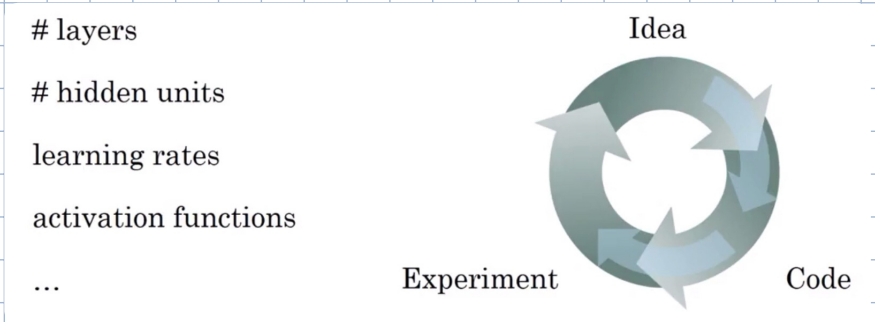
Course 2: Improving Deep Neural Networks:

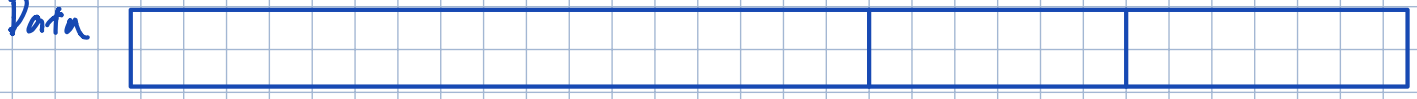
Hyperparameter tuning, Regularisation and Optimisation

Week 1: Practical aspects of Deep Learning

Applied ML is a highly iterative process



Train / dev / test sets



training set

hold-out set
OR cross-validation set
OR development set
→ "dev set"

test set

Previous Era:

70% - 30% split OR 60% - 20% - 20%

Big data era:

% of dev set and test set can be much smaller.

98% - 1% - 1% (for large amount of data)

Mismatched train/test distribution

e.g. a model to find picture of cats among user-uploaded picture.

training set:
cat pictures from
web pages

↓

nice, clear pic

Dev/test set:
cat pictures from users
using your app

↓

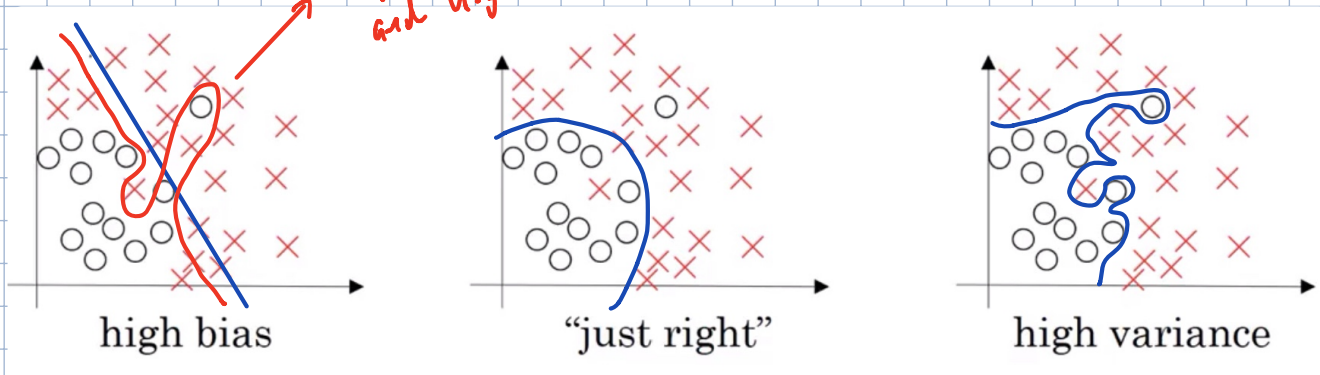
blur, casual pic

→ distribution of data maybe different

↳ make sure the dev/test set come from the same distribution
↳ we will be using the dev set to evaluate a lot of different models

It might be ok to not have a test set (only dev set)

Bias / Variance



underfitting

overfitting

In high dimension problem, can't visualize decision boundary


y=1 y=0

training set error:	1%	15%	15%	0.5%
dev set error:	11%	16%	30%	1%
	high variance (can't generalise well)	high bias ↓ underfit	high bias AND high variance	low bias low variance

Assume Human can achieve ~ 0% error

↳ or Optimal (Bayes) error ~ 0% error

→ really blur image ⇒ Bayes error will be higher

Basic Recipe for ML

After trained an initial model,

first ask: does the algorithm have high bias? → training set performance

↳ if yes → bigger network / different architectures

(keep trying until no longer high bias)

i.e. at least fit the training set well

then ask: do we have a variance problem? → dev set performance

↳ if yes → more data / regularisation / architecture

Bias Variance trade-off.

↑ ↓

↓ ↑

in modern era, getting a bigger network almost always just reduce bias w/o hurting variance. Getting more data pretty much always reduce variance w/o hurting bias.

With regularisation, training a bigger network almost never hurts.

Regularisation in neural network

If we find the network is over fitting the data, one of the first thing to try is probably regularisation. (another reliable way is to get more data)

Lets develop this idea using logistic regression.

recall we want to minimise the cost function J:

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

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

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

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

If we use L1 regularisation, w will end up being sparse.

↳ lasso

λ = regularisation parameter

How about neural network?

In a NN, we have a cost function that's a function of all our parameters:

L: no. of layers

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

where the squared norm of a matrix

$n^{[l]}$: no. of neurons in layer l

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

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

"Frobenius norm"

↳ sum of square of elements of a matrix

with regularisation

How to compute gradient descent with this?

→ Previously, we would complete $dw^{[l]} = (\text{back prop}) + \frac{\lambda}{m} w^{[l]}$

where back prop gives us $\frac{\partial J}{\partial w^{[l]}}$ ← still holds with regularisation

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

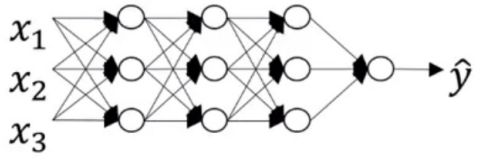
w/o regularisation

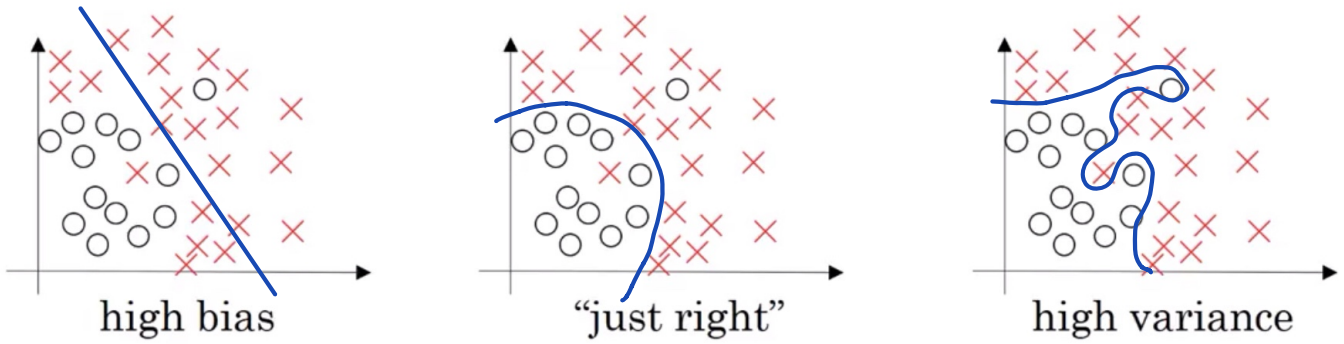
L2 is also called weight decay: $w^{[l]} := w^{[l]} - \alpha\left[(\text{from back prop}) + \frac{\lambda}{m} w^{[l]}\right]$

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

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

Why Regularisation reduces overfitting?





Say we have a neural network that is currently overfitting.

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

↙ penalises the weight matrices from being too large

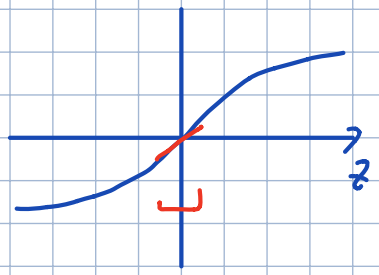
if λ is very large $\longrightarrow w^{[l]} \approx 0$

↳ a lot of neurons will be zero (sort of)

↳ became a simplified network

↳ less variance

Another intuition:



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

if z is small $\longrightarrow$ linear region of tanh is used

if λ is large, then $W^{[l]}$ is small

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

↳ z is also small

↳ $g(z)$ is roughly linearly

↳ every layer will be roughly linear

↳ whole network is linear

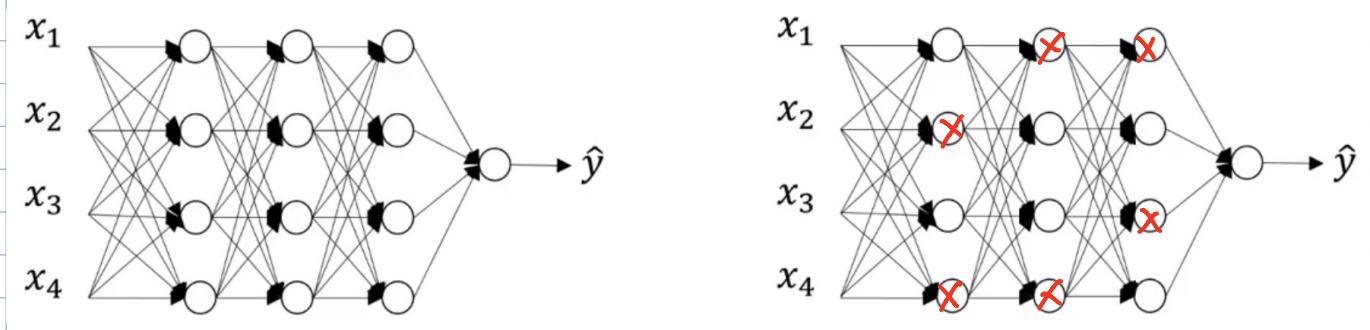
↳ less variance

when implementing gradient descent, can plot:

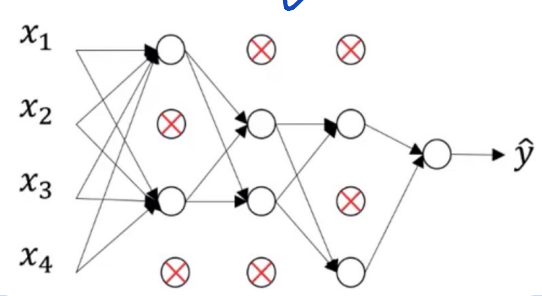
must use J with regularisation term

Dropout Regularisation

Let's say we train a neural network and there's overfitting.



remove all outgoing also



With dropout, we go through each of the layers of the network and set some probability of eliminating a node in the network

e.g. for each node, we toss a coin, have a 0.5 chance of keeping each node and 0.5 chance of removing the node.

↳ actually works as regularisation, as the network is simpler.

There are a few ways of implementing dropout, the most common way is called "Inverted dropout"

Let's say we want to illustrate this with layer $l=3$

we set a vector "d"

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

- `d3`: dropout vector for layer 3
- each element: 0.8 chance = 1, 0.2 chance = 0
- `keep_prob`: a number e.g. 0.2 (probability a given hidden unit will be kept)

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

- `a3`: activation of layer 3
- `multiply`: element-wise

```
a3 /= keep_prob
```

← we need to scale-up

e.g. 50 neurons in 3rd hidden layer

↳ 10 neurons shut off

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

$a^{[3]}$ reduced by 20%

expected value reduced.

need scale-up

on various pass (e.g. gradient descent), we want to shrink out different neurons.

⇒ At test time, making prediction:

Given some $a^{[0]} = X$

NO Dropout in prediction

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = \dots$$

↓

$\hat{y}$

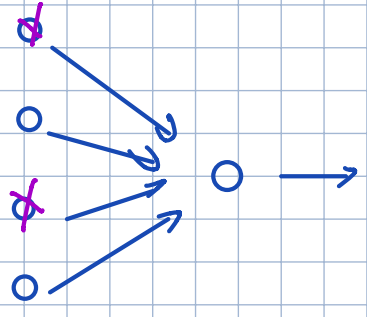
⇒ if randomly dropout in test time → add noise only

Understanding Dropout

Apart from resulting a smaller network, which has regularisation effect, here is another intuition

⇒ Can't rely on any one feature, so have to spread out weights.

Let's look at the perspective of a single unit.



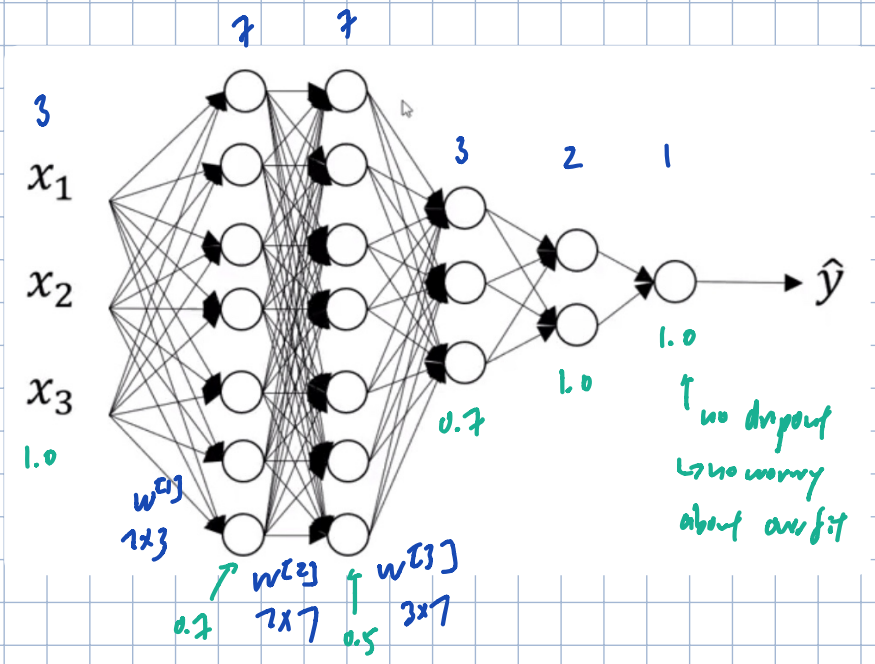
for dropout regularisation, input can get randomly eliminated

↳ this neuron can't rely on any one feature

↳ can't put a lot of weight on any one input

↳ motivated to spread out the weight

↳ shrink the squared norm of weights similar to L2



no. of neurons in previous layer = no. of col of w

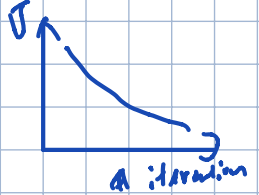
no. of neurons in current layer = no. of row of w

Not all layer uses dropout, because $w^{[2]}$ is the largest ⇒ maybe apply dropout on $w^{[2]}$

So for layer 2, keep-prob = 0.5

Downside: cost function J is not well-defined with dropout.

↳ on every iteration, we randomly kill off a bunch of nodes.



→ turn off dropout to see J decreases

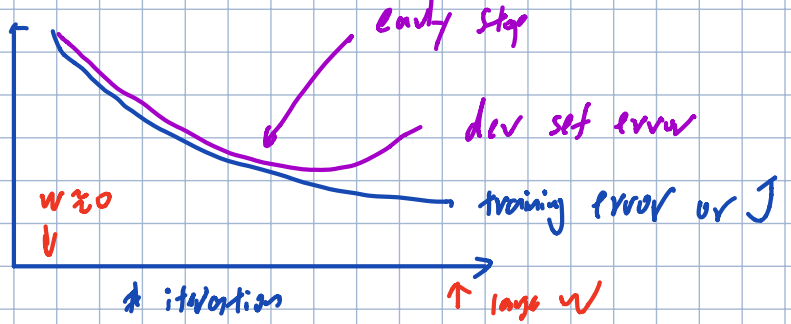
→ then turn on dropout and hope that no bugs.

Other regularisation method

① Data augmentation → create fake training examples

↳ more data → sort of regularise → reduce overfitting

② Early Stopping



random init

↳ w ≈ small values

Early stop gives W with medium size value

↳ Similar to L2 regularisation

Down side:

we suppose to optimize cost function J → gradient descent, ...

↳ also not overfitting → Regularisation, ...

↳ 2 task separately, use different tools → orthogonalisation

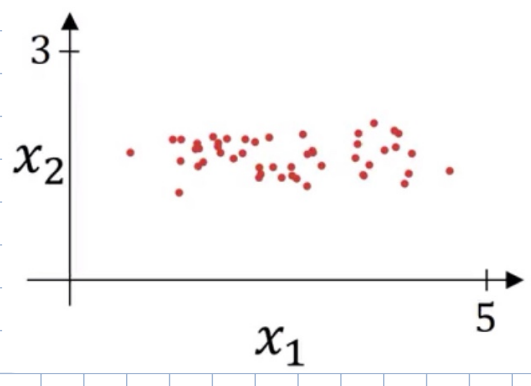
Early stop couples these 2 tasks

Setting up optimisation problem

Normalising Input

This is to speed up training

e.g. training set with 2 input features



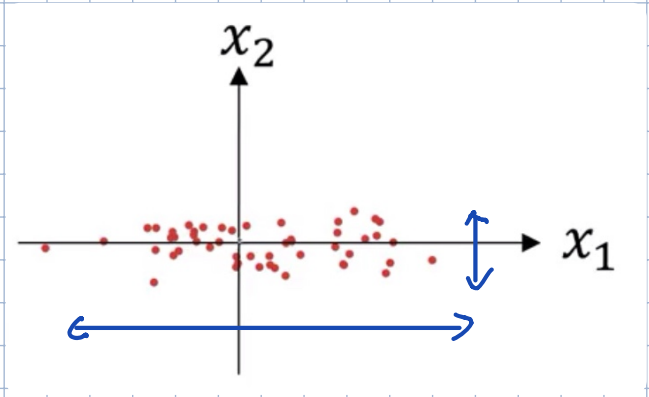
$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Normalise inputs involves 2 steps:

① Subtract mean:

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

$$x := x - \mu$$

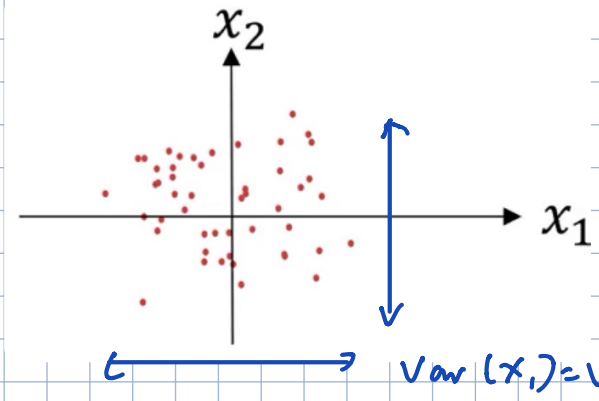


② normalize the variance

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \quad x \ast 2$ ← element wise

↑ we already subtracted mean in ①

$$\frac{x - \mu}{\sigma}$$



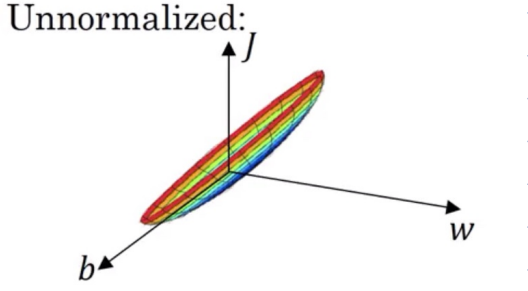
$Var(x_1) = Var(x_2) = 1$

* If we use this to scale training data

⇒ use the <u>same</u> μ and σ to scale test data!

<u>Why normalize inputs?</u>

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

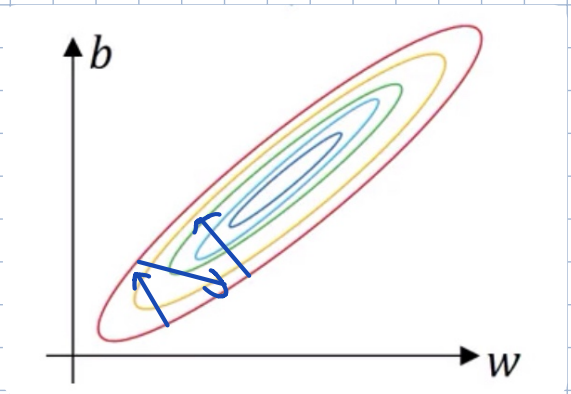


if $x_1 : 1, \ldots, 1000$

$x_2 : 0, \ldots, 1$

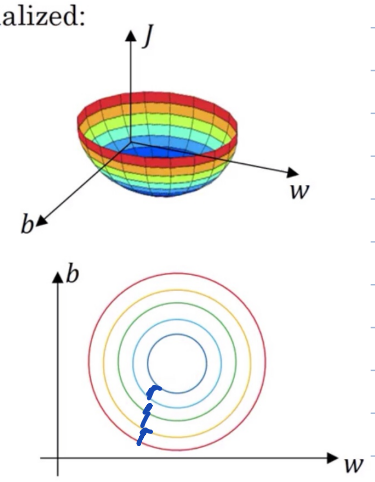
↳ w_1 and w_2 will have very different value

↳ elongated bowl



need small learning rate of gradient descent

Normalized:



can use larger learning rate

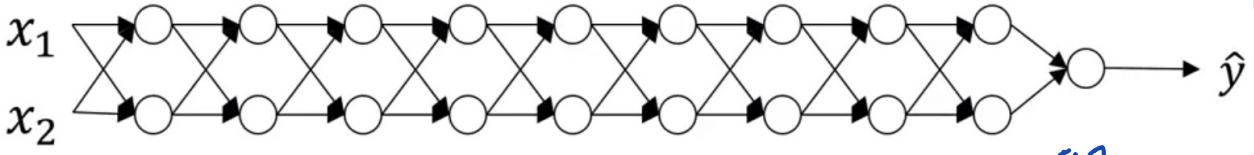
Vanishing / Exploding gradients

One of the problems when training NN, especially very deep NN is data vanishing and exploding gradients.

i.e. the derivatives / slopes can get very big or small

↳ use careful choices of the random weight initialisation to significantly reduce this problem

Let's say we train a very deep NN like this:



$w^{[1]}$ $w^{[2]}$ $w^{[3]}$ ... $w^{[L]}$

Say for simplification, we use linear activation

$$g(z) = z$$

and ignore b, i.e. $b^{[l]} = 0$

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

Say each $W^{[l]}$ is a bit larger than identity

$$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

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

↳ if L is large, then $\hat{y}$ will be large

If $W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$

↳ $\hat{y} = W^{[L]} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-1} X$

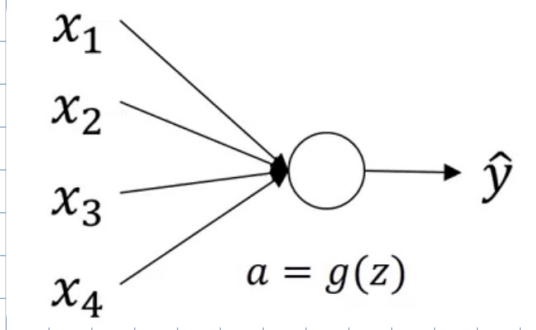
↳ if L is large, then $\hat{y}$ will be small

⇒ can be inferred to derivatives

if gradient is very small → takes long for gradient descent

weight initialisation for deep networks

single neuron example



$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b^{\nearrow 0}$

larger n ⟶ smaller w_i

∵ z is sum of $w_i x_i$, so if a lot terms we want each term be smaller

We can set $Var(w_i) = \frac{1}{n}$

where n is # of input feature into a neuron

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

if activation is ReLu, then use $\frac{2}{n^{[l-1]}}$

So if the input features of activations are roughly mean 0 and standard variance → results W not too large / smaller than 1.

Other variants:

if tanh is used

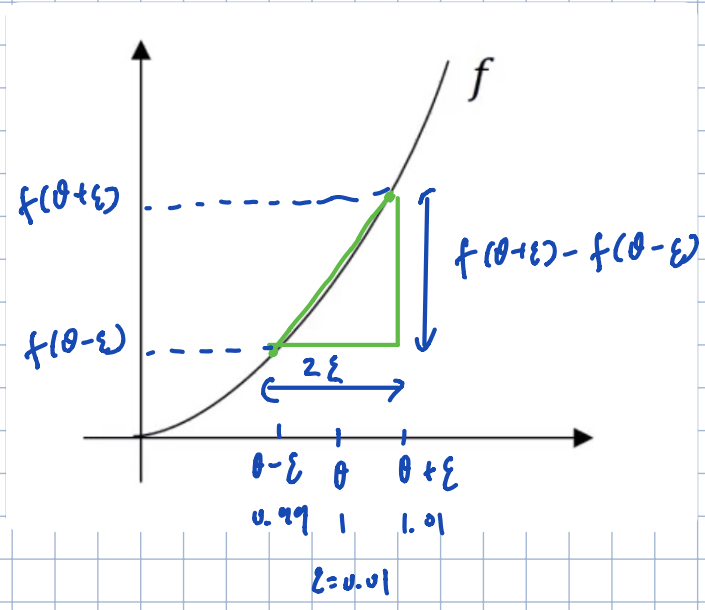
↳ $\sqrt{\frac{1}{n^{[l-1]}}}$ is used (Xavier initialisation)

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

Numerical approximation of gradients

When implementing back propagation, there is a test called gradient checking, that can help checking the back prop is correct.

↳ first, need approximation of gradients



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

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

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

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

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

approx error : 0.0001

→ error $O(\varepsilon^2)$

if $\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{\varepsilon}$ → error $O(\varepsilon)$

Gradient Checking

The NN will have some parameters:

take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ (concatenate)

↳ $J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) \rightarrow J(\theta)$

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

(θ and $d\theta$ has the same dimension)

→ Is $d\theta$ the gradient/slope of J?

Grad check: (recall $J(\theta) = J(\theta_1, \theta_2, \dots)$

for each i: i.e. each component of θ

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

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

↳ end up with $d\theta_{approx}$, same dimension as $d\theta$

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

$\| \cdot \|_2$: 2 norm: sum of squares of elements of the difference, then sqrt

$\| d\theta_{approx} \|_2$, $\| d\theta \|_2$: sum of squares of element then sqrt

↝ $\varepsilon = 10^{-7}$

if it ≈	
10^{-7}	good
10^{-5}	hmm...
10^{-3}	wrong

Gradient Checking Implementation Notes

→ Do Not use in training → only to debug

i.e. don't calculate $d\theta_{approx}[i]$ in training

Just get $d\theta$ during back prop

↳ do not run this in every iteration of gradient descent

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

→ remember regularization term, as it is part of J

↳ $d\theta$ = grad of J wrt θ

→ Does NOT work with dropout

↳ implement grad check w/o dropout

↳ then turn on dropout

→ Run at random init; then maybe again after some training

↳ sometimes (rare), gradient descent is correct when W and b are close to 0, as W, b gets bigger, might not be correct.

Week 2: optimisation algorithms

Mini-batch gradient descent

Deep NN works well with large data set

↳ but train on large data is slow

Batch vs mini-batch gradient descent

vectorisation allows you to efficiently compute on m examples

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

(n_x, m)

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

$(1, m)$

what if $m = 5,000,000$?

traditionally, train 5 mil → 1 step of gradient descent ← batch method

but we can split to mini batches (say 1,000 examples each) ← mini batch method

↳ can train on mini batch and progress through gradient descent

⇒ we have 5000 mini batches

mini batch no. t: $X^{\{t\}}, Y^{\{t\}}$

$X^{\{t\}}: (n_x, 1000)$

$Y^{\{t\}}: (1, 1000)$

$x^{(i)}$: i^{th} training example

$z^{[l]}$: l^{th} layer, z value

$X^{\{t\}}$: t^{th} mini batch train examples

mini-batch gradient descent:

for $t = 1, \ldots, 5000$ {

in this for loop implement 1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$ } vectorized!

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

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

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

Backprop to compute gradient wrt $J^{\{t\}}$ (using $X^{\{t\}}, Y^{\{t\}}$)

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

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

}

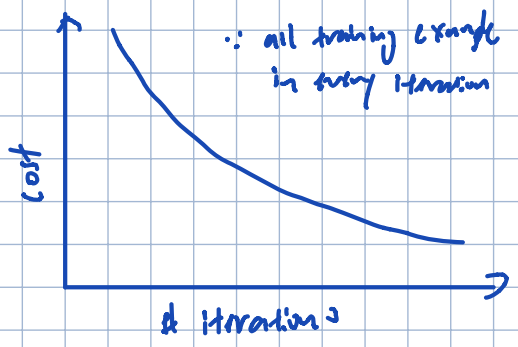
"1 epoch" of training

↳ repeat for multiple pass of the whole batch

understanding mini-batch gradient descent

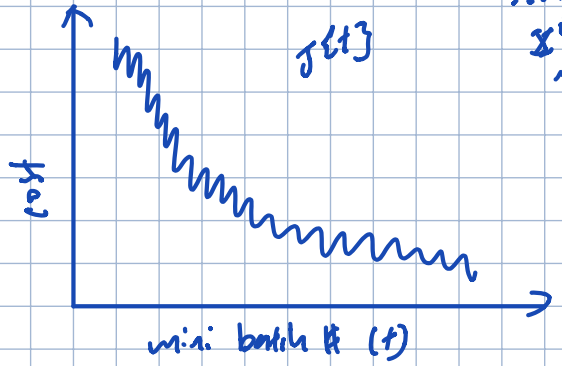
Batch gradient descent

∴ all training example in every iteration



mini-batch gradient descent

∴ every iteration $X^{\{t\}}, Y^{\{t\}}$ ↑ $J^{\{t\}}$



each mini-batch might easy/hard to train → low/high $J^{\{t\}}$

Choosing the size of mini-batch

If mini-batch size = m

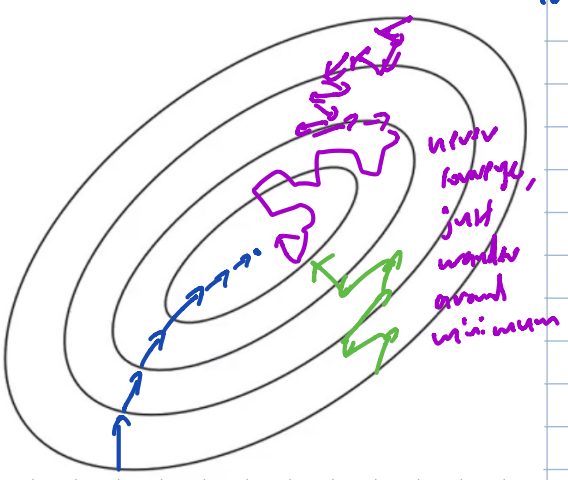
↳ batch gradient descent ⟶ just 1 mini-batch $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

If mini-batch size = 1

↳ stochastic gradient descent

i.e. every example is its own mini-batch

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



In practice: mini-batch size will be somewhere in between 1 and m

Batch gradient descent (mini batch size = m)

↳ too long per iteration

stochastic gradient descent (mini batch size = 1)

↳ lose speed up from vectorization

in-between: Faster Learning

↳ get a lot of vectorization

↳ make progress w/o entire training set

Some guidelines

⟶ If we have a small training set, just use batch gradient descent (e.g. $m \leq 2000$)

⟶ Typical mini batch size: 64, 128, 256, 512 (power of 2, might run faster)

→ make sure all of $X^{\{t\}}, Y^{\{t\}}$ fits in CPU/GPU memory

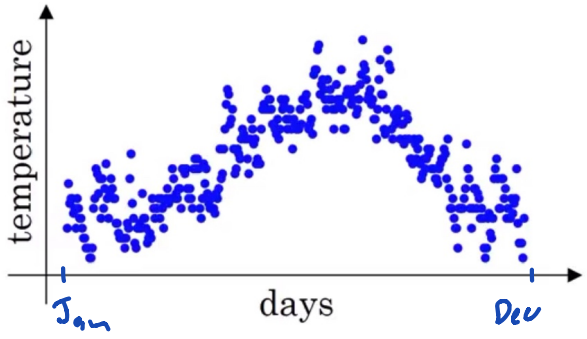
→ mini batch size is another hyper parameter

Exponentially weighted averages

There are a few algorithms that are frequently used, and faster than gradient descent. Need to know exponentially (moving) weighted averages.

Temperature in London

$\theta_1 = 40°F$ 4°C 1 Jan
$\theta_2 = 49°F$ 9°C :
$\theta_3 = 45°F$:
⋮
$\theta_{180} = 60°F$ 15°C
$\theta_{181} = 56°F$:
⋮



Moving average

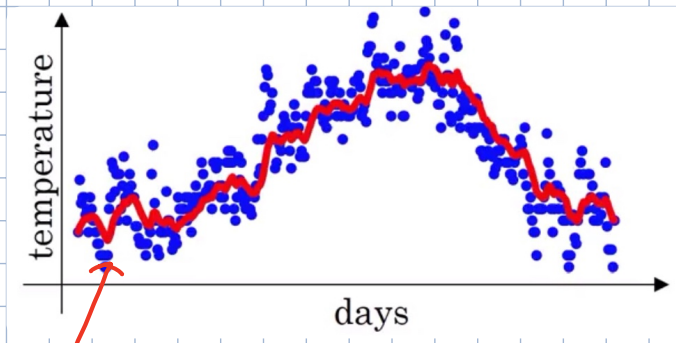
first initialise $V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

$V_3 = 0.9 V_2 + 0.1 \theta_3$

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



exponentially weighted average

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

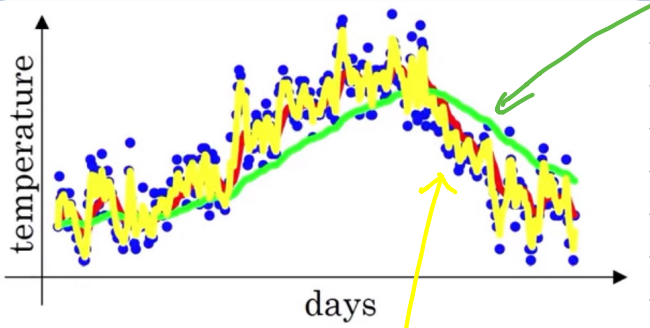
we can think V_t as approximately average over

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

⇒ if $\beta = 0.9$ ⇒ $\approx$ average over 10 days of temperature

if $\beta = 0.98 \Rightarrow \frac{1}{1-0.98} \approx 50$

↳ average over last 50 days of temperature



values like 0.98 gives a lot of weight on previous value

↳ more latency

↳ adapts more slowly

$\beta = 0.5 \Rightarrow \frac{1}{1-0.5} = 2$

↳ last 2 days average

↳ more susceptible to noise

↳ but adapt quickly

β is another hyperparameter.

Understanding exponentially weighted averages

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

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

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

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

⋮

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

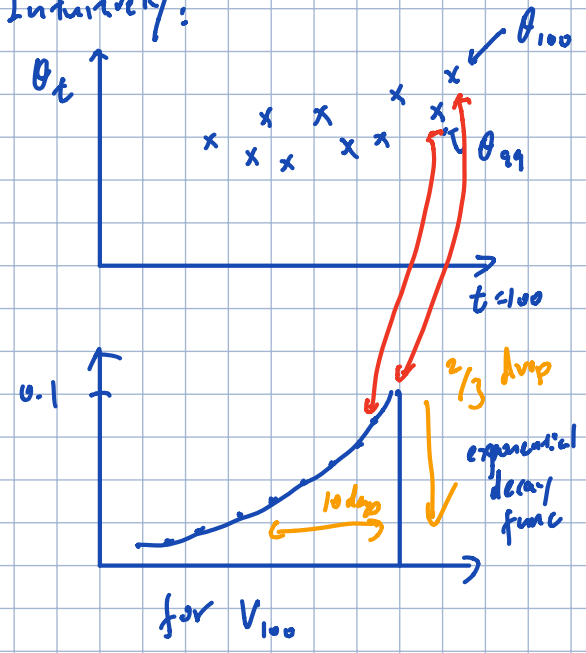
$= 0.1\theta_{100} + 0.9 (0.1\theta_{99} + 0.9 V_{98})$

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

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

All coef adds up to (or close to) 1

Intuitively:



So it is taking element wise product between these 2 function and sum it up

In this case

$\because 0.1 \times (0.9)^n$ $\quad 0.9^{10} \approx 0.35 \approx 1/e$

more generally:

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

* it takes about 10 days for the height of decay from decay to 1/3

if $\beta = 0.98$, then $0.98^{50} \approx 1/e$ $\Rightarrow$ takes about 50 days

Implementation of exponentially weighted averages

$V_0 = 0$

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

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

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

...

$V_\theta = 0$

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

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

...

equivalently

$V_\theta = 0$

Repeat {

Get next θ_t

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

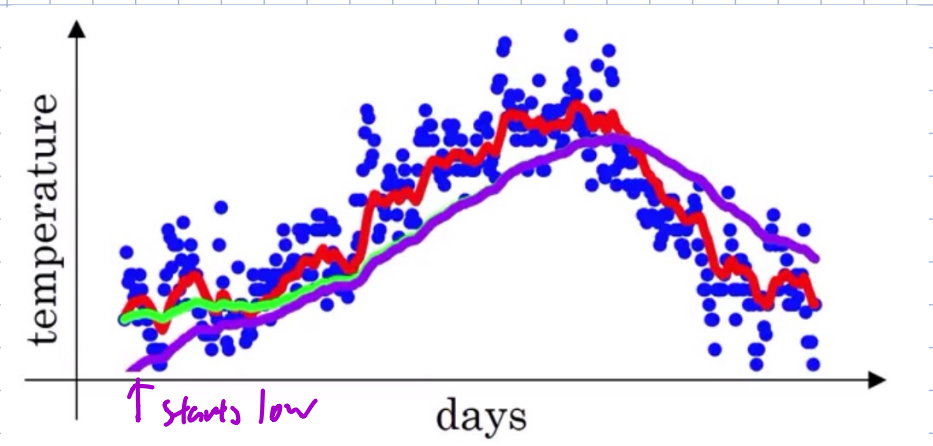
}

Advantage: just 1 variable to track

if moving window, might need to store the last 10 or 50 values

Bias correction in exponentially weighted averages

Can be used to compute the averages more accurately.



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

we will get the purple line, instead of the green line (for $\beta = 0.9$ ish)

When implementing a moving average, we initialise it

$V_0 = 0$ $\quad \therefore v_0 = 0$

then $V_1 = \cancel{0.9 V_0} + 0.02\,\theta_1$ ← low value at start

Hence, to make this averaging more accurate, esp for the initial phase

use $\dfrac{V_t}{1-\beta^t}$ instead of V_t

i.e. if $t=2$, $\quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

so estimate on day 2 $= V_2 / 0.0396 = \dfrac{0.98 V_1 + 0.02\theta_2}{0.0396}$

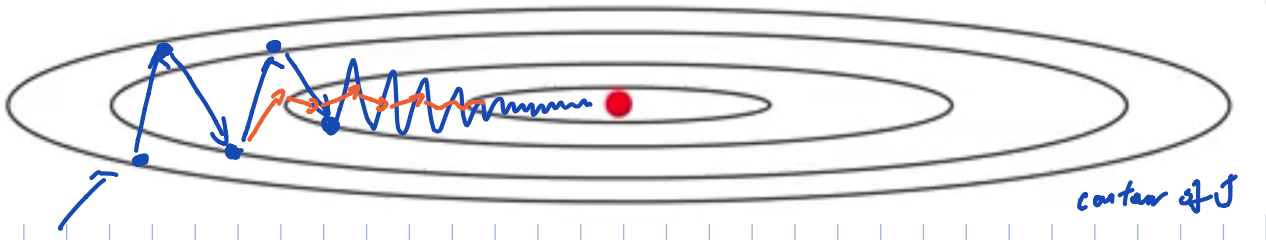
$$= \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

as $t \to \infty$, $\beta^t \to 0$

Gradient Descent with Momentum

Almost always works faster than the standard gradient descent algorithm

Basic idea: compute an exponentially weighted average of our gradients, then use that gradient to update our weights instead



standard gradient descent: oscillation prevents us using a large learning rate we might want

↕ learn slower

⟷ learn faster

with momentum:

on iteration t:

compute the usual dW, db on current mini-batch

friction ↓ velocity

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

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

bowl-shape J

} provides acceleration

$W := W - \alpha V_{dW}$

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

} smooths out the raw ponceros that is ↓ to descent direction

Implementation details

$V_{dW} = 0$ same dimension as dW, $V_{db} = 0$ same dimension as db and b

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

no bias correction needed in most case

Hyperparameters: α, β $\qquad \beta = 0.9$

↳ e.g. 0.9

Average over last 10 iterations' gradient

RMS prop

Also can be used to speed up gradient descent.

On iteration t:

compute the usual dw, db on current mini-batch

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

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

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

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

make some dimensions up (if $S_{dw}, S_{db} \approx 0$)

β_2 for RMS prop parameter

β_1 for momentum

very similar effect as momentum $\rightarrow$ can use a larger learning rate α

Adam Optimisation Algorithm (Adaptive Momentum Estimation)

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

On iteration t:

compute dw, db using current mini-batch

momentum exponentially weighted average $\rightarrow$ $V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$; $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

RMS prop $\rightarrow$ $S_{db} = \beta_2 S_{dw} + (1-\beta_2) dw^2$; $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$

bias correction $\rightarrow$

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

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

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

Hyperparameters choice of Adam:

α: needs to be tuned

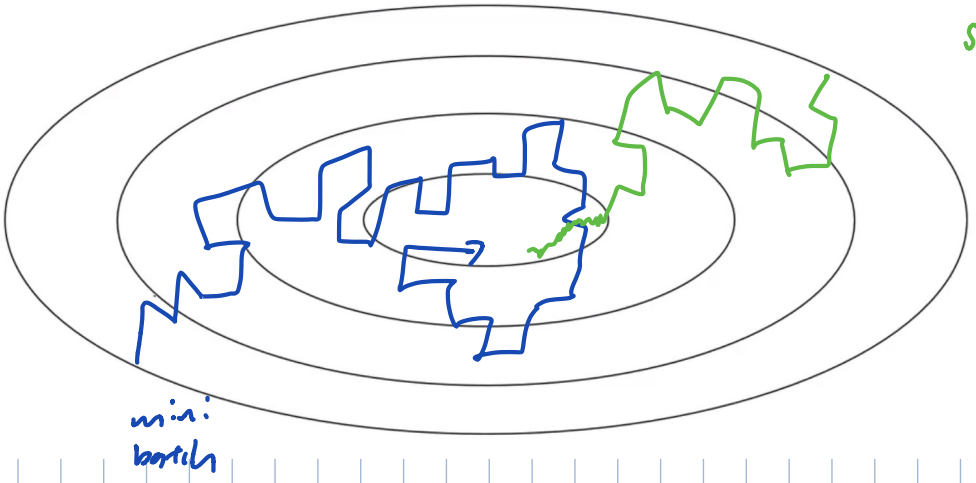
β_1: commonly use 0.9 — (dw) moving average

β_2: 0.999 — (dw^2) moving average

ε: 10^{-8}

Learning Rate Decay

slowly reduce learning rate over time → might speed up learning



slowly reduce α
↳ oscillate tighter near minimum

1 epoch = 1 pass through the data

→ epoch 1

→ epoch 2

set $\alpha = \frac{1}{1 + \text{decay\_rate} \times \text{epoch\_num}} \alpha_0$ ← initial learning rate

e.g. $\alpha_0 = 0.2$, decay_rate = 1

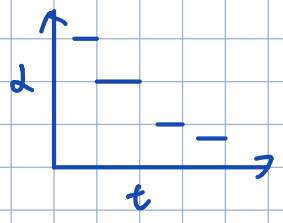
Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	⋮

Other learning rate decay method

$\alpha = 0.95^{\text{epoch\_num}} \cdot \alpha_0$ exponentially decay

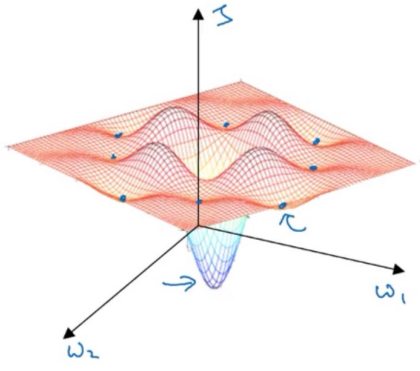
$\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \cdot \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$

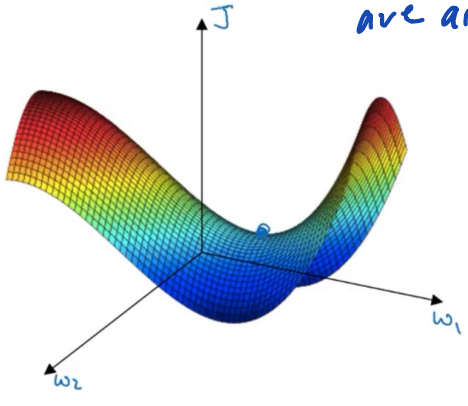
or discrete decay



The problem of local optima

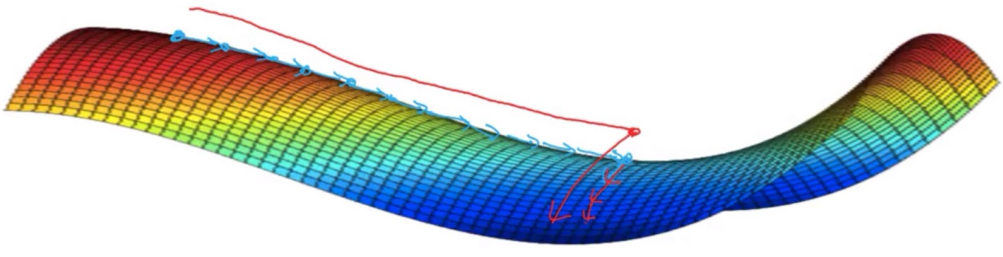
Local optima in neural networks





many "local optima" in NN are actually saddle points
i.e. if 20,000 dimensions then for it to be local minima all 20,000 directions have to be ∪

Problem of plateaus → can really



Week 3:

Hyperparameter tuning, Batch Normalisation and Programming Frameworks

Tuning Process

Some guidelines to systematically organise hyperparameter tuning process.

e.g. α → 1st input

β ~ 0.9 — 2nd input

$\beta_1, \beta_2, \varepsilon$ (Adam) — 3rd input

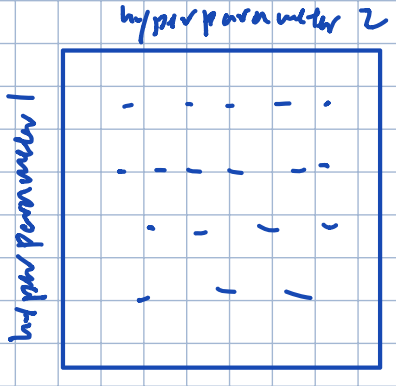
(#) layers

(#) hidden units for different layers

learning rate decay

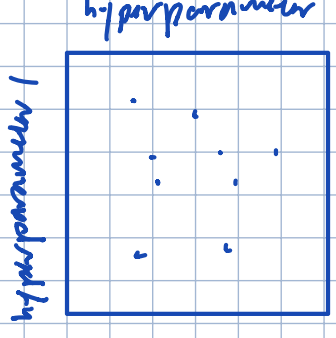
mini batch size

Grid Search



works ok when # hyperparameter was relatively small

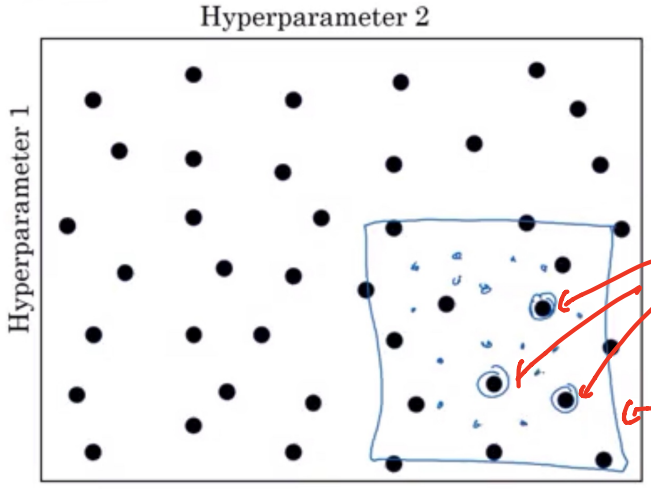
Random Search (for deep learning)



works better because not all hyperparameter has the same importance.

Another common practice is to use coarse to fine sampling scheme

Coarse to fine



→ these points works well
↳ zoom in
↳ sample more densely

Using an appropriate scale to pick hyperparameters

Sampling hyperparameters at random ≠ sampling uniformly at random over the range of valid values. → it is important to pick the right scale.

e.g. picking hyperparameters at random

say we try to choose # hidden units, $n^{[l]}$, for layer l

we think a good range of value is 50 ↔ 100:

$n^{[l]} = 50, \ldots, 100$

examples of uniform random sampling are reasonable but this is not true for all hyperparameter

or # layers L: 2, 3, 4, 5

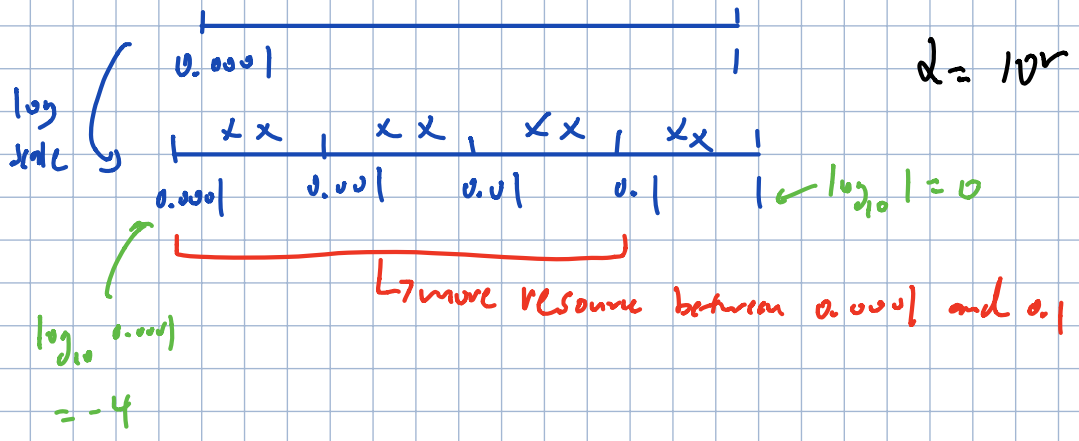
Say we searching for learning rate, α

$\alpha = 0.0001, \ldots, 1$

$r \in [-4, 0]$

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

$\alpha = 10^r$



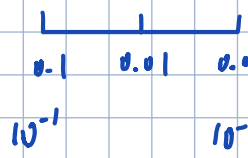
Tricky case: sampling the hyperparameter (β) for exponentially weighted averages.

$\beta = 0.9, \ldots, 0.999$

average last 10 values

average last 1,000 values

Can think of $1 - \beta = 0.1, \ldots, 0.001$



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

$\hookrightarrow 1 - \beta = 10^r \rightarrow \beta = 1 - 10^r$

reason:

β: 0.9 $\longrightarrow$ 0.9005

Hardly any change in results

0.999 $\longrightarrow$ 0.9995

huge impact

$\left(\frac{1}{1-\beta}\right)$

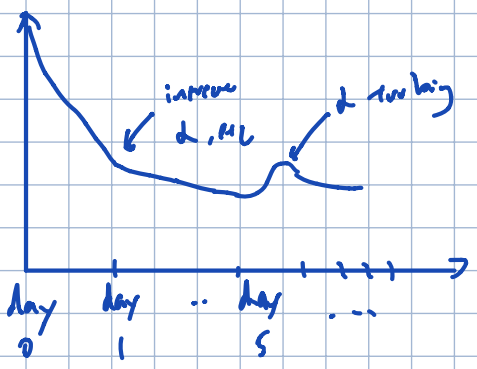
last 1000 example

last 2000 example

Hyperparameters tuning in practice: Pandas vs Caviar

① Babysit 1 model

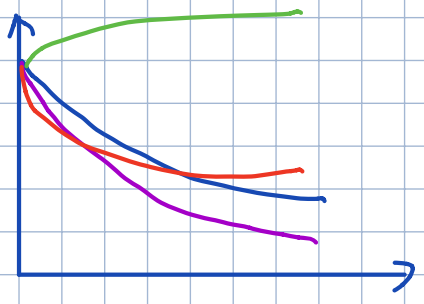
large dataset, but limited computational resources



watching performance and patiently nudging learning rate up or down

"Panda"

② Train many model in parallel

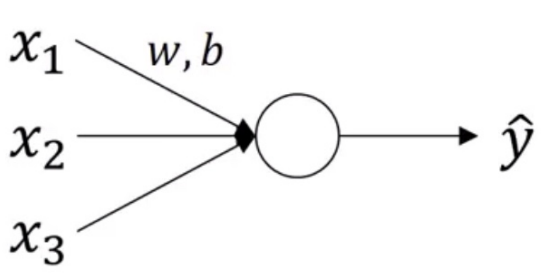


"Caviar"

Batch Normalisation

Normalising activations in a network

Makes hyperparameter search problem much easier, and makes NN much more robust



when training a model (e.g. logistic Regression), normalising the input features can speed up learning

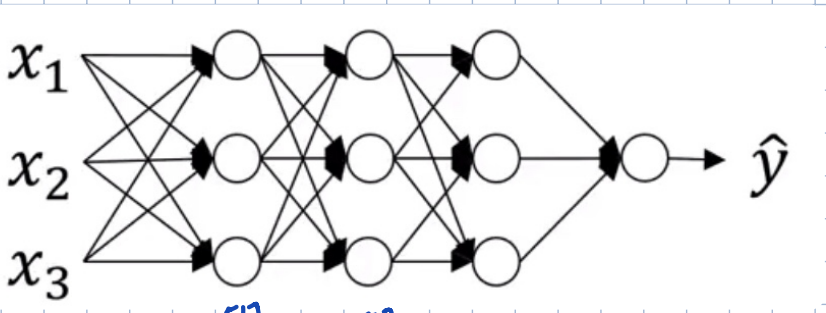
recall: $\mu = \frac{1}{m}\sum_i x^{(i)}$

$X = X - \mu$

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

$X = X/\sigma^2$

How about a deeper model?



not only has input features, but also activations $a^{[l]}$...

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

if we want train parameters $w^{[2]}$, $b^{[2]}$,

↳ normalise $a^{[2]}$ to make the training of $w^{[2]}$, $b^{[2]}$ more efficient

For any hidden layer, can we normalise, the value of $a^{[l]}$ so as to train $w^{[l+1]}$, $b^{[l+1]}$ fast?

↳ Batch Normalisation!

↳ we will be normalising $z^{[l]}$ (instead of $a^{[l]}$)

Implement Batch Norm

Given some intermediate values in NN: $z^{(1)}, \dots, z^{(m)}$ for some layer l ($z^{[l](i)}$)

Compute $\mu = \frac{1}{m}\sum_i z^{(i)}$ (for some layer l, omitting $[l]$ notation)

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

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

every component of z: mean = 0, var = 1

we don't want the hidden unit always have mean 0 and var 1

↳ have different distribution

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

γ, β: learnable parameters from model

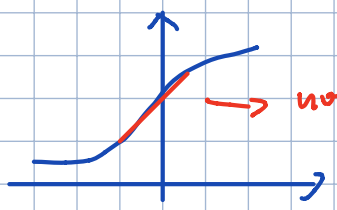
from e.g. gradient descent we will update γ and β just as w and b

γ and β allow us to set the mean of $\tilde{z}$ to any value:

if $\gamma = \sqrt{\sigma^2 + \varepsilon}$, $\beta = \mu$ } invert back to $z^{(i)}$ (from $z^{(i)}_{norm}$)

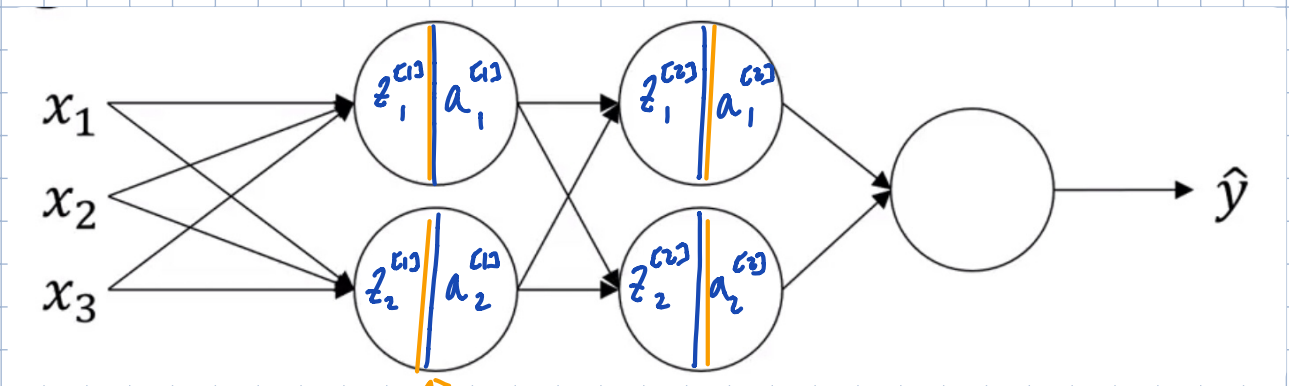
i.e. $\tilde{z}^{(i)} = z^{(i)}$

we will use $\tilde{z}^{[l](i)}$, instead of $z^{[l](i)}$, for the later network



→ not all value cluster around mean=0, where activation function are in the linear regime.

Fitting Batch Norm into a NN



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

Parameters of our network: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}$

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

↑ nothing to do with momentum hyperparameter β

RMSprop

Adam

We can still use gradient descent, but need

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

In practice, batch norm is usually applied with <u>mini-batch</u> of our training set:

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

first mini batch

(just this mini-batch)

then: $X^{\{2\}} \rightarrow z^{[1]} \xrightarrow[\text{BN}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow \dots$

mean and var

since we the data from $X^{\{2\}}$

$\vdots$

Parameters $W^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$

↳ $z^{[l]} = W^{[l]} a^{[l-1]} \boxed{+ b^{[l]}}$ → gets cancel out during Batch Norm

↳ any constant added here doesn't change anything during Batch Norm (cancel out during mean subtraction)

↳ hence

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

↳ $z^{[l]}_{norm}$

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

for 1 example

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

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

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

$\gamma^{[l]} : (n^{[l]}, 1)$

Implementing Gradient Descent with Batch Norm (mini-batch)

for $t = 1 \ldots$ num_mini batch

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

update

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

} gradient descent

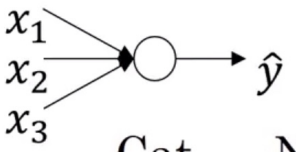
Also works with GD with momentum, RMSprop, Adam

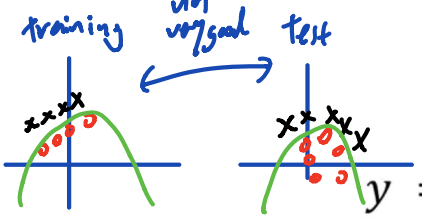
Why does Batch Norm work?

First reason: similar to normalise input in logistic regression for GD.

Just that this is applied to further hidden nodes

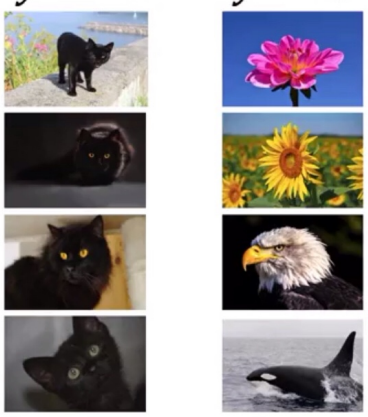
Second reason: makes weights deeper in NN more robust to changes to weights in earlier layers of NN





same func might work but algorithm would learn this by just looking at train data

Cat $y = 1$ | Non-Cat $y = 0$



$y = 1$ | $y = 0$

covariate shift

$X \rightarrow y$

if change, might need to retrain

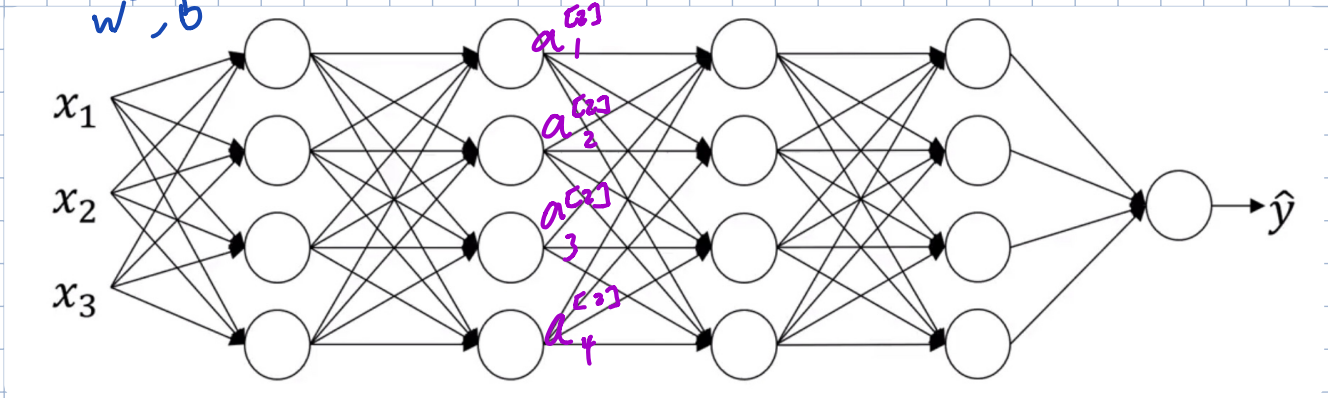
train on black cats

don't do very well

predict on color cats

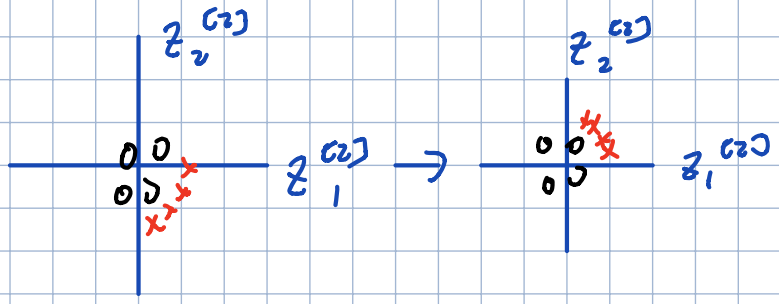
This is a problem for NN : let's look at 3rd layer below

$w^{[2]}, b^{[2]}$ $w^{[3]}, b^{[3]}$ $w^{[4]}, b^{[4]}$



3rd layer job: take $a^{[2]} \xrightarrow{\text{map}} \hat{y}$. But $a^{[2]}$ are from layer 2, which changes from 1st layer $\Rightarrow$ i.e. $a^{[2]}$ changes all the time $\rightarrow$ covariate shift

Batch norm : reduces the amount that the distribution of these hidden unit values shifts around.



even if exact value of $z_1^{[2]}$, $z_2^{[2]}$ changes, its mean / var will at least stay the same

allows input to the nodes more stable

OR

It weakens the coupling between early layers parameters has to do with later layers, so it allows each layer to learn by itself.

third reason: slight regularisation effect

Batch Norm as regularization

$\{x^{\{t\}}, z^{[1]}\}$

- 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.

$z^{[1]} \rightarrow \tilde{z}^{[1]}$

increase mini batch size

$\rightarrow$ reduce noise

$\rightarrow$ reduce regularisation effect

Batch Norm at test time

At test time, may not have mini batch, maybe one at a time. we have to adapt our NN to do that.

Usually Batch Norm has this issue, as it changes forward prop (i.e. prediction)

Batch Norm at test time

during training

(computed using) mini-batch

single mini-batch

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

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

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

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

In test time, only 1 example at a time to predict.

In test time, come up with some separate estimate of μ and σ^2

↳ estimate using exponentially weighted average (across mini batch)

e.g. for layer l

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

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

$\sigma^{2\{1\}[l]}$ $\sigma^{2\{2\}[l]}$ $\sigma^{2\{3\}[l]}$ → σ^2

than when making a prediction:

$$z_{\text{norm}} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z} = \gamma z_{\text{norm}} + \beta \Rightarrow \hat{y}$$

from NN

OR, you can run the entire training set, then get average/final μ, σ^2

Multi class classification

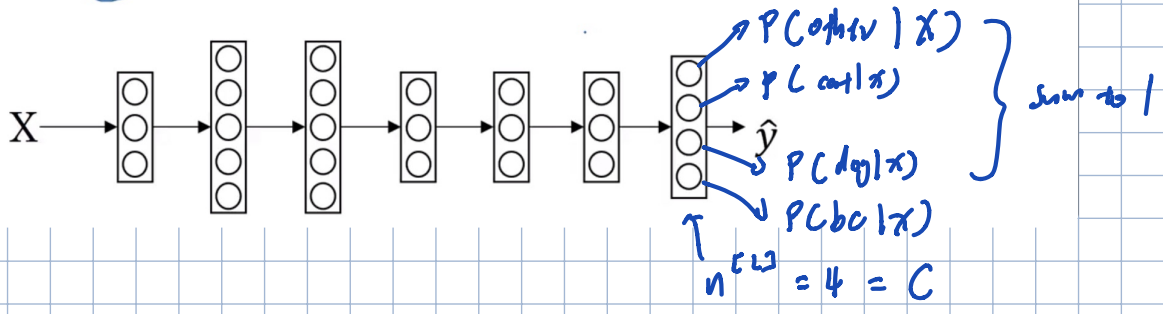
Softmax Regression

It is a generalized form of logistic regression

Recognizing cats, dogs, and baby chicks, other

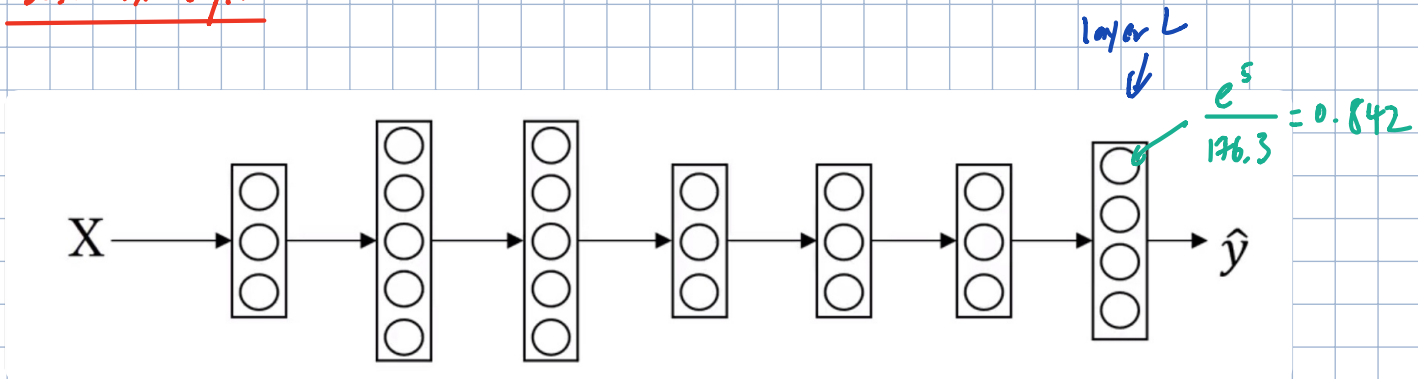
3 1 2 0 3 2 0 1

C = #classes = 4 $(0, \dots, 3)$



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

Softmax layer



At the final layer L:

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

Activation function:

temporary variable $\rightarrow$ $t = e^{(z^{[L]})}$ ← element-wise. Here for 1 example $z^{[L]} = (4,1)$, so t is also (4,1)

normalize sum=1 $\rightarrow$ $a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i}$ also (4,1)

each element

$$\hookrightarrow a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

e.g. $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \Rightarrow t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$

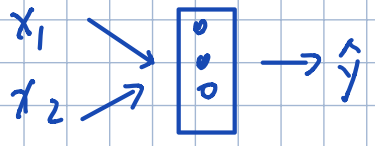
$\hookrightarrow \sum_{i=1}^{4} t_i = 176.3 \Rightarrow a^{[L]} = t / 176.3 = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$

Softmax activation function:

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

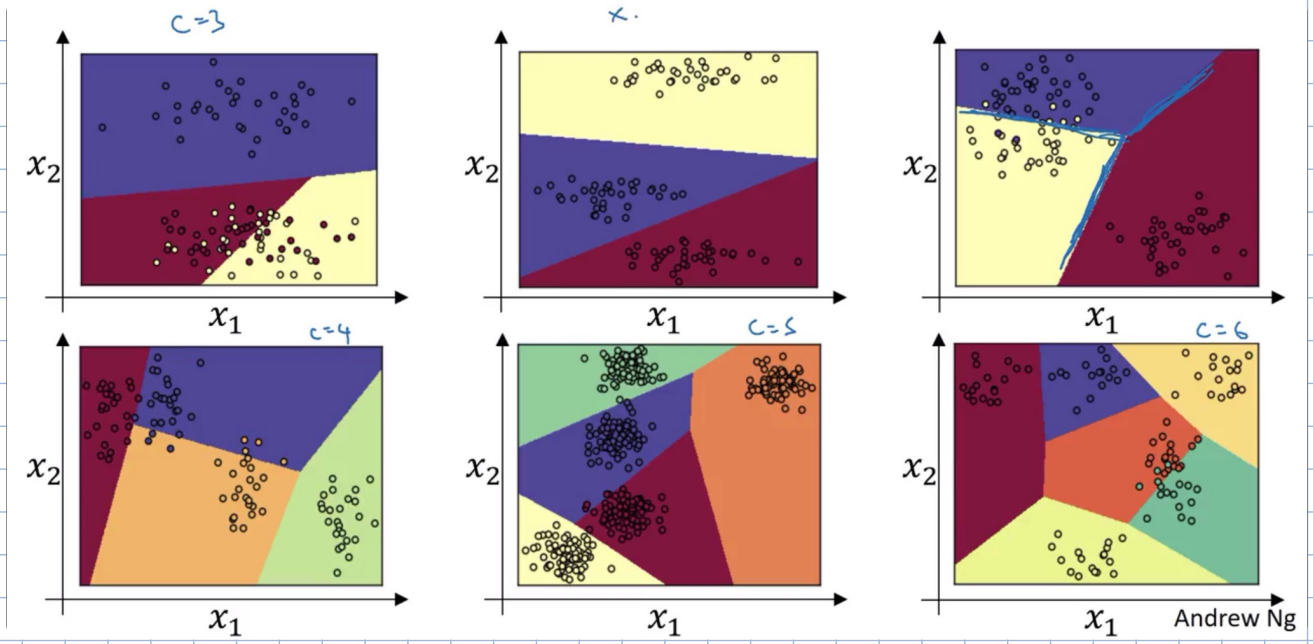
$a^{[L]}$: $(4,1)$ $\quad z^{[L]}$: $(4,1)$

A NN with no hidden layer:



output layer

$$z^{[1]} = W^{[1]} x + b^{[1]}$$
$$a^{[1]} = \hat{y} = g(z^{[1]})$$



Training a softmax classifier

(4,1)

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

$C = 4$

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

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

softmax activation function generalises the logistic activation function to C classes (rather than 2 classes) (so it is still a linear classifier)

If $C=2$, softmax reduces to logistic regression

i.e. $a^{[L]} = \begin{bmatrix} 0.842 \\ \cancel{0.158} \end{bmatrix}$ ← redundancy → 1 value ⟺ logistic regression

How we train a NN with a softmax output layer?

e.g. true label $y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ cat prediction $\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} = a^{[L]}$ $C=4$

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

here $y_1 = y_3 = y_4 = 0$, $y_2 = 1$ ⟶ $-y_2 \log \hat{y}_2 = -\log \hat{y}_2$

i.e. it algorithm try to make $\mathcal{L}(\hat{y}, y)$ small (through gradient descent)

↳ make $-\log \hat{y}_2$ small

↳ make $\hat{y}_2$ big

this loss function look at true label, then try to make the corresponding prediction probability big.

cost function J on the entire training set

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

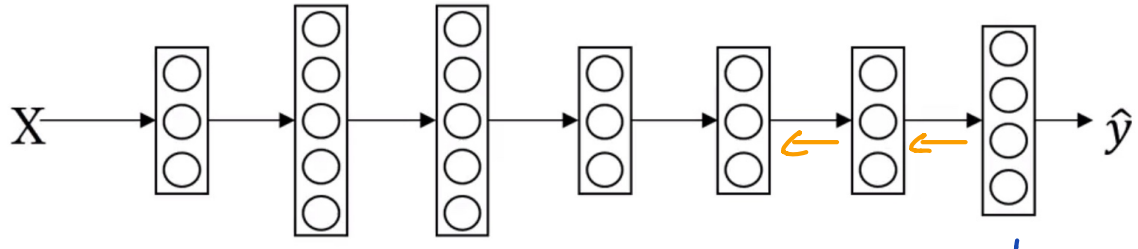
vectorized:

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

$(4, m)$

$$\hat{Y} = [\hat{y}^{(1)}\ \hat{y}^{(2)} \dots \hat{y}^{(m)}] = \begin{bmatrix} 0.3 & \dots \\ 0.2 & \\ 0.1 & \\ 0.6 & \end{bmatrix}$$

Gradient descent with softmax



$z^{[L]} \rightarrow a^{[L]} = \hat{y}$

$(4,1)$

$\mathcal{L}(\hat{y}, y)$

Back prop (with softmax at layer L):

$$dz^{[L]} = \hat{y} - y$$, then starts the back prop

$(4,1)$

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

Deep Learning frameworks

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)

TensorFlow

Motivation

e.g. minimise $J(w) = w^2 - 10w + 25$ $= (w-5)^2$ → $w=5$ is minimum J

```
In [1]: import numpy as np
        import tensorflow as tf

In [5]: w = tf.Variable(0,dtype=tf.float32)
        #cost = tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        cost = w**2 - 10*w + 25
        train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

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

        0.0

In [6]: session.run(train)
        print(session.run(w))

        0.1

In [7]: for i in range(1000):
            session.run(train)
        print(session.run(w))

        4.99999
```

w has to be declared as tf.variable

Starts a session

initialise global variable

→ run 1 step of gradient descent

Code example

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
     session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```