COURSERA - Improving Deep Neural Networks -

⇒ Train | Dev | Test sets

Applied ML is highly iterative process

layers # hidden units learning rates, activation functions

→ Data.

Training set Dev set Test set

- Hold out
- cross validation set.

Mismatched train | test distribution

Training set: cat pictures from webpages ⟷ Dev | test | sets: cat pictures from your app

→ make sure dev and test set come from same distribution.

→ Not having a test set might be okay (only dev set).

⇒ Bias / Variance

let's take cat classification.

		overfit	underfit	just right.
Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance, low bias	high bias, low variance.	high bias, high variance	low bias, low variance

bias → due to inaccuracy of training set (high training error)

variance → due to difference between dev set error and training set error

⇒ Basic Recipe for Machine Learning

High bias? (training data performance) —try→ bigger network, Hidden layers. (NN architecture)

S. we firstly try to get rid of bias problem.

↓

High variance? (dev set performance) —try→ more data, regularisation (NN architecture)

↓

Done

⇒ Regularisation.

Try when high variance is there. → overfitting.

Logistic regression

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

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

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

$\lambda \rightarrow$ regularisation parameter.

L2 regularisation $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L1 regularisation $\rightarrow \frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}\|w\|_1$

↳ w will be sparse (lot of zeros)

Neural Network

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

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

"Frobenius norm"

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

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

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

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

called as weight decay.

⇒ Why regularisation reduces overfitting ?

⇒ Dropout regularisation

↳ Dropout some nodes randomly during iterations

Implementing dropout ("Inverted dropout")

Illustrate 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 = a3 / keep_prob   (scaling factor).
```

For example, 50 units ⟶ 10 units shut off

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

↳ reduced by 20%

thus, scale up a little by $a^{[3]} = \frac{a^{[3]}}{0.8}$.

⟶ No dropout is done during test time.

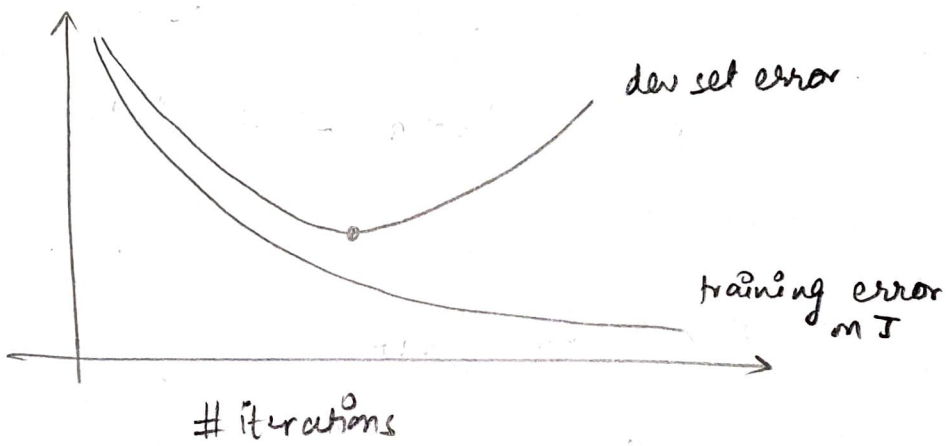
⇒ Other regularisation techniques

(i) Data augmentation

↳ use flipped, tilted images

↳ distorted digits

(ii) Early stopping



Stop where dev-set error is minimum.

⇒ Normalising inputs

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

subtract mean

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

$X = X - \mu$

Normalise variance

$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} X^{(i)} ** 2$ (element wise.)

$X = \frac{X}{\sigma}$ → standard deviation

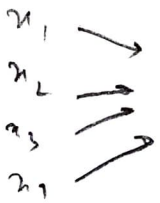
So, in total. $X = \frac{X - \mu}{\sigma}$

⇒ Vanishing / Exploding gradients

↳ This can be solved by careful initialisation of weights.

⇒ Weight initialisation for Deep NN

single neuron example



$z = w_1 x_1 + w_2 x_2 \dots w_n x_n + b$ (b → 0)

large n → small w_i

$Var(w_i) = \frac{1}{n}$, $\frac{2}{n}$

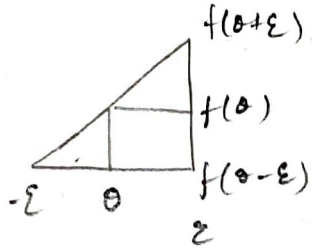
$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

This helps reduce the vanishing / exploding gradient problem

For relu → $\sqrt{\frac{2}{n}}$, tanh → $\sqrt{\frac{1}{n}}$, other. $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

⇒ Numerical approximation of gradients

let $f(\theta) = \theta^3$ $\quad g(\theta) = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$



$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ is much more close to $f'(\theta)$

↳ error $O(\varepsilon^2)$ order

whereas $\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} \rightarrow O(\varepsilon)$ order error.

⇒ Gradient checking

Take $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \dots W^{[l]}, b^{[l]}$ and reshape into a vector θ

$$J(w^1, b^1, w^2, b^2 \dots b^l) = J(\theta)$$

Take $dw^1, db^1 \dots dw^{[l]}, db^{[l]}$ and reshape into a big vector $d\theta$

(concatenate)

Gradient checking

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

for each i:

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

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

check $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx$ 10^{-7} great, 10^{-5} aware, 10^{-3} concerned

$$\varepsilon = 10^{-7}$$

⇒ Gradient checking implementation notes

→ Don't use in training - only to debug.

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

→ Remember regularization.

→ Doesn't work with dropout

→ Run at random initialisation again after training.

⇒ Week-2 optimization Algorithms

⇒ Mini Batch Gradient Descent

If m is very large, like m = 5,000,000

minibatches of m = 1000

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

$$Y = [\underbrace{y^{(1)} y^{(2)} \dots}_{Y^{\{1\}}} \underbrace{\dots}_{Y^{\{2\}}} \dots \underbrace{y^{(m)}}_{Y^{\{5000\}}}]$$

$X^{\{1\}} \rightarrow (n_x, 1000)$ $X^{\{2\}} \rightarrow (n_x, 1000)$ ...

$Y^{\{1\}} \rightarrow (1, 1000)$, $Y^{\{2\}} \rightarrow (1, 1000)$

process mini batches.

Mini batch gradient descent

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

for $t = 1 \dots 5000$ {

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

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

} vectorise 1000 examples.

compute cost

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

Back prop to compute gradients. cost $J^{\{t\}}$ (using $X^{\{t\}}, Y^{\{t\}}$)

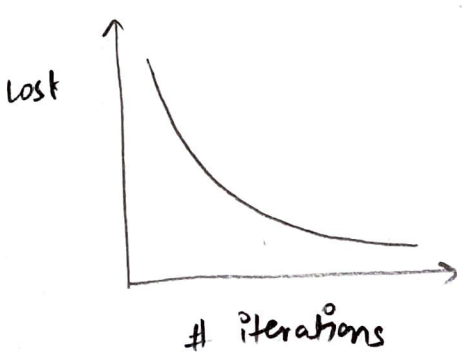
$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

}

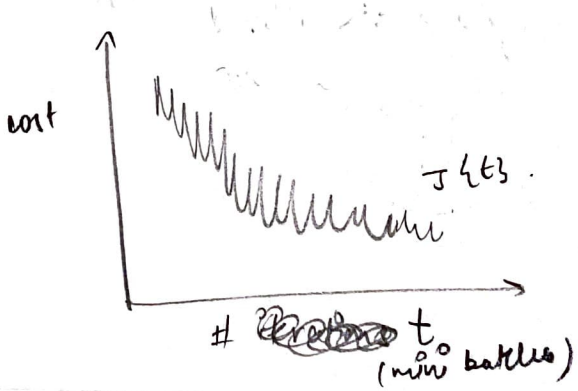
one "epoch"

→ single pass through training set.

Batch grad. descent



Mini batch GD



Fluctuations due to different training sets, however, overall trend is downward

Choosing your mini batch

mini-batch size = m : Batch grad descent $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

mini-batch size = 1

Stochastic Gradient Descent

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

. Every example is its own mini batch.

practical mini-batch size is somewhere between those.

BGD → Too long time per iteration.

SGD → Lose speedup from vectorisation.

In between → Fast learning → vectorisation

SGD → makes progress without processing extremely tiny set.

small training set : Use batch GD.
($m \leq 2000$)

Typical mini batch sizes:

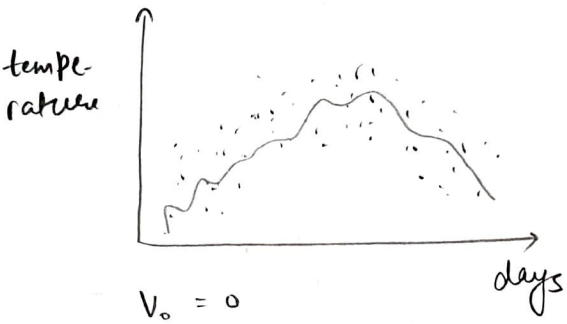
64, 128, 256, 512

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

⇒ Exponentially weighted averages

Temperature in London

$\theta_1 = 40°F$
$\theta_2 = 49°F$
$\theta_3 = 45°F$
⋮
$\theta_{180} = 60°F$
$\theta_{181} = 56°F$
⋮



$V_0 = 0$
$V_1 = 0.9V_0 + \theta_1$
$V_2 = 0.9V_1 + \theta_2$
$V_3 = 0.9V_2 + \theta_3$.
⋮

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

$\beta = 0.9$: $\approx$ 10 days ← Just right
$\beta = 0.98$: $\approx$ 50 days → Too smooth, bad latency.
$\beta = 0.5$: $\approx$ 2 days → Too much noise.

V_t is interpreted as approximate average over $\approx \frac{1}{1-\beta}$ days of temperature.

⇒ Exponentially weighted averages

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

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

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

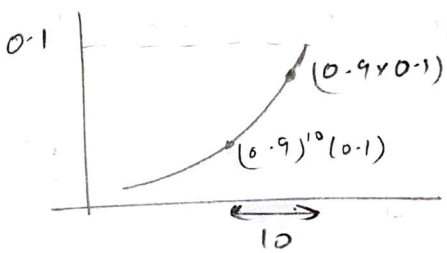
$$\vdots$$

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

$$= 0.1\theta_{100} + 0.9 \times 0.1\theta_{99} + 0.9 \times 0.9 \times 0.1\theta_{98}$$

$$+ (0.9)^3(0.1)\theta_{97} \cdots$$

$*$



$\varepsilon = 1 - \beta$.

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

$$(0.9)^{10} = \frac{1}{e} \approx 0.35 \approx \frac{1}{3}^{rd}$$

⇒ Understanding exponentially weighted averages

implementation

$V_0 := 0$

$V_0 := \beta V + (1-\beta)\theta_1$

$V_0 := \beta V + (1-\beta)\theta_2$

$\vdots$

$V_0 := 0$

repeat {

get next θ_t

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

}

Just one line of code.

⇒ Bias correction

During initial phase $v \approx 0$, we need to correct this;

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

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

$v_0 = 0$

$v_1 = 0.98 v_0 + 0.02\theta_1$

$v_2 = 0.98 v_1 + 0.02\theta_2$

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

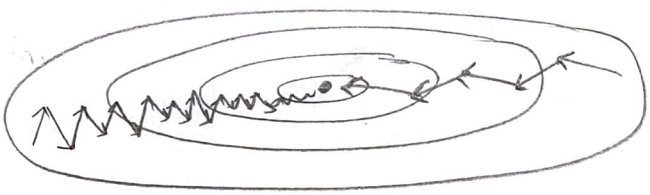
$= 0.0196\theta_1 + 0.02\theta_2$

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

$= 0.0396$

$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

⇒ Gradient Descent with momentum



Momentum:

on iteration t:

compute dw, db on current minibatch

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

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

$W = W - \alpha V_{dw}.$

Implementation details

Hyperparameters: α, β — 0.9 generally

On iteration t:

$V_{dw} = 0$ (same dim as dW) $\quad V_{db} = 0$ (same dim as db)

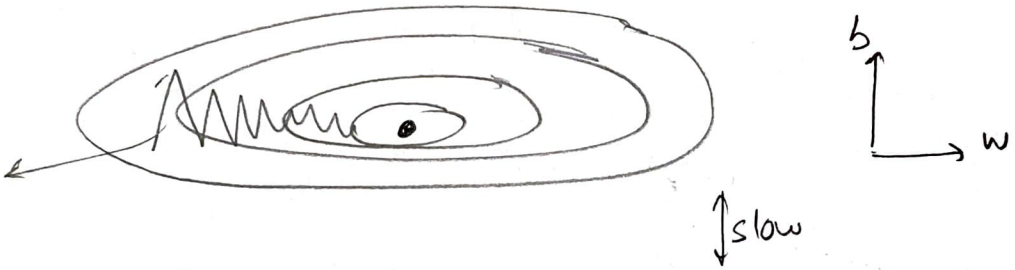
compute dW, db on current mini-batch.

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

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

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

$\Rightarrow$ **RMS Prop** $\rightarrow$ Root mean square prop



On iteration t

compute dw, db on current mini-batch.

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2 \quad \text{(elementwise square)}$$

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

$$W = W - \frac{\alpha\, dw}{\sqrt{S_{dw}+\varepsilon}} \qquad b = b - \frac{\alpha\, db}{\sqrt{S_{db}+\varepsilon}}$$

just to ensure, nothing explodes.

Now, we want learning to go fast in w direction.

So, dw should be small and db should be greater and that is indeed the case with our contours.

⇒ Adam optimisation Algorithm

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

(Initialisation)

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)

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

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

("RMS prop" β_2)

$$V_{dw}^{corrected} = \frac{V_{dw}}{1-\beta_1^t} \qquad V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$$

$$S_{dw}^{corrected} = \frac{S_{dw}}{1-\beta_2^t} \qquad S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$$

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

Hyperparameters choice

α : needs to be tuned

β_1 : 0.9 (dw)

β_2 : 0.999 (dw^2)

ε : 10^{-8}

} generally accepted values.

Adam : Adaptive moment estimation

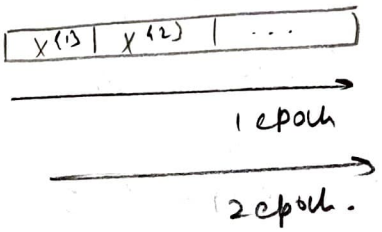
⇒ Learning rate Decay

↳ decrease learning rate over iterations.

slowly reduce α.

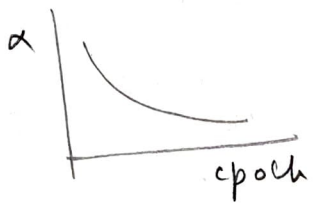
1 epoch = 1 pass through the data.

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



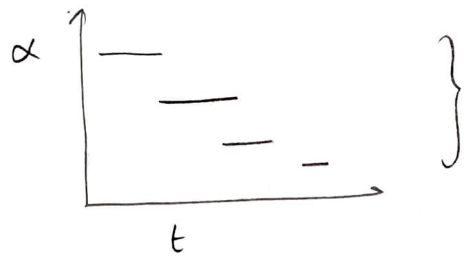
let $\alpha_0 = 0.2$ decay rate = 1

epoch	α
1	0.1
2	0.067
3	0.05
4	0.04



Also, $\alpha = 0.95^{\text{epoch num}} \times \alpha_0$ (exponential decay)

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



} discrete α decay.

⇒ The problem of local optima

↳ In higher dimensions, most of the points are saddle points and there is a very less chance of an absolute minimum.

⇒ Week-3 Hyperparameter tuning

⇒ Hyperparameters

Most important ← α # layers

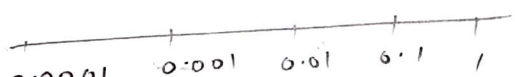
$\beta \sim 0.9$ # hidden units

$\beta_1, \beta_2, \varepsilon$ learning rate decay.
(0.9) (0.999) (10^{-8})

mini-batch size.

Try random values → Don't use a grid, try random values.

α — log scale



$r = -4 * np.random.rand()$

↳ $r \in [-4, 0]$

$\alpha = 10^r$ → 10^{-4} to 1

⇒ Batch Normalisation

normalising inputs (X) works and speeds up GD.

So, can we normalise every $a^{[l-1]}$ to train every $w^{[l]}$, $b^{[l]}$ faster. We would actually normalise $z^{[l-1]}$.

→ Implementing batch norm

Given some intermediate values in NN. $z^{[l](i)}$

$z^{(1)}, z^{(2)} \dots z^{(m)}$

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

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

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

learnable parameters.

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

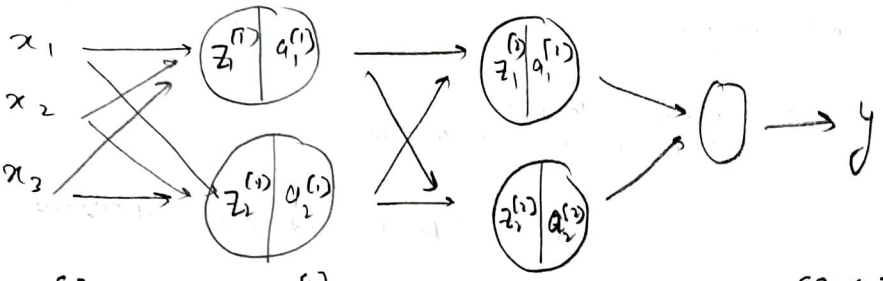
If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ $\beta = \mu$. } normalisation step is inverted.

$$z^{\sim(i)} = z^{(i)}$$

So, using γ, β to make different mean & variance other than 0 and 1.

We do not always want mean and variance for $z^{(i)}$'s to be 0 and 1. With γ and β we can control them

→ Fitting Batch Norm into a Neural Network



$w^{[1]}, b^{[1]}$ $\quad$ $\beta^{[1]}, \gamma^{[1]}$ $\quad$ $w^{[2]}, b^{[2]}$

$X \longrightarrow z^{[1]} \longrightarrow \tilde{z}^{[1]} \longrightarrow a^{[1]} = g(\tilde{z}^{[1]}) \longrightarrow z^{[2]}$

Batch norm

$a^{[2]} \leftarrow \tilde{z}^{[2]}$ $\quad$ $\beta^{[2]}, \gamma^{[2]}$

Parameters → $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \dots w^{[L]}, b^{[L]}$

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

↳ different β used in momentum & Adam.

We update β and γ similar to w and b

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

Working with mini batches

$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow$

$X^{\{2\}} \longrightarrow z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow \dots$

$X^{\{3\}} \longrightarrow \dots$

Parameters $w^{[1]}$, ~~$b^{[1]}$~~, $\beta^{[1]}$, $\gamma^{[1]}$

↳ no longer required as it gets cancelled in znorm.

parameters $w^{[l]}$, ~~$b^{[l]}$~~, $\beta^{[l]}$, $\gamma^{[l]}$

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

$z^{[l]} \to (n^{[l]}, 1)$ $\quad z^{[l]} = w^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$

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

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

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

elementwise multiplication.

⇒ Implementing GD

for t = 1 .... num mini batches.

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

In each hidden layer, BN to get $\tilde{z}^{[l]}$ from $z^{[l]}$

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

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

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

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

works same manner with Adam, RMSprop, momentum.

⇒ Batch norm at test-time

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

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

m = mini batch size here.

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

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

μ and σ used here cannot be applied at test time because they are different for different batches.

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

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

$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]} \ldots$ → exponentially weighted. $\mu^{[l]}$

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

$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]}$ → exponentially weighted $\sigma^{2[l]}$

$$Z_{norm} = \frac{Z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$ } test time $\quad \tilde{Z} = \gamma Z_{norm} + \beta$.

⇒ Softmax regression

↳ multiclass logistic regression.

For example, we need to recognise, cats, dogs, baby chicks.

C = # classes = $4 \quad (0, 1, 2, 3)$

$X \rightarrow$ [NN] $\rightarrow \hat{y}$

- 0 → $P(\text{other} | X)$
- 0 → $P(\text{cat} | X)$
- 0 → $P(\text{dog} | X)$
- 0 → $P(\text{bc} | X)$

↳ $n^{[L]} = 4 = C$

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

Softmax layer

$$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]} \quad (4,1)$$

Activation function :

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

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum t_i} \quad \text{if} \quad \frac{t_i}{\sum t_i} = a_i^{[L]}$$

Loss function in softmax

$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ → cat $\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ say.

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

For this example $y_1 = y_3 = y_4 = 0$

$$\Rightarrow \mathcal{L}(\hat{y}, y) = -y_2 \log \hat{y}_2 = -\log \hat{y}_2$$, make $\hat{y}_2$ big to minimise loss function.

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

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

$\hat{Y}$ = $(4 \times m)$ matrix.

→ GD with softmax

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

Backprop: $dz^{[1]} = \hat{y} - y$ → derive

$(4,1)$

↳ $\frac{dJ}{dz^{[L]}}$