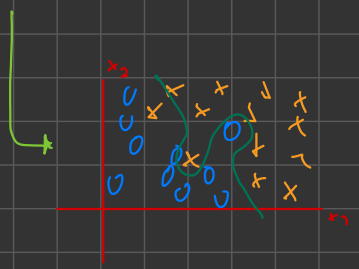
high bias | just right | high variance

Underfit | | Overfit

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

← We want this

high variance | low variance



Regularization

Logistic regression

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

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

λ : regularization parameter

L_2 regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

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

Neural Network

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

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

"Frobeius norm"

$dw = (\text{from back prop}) + \frac{\lambda}{m} w^{[l]}$ $\qquad \frac{d\mathcal{L}}{dw^{[l]}} = dw^{[l]}$

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

"Weight decay" $w^{[l]} := w^{[l]} - \alpha[(\text{from back prop}) + \frac{\lambda}{m} w^{[l]}]$

$:= w^{[l]} - \alpha\frac{\lambda}{m} w^{[l]} - \alpha(\text{back prop})$

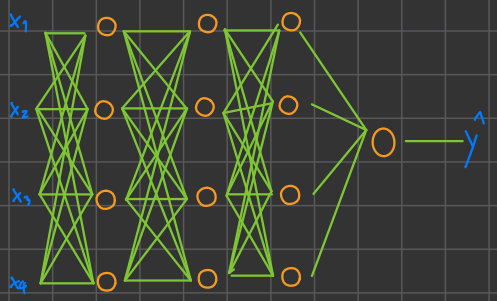
$(1 - \frac{\alpha\lambda}{m})$ ← Why it's call weight decay because this term

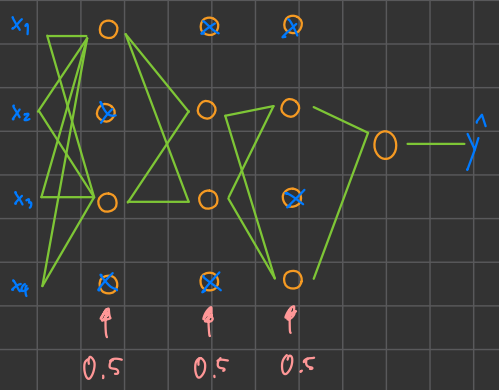
Why regularization prevent overfit

- when train move $w \approx 0$ and our model became high bias and λ will help to just right
- to help fuction became linear

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

Dropout regularization





How to implementing dropout

- Inverted dropout

Illustrating with layer $l = 3$ keep.prob 0.8

$d3 =$ np.random.rand(a3.shape[0], a3.shape[1]) < keep.prob

$a3 =$ np.multiply(a3, d3) ※ $a3 *= d3$

$a3 /=$ keep.prob → to not change expect value of next layer ex 50 units → 10 unit shut off

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

↳ reduced by 20% / = 0.8

Why does it work

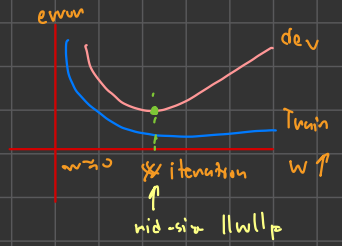
- Can't rely on any one feture, so have to spread out weights.
  - ↳ Shrink weight

- Recommend in Computer Vision → because image have many pixcle mean many feature so dropout

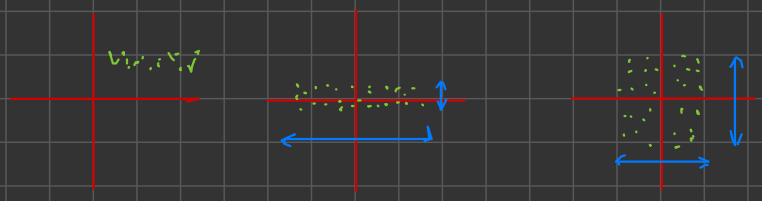
Other regularization methods

- Data augments
  - rotate
  - Filp

- Early stopping
  - plot test and Dev set error when find minimum error dev set stop train



Normalizing



subtract mean:

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

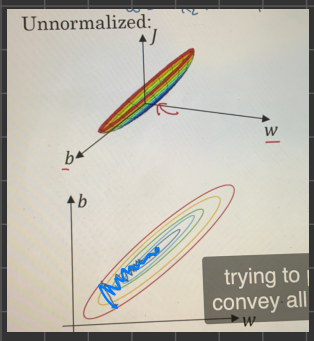
$$x = x - \mu$$

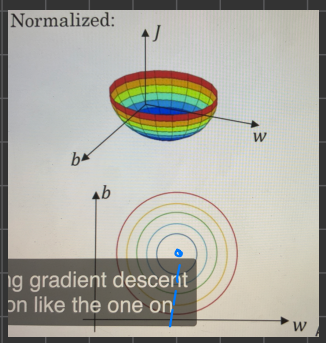
Norm variance

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

$$x /= \sigma^2$$

Why we normalize



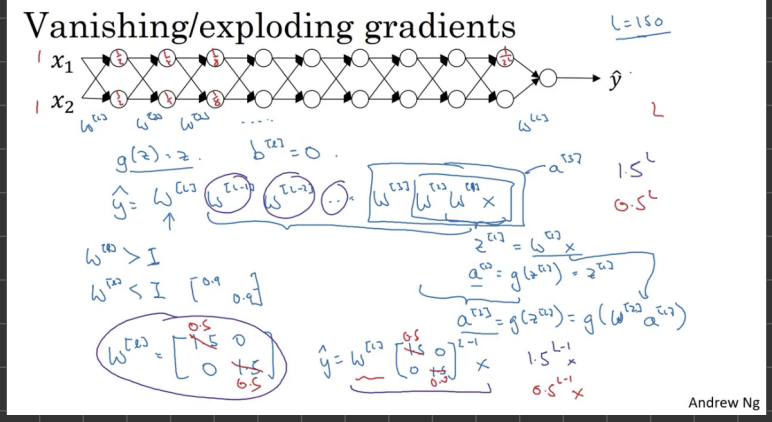


- The gradient decent can setep easier (not oscillate)
- Prevent one Feature have more pionity than others (similar scale)
- easy to optimize (1 → 1000 to 0 → 1, -1 → 1

Vanishing / Exploding gradients

13:32 Fri 22 May

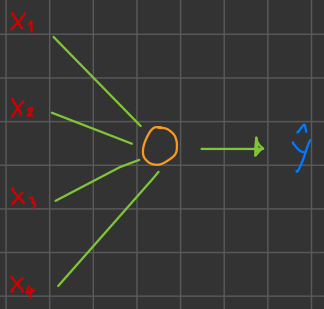
Vanishing / Exploding gradients



When weight became Vanishing or exploding gradient decent need to step small / big that cause our gradient decent learn longer (small step → learn slow big step → learn harder because it osillate)

Weight Initialization for Deep NN

Single neuron example



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

larger n → smaller w_i

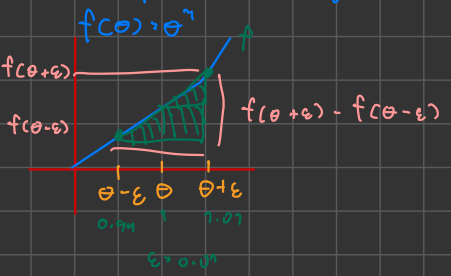
$Var(w_i) = \frac{1}{n}$ → "Relu" $\frac{2}{n}$

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

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

Xavier initialization

Numerical approximation of gradients



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

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

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

approx error : 0.0001

Gradient checking

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

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

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

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

for each i:

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

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i} \quad d\theta_{approx} \overset{?}{\approx} d\theta$$

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx 10^{-9}$ → great!

$\varepsilon = 10^{-9}$ 10^{-7} — Buy!

Tip

- Don't use in training - only to debug
- If algorithm fails grad check, look at components to try to identify bug

$db^{[l]}$ $dw^{[l]}$

- Remember regularization
- Doesn't work with drop out
- Run at random initialization; perhaps again after some training → maybe w and b ≈ 0 so do it again maybe help :)

Week 2

Mini-batch

Vectorization allows you to efficiently compute on m examples

$X = [x^{(1)}, x^{(2)}, \dots, x^{(m)}]$

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

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

mini-batch of 1,000 each

$X = [\underbrace{x^{(1)}, x^{(2)}, \dots, x^{(1000)}}_{X^{\{1\}}\ (n_x, 1000)} \mid \underbrace{x^{(1001)}, \dots, x^{(2000)}}_{X^{\{2\}}\ (n_x, 1000)} \mid \dots \mid \underbrace{\dots, x^{(5{,}000{,}000)}}_{X^{\{5000\}}\ (n_x, 1000)}]$ (n_x, m)

$Y = [\underbrace{y^{(1)}, y^{(2)}, y^{(3)}, \dots, y^{(1000)}}_{Y^{\{1\}}\ (1, 1000)} \mid \underbrace{y^{(1001)}, \dots, y^{(2000)}}_{Y^{\{2\}}\ (1, 1000)} \mid \dots \mid \underbrace{\dots, y^{(5{,}000{,}000)}}_{Y^{\{5000\}}\ (1, 1000)}]$ $(1, m)$

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

$x^{(i)}, z^{[l]}, X^{\{t\}}, Y^{\{t\}}$

Mini-batch gradient decent

for $t = 1 : 5000$ each

Forward prop. on $X^{\{t\}}$ ← step of gradient decent using $X^{\{t\}}, Y^{\{t\}}$ (as if $m = 5000$)

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

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

⋮

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

Vectorization Implementation (1000 examples)

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

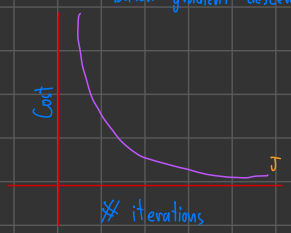
Back prop to compute gradient decent $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

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

it's call

"1 epoch" → pass through training set

Batch gradient descent



Mini-batch gradient descent

because we speartrate the training example so that cause noise on training

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

Choosing mini-batch size

if mini batch size = m : gradient decent. $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

if mini batch size = 1 : Stochastic gradient decent. Every example has it own mini-batch ● (noise and may be can wrong direction)

$(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$

In pratice : Some where in between 1 and m

Batch size = m

↳ Too long per iteration

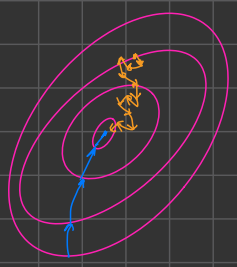
In-between (not too big or small)

- Fastest learning (vectorization)
- Make progress without needing to wait till your process entries training set

Batch size = small size

↳ lose speed in form vectorization

(can solve by reduces learning rate



Suggest

If your have small training set : Use batch gradient decent

(m ≤ 2000)

Typical mini-batch size :

64, 128, 256, 512, 1024

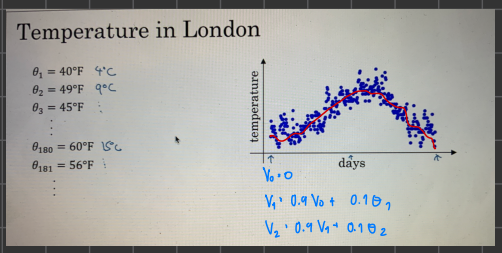
2^6 2^7 2^8 2^9 2^{10}

Acess Time help

make sure mini-batch fit in CPU (GPU memory

$(X^{\{t\}}, Y^{\{t\}})$

Exponentially weighted averages

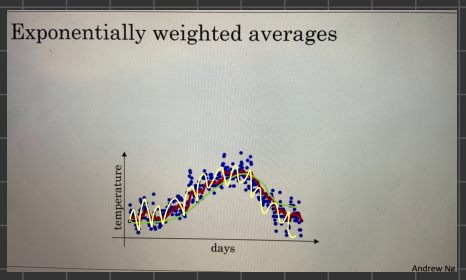


$V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

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



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

$\beta = 0.9$ ≈ 10 days

$\beta = 0.98$ ≈ 50 days

$\beta = 0.5$ ≈ 2 days

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

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

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

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

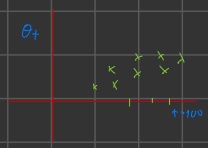
$V_{99} = 0.9 V_{98} + 0.1 \theta_{99}$

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

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

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

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

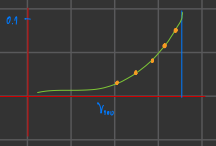


$0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$ ← The point to change

When β^{ε} will be $\frac{1}{e}$

Point of change

$0.98^{?} \to 0.98^{50} \approx \frac{1}{e}$



Implementation

$V_0 = 0$

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

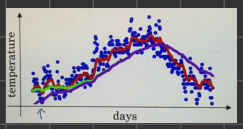
$V_0 = \beta v + (1-\beta)\theta_2$

$V_0 = 0$

Repeat & Get next θ_t

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

Bias Correction



When add more to average the windows will move right and slow to change

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

$V_0 = 0$

$V_1 = 0.98(0) + 0.02\,\theta_1$

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

$V_2 = 0.98(0.02\,\theta_1) + 0.02\,\theta_2$

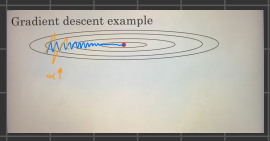
$= 0.0196\,\theta_1 + 0.02\,\theta_2$

$\frac{V_t}{1-\beta^t}$ ← This help an Exponentially weight to help estimate value exactly bia

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

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

Momentum



↕ slower learning

↔ faster learning

Momentum:

on iteration t:

Compute dw, db on current mini-batch

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

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

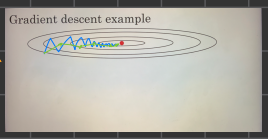
direction ← velocity

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

$V_{dw} = \beta V_{dw} + dw$

Hyperparameter: α, β $\beta = 0.9$

RMSProp



↕ slow

↔ fast

On iteration t:

Compute dw, db on current mini-batch

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

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

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

add ε to S_{dw} S_{db} to prevent divide by 0

$\varepsilon = 10^{-8}$

Adam optimization

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

on iteration t:

Compute dw, db using current mini-batch

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

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

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

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

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

Combine Momentum and RMSProp

α: needs to be tune

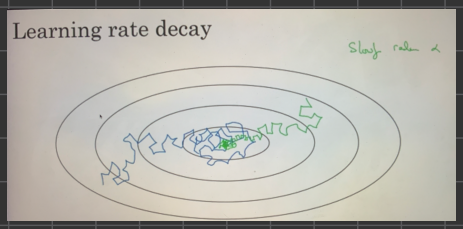
β_1: 0.9 (dw)

β_2: 0.999 (dw^2)

ε: 10^{-8}

Adam: Adaptive Moment Estimation

Learning rate decay



1 epoch = 1 pass through data

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

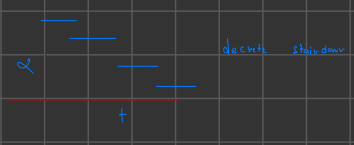
Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	⋮

$\alpha_0 = 0.2$, decay rate = 1

Other

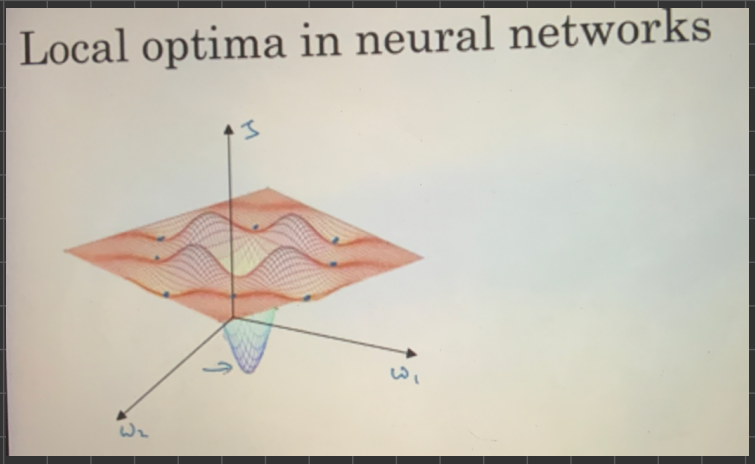
$\alpha = 0.95^{\text{epoch num}} \alpha_0$ → Exponentially decay

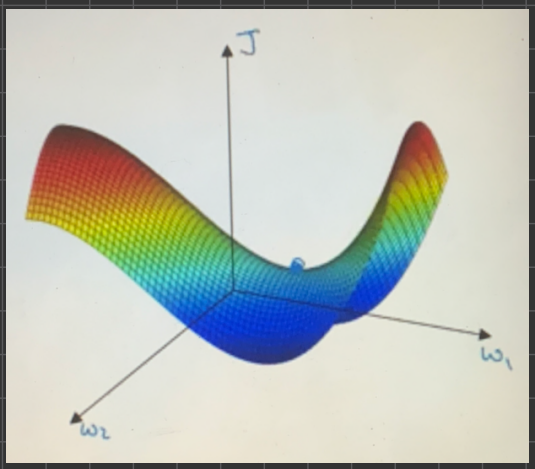
$\alpha = \frac{k}{\sqrt{\text{epoch num}}} \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$



Manual decay

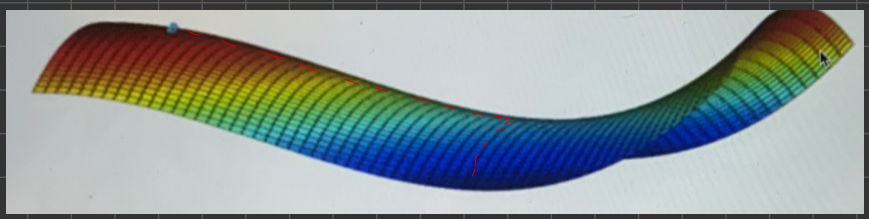
The problem of local optima





Feature
RE 50 000
J= 2^30,000

Plateaus ← The real problem



- Unlikely to get stuck in bad local optima
- Plateaus can make learning slow

Week 3

Hyperparameter

α ← most importance

β ≈ 0.9

$\beta_1, \beta_2, \varepsilon$ ≈ 0.9, 0.999, 10^{-8}

※ layer ←

※ hidden units ← need to select

Learning rate decay ←

Mini-batch size ← $2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}$

- Try to random value : Don't use a grid → to select Hyperparameter

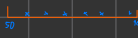
  because some Hyperparameter have different weight than one it's hard to weight which one is most importance

Coarse to fine

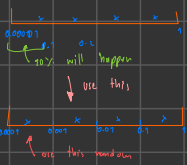
When find the best one of H_1 from H_1, H_2 then to create the new windowns around H_1 for find suitable H_1 and H_2

Using an appropriate scale to pick hyperparameter

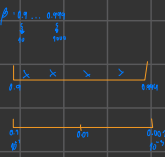
$n^{[2]}$: 50, ..., 100



α : 0.0001, ..., 1



β : 0.9 ... 0.999



※ layer L : 2 - 4

2, 3, 4, 4

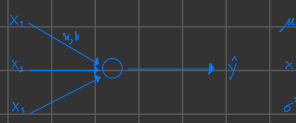
r = np.random.rand() ← r ∈ [-4, 0]

$\alpha = 10^r$ ← $10^a, \ldots, 10^b$

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

$\beta = 10^r$, $\beta = 1 - 10^r$

Batch Normalization

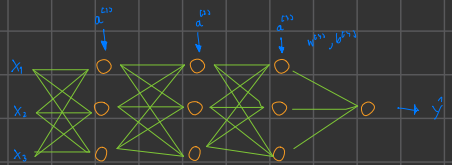


$\mu = \frac{1}{m} \sum_i x^{(i)}$

$x := x - \mu$

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

$x := x / \sigma^2$



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

Normalize $z^{[2]}$

Implement Batch Norm

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

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

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

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

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

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

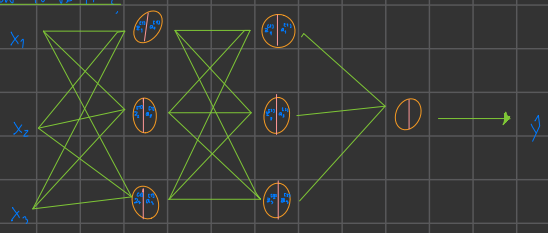
$\gamma = \sqrt{\sigma^2 + \varepsilon}$ ←

$\beta = \mu$ ←

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

Use $\tilde{z}^{(i)[2]}$ instead of $z^{(i)[2]}$

How to use it ?



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

Parameter : $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}$

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

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

not same as Momentum or RMSProp or Adam

Working with mini-batch

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

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

When use batch norm, can omit this

Parameter : ~~$b^{[1]}$~~, $\beta^{[1]}, \gamma^{[1]}$

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

$z^{[l]}, W^{[l]} a^{[l-1]}$

$z^{[l]}_{norm}$

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

with GD

for t=1 : Mini-batch_epoch

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

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

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

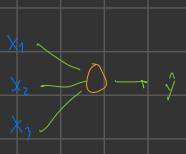
Update parameter $W^{[l]} := W^{[l]} - \alpha dw^{[l]}$

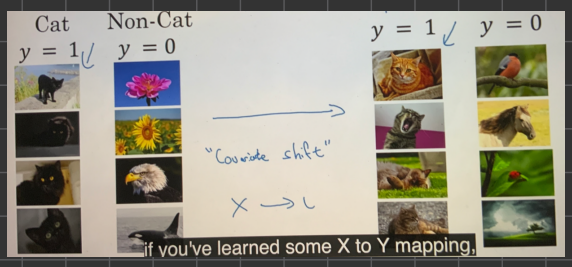
$\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$

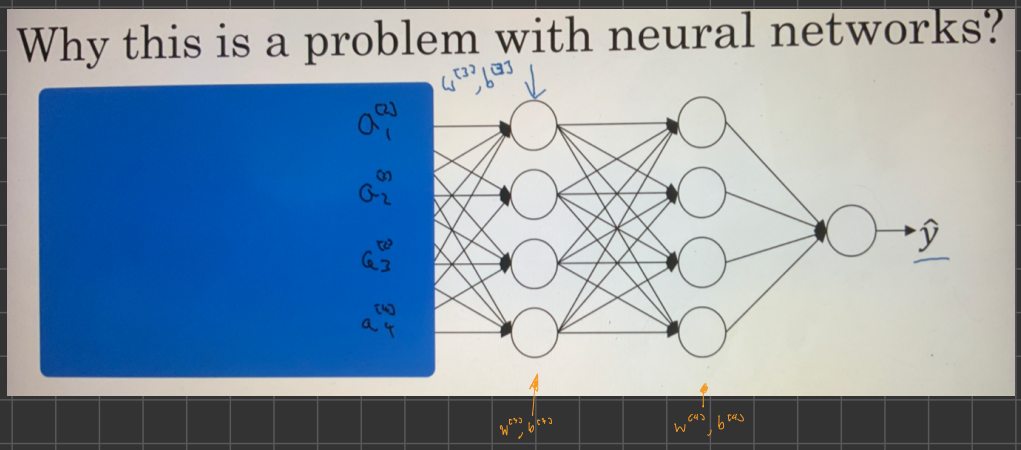
$\gamma^{[l]} := \dots$

Can use with Adam, momentum, RMSProb

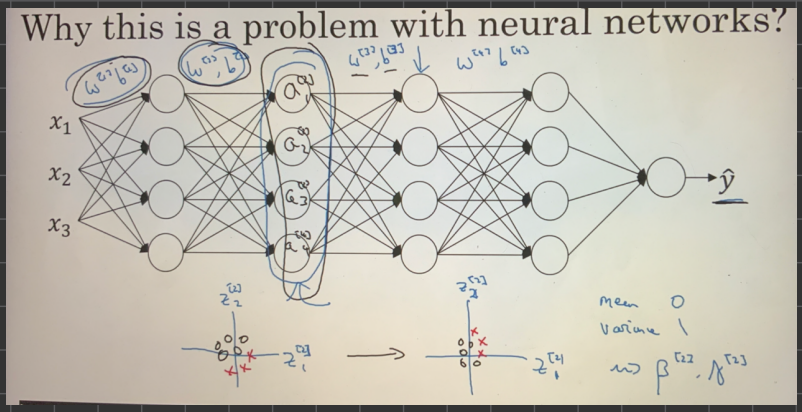
Why does it work







Batch Normalization try to help deep layers not to change mean and variance even the value of deeper layer is update that help deeper layer has ground



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.

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

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

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

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

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

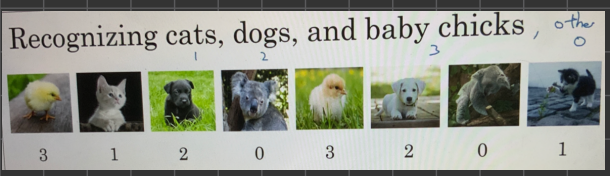
$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$

$\theta_1, \theta_2, \theta_3, \ldots$

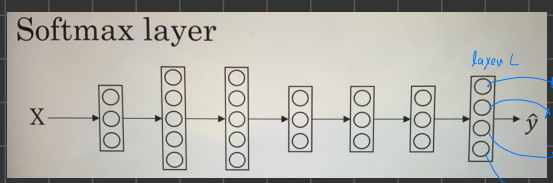
$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$, $\tilde{z} = \gamma z_{norm} + \beta$

Softmax Regression

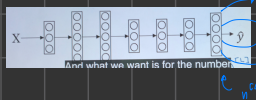
Recognizing cats, dogs, and baby chicks, other



3 1 2 0 3 2 0 1



C = # classes = 4 $(0, \ldots, 4)$



$P(\text{coffee}|x)$
$P(\text{cat}|x)$
$P(\text{dog}|x)$
$P(\text{the}|x)$

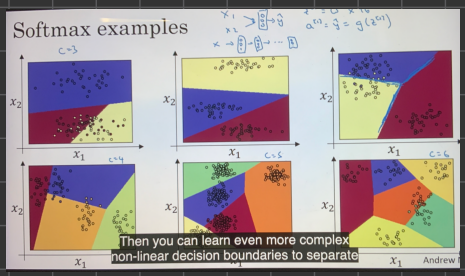
$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$

Activation function : softmax

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

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

more example



How to use it

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

"soft max"

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

"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 regress $a^{[L]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$

Loss fuction

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

$\rightarrow y_1 = y_2 = y_4 = 0$

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

sum 1

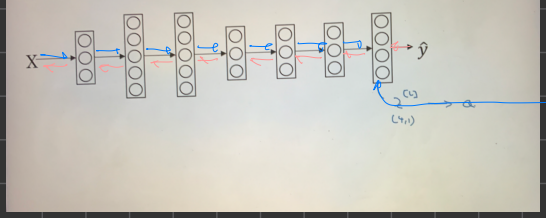
$-y_2 \log \hat{y}_2 = -\log \hat{y}_2$ make $\hat{y}_2$ big as possible

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

$Y = [\, y^{(1)} \; y^{(2)} \dots y^{(m)} \,] = \begin{bmatrix} 0 & 0 & 1 & \dots \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}$ $(4, m)$

$\hat{Y} = [\, \hat{y}^{(1)}, \dots, y^{(m)} \,] = \begin{bmatrix} 0.3 & \dots \\ 0.2 & \\ 0.1 & \\ 0 & \end{bmatrix}$ $(4, m)$

Gradient descent with softmax



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

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

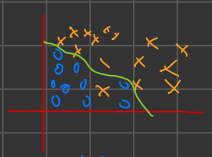
$\frac{\partial J}{\partial z^{[L]}}$

Summary Optimized NN

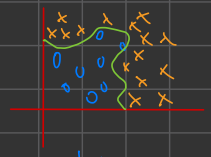
Bias / Variance

high bias

underfit



just right



high variance

overfit

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	high variance	high bias	high bias high variance	low bias low variance

← We want this

Regularization

in NN we have sigma to sum overall cost

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

omit b λ: regularization parameter

L_2 regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w \xrightarrow{NN} \|w^{[L]}\|_F^2 = \sum_{i=1}^{n^{[L-1]}} \sum_{j=1}^{n^{[L]}} (w_{ij}^{[L]})^2$ $w: (n^{[L]}, n^{[L-1]})$ "Frobenius norm"

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

dw: Backprop $+\frac{\lambda}{m} w^{[L]}$, $w^{[L]} := w^{[L]} - \alpha\, dw^{[L]}$

(In regularization we call weight decay)

Dropout regularization

random prob of layer in NN to delete node to help NN learn faster (, but Maybe can worse if set too high)

Example

L=3 keep_prop = 0.8

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

a3 = np.multiply(a3, d3)

a3 /= keep_prob → to not change expect value

Normalizing

- Max-min Norm
- Z-Norm

Weight Initialization

$w^{[L]}$ = np.random.randn(shape) * np.sqrt($\frac{1}{n^{[L-1]}}$)

"Xavier initialization"

Gradient checking

- help to make sure our backprop is correct

→ help NN to learn faster

Mini batch

separate the data into set to help NN learn faster

Momentum

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

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

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

RMS prob

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

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

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

Adam

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

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

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

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

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

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

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

α: need to tune

β_1: 0.9 (dw) Momentum

β_2: 0.999 (dw^2) RMS prob

ε: 10^{-8}

Learning rate decay

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

$\alpha = \frac{k}{\sqrt{\text{epoch num}}} \cdot \alpha_0$

Batch Norm

Normalization in hidden layer not only input

(it help deep layer do

have same mean and variance

, even it value is change, like to have ground on it)

Softmax Regression

it change activate function of output to possibility

$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{n^{[L]}} t_i}$, $t_i = e^{(z_i^{[L]})}$