Train/Dev/Test Sets.

- Applied ML is iterative progress:

  You don't know the value of # layers, # hidden unit ...

  You just try ⇒ experiment → idea → try → ...

Dataset makes the cycle quickly.

if you have: 100/1000/10000 Data. 70/30. OR 60/20/20.

if you have: 1,000,000 Data. Maybe. 10000 for Dev and Test.

Like 98% train. 1% dev. 1% test.

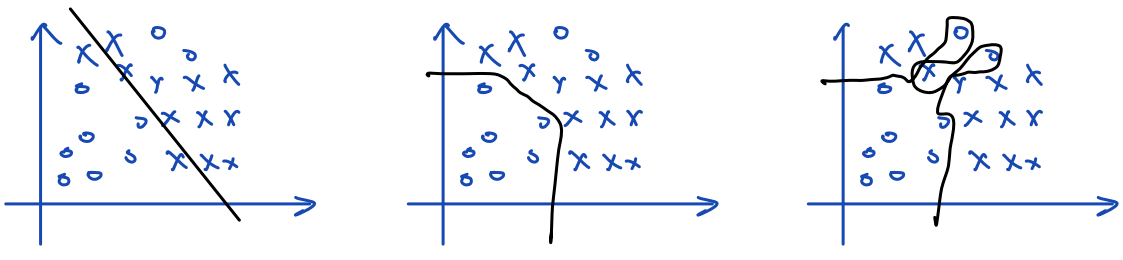
- Mismatched train/test distribution.

E.g { Training set from webpage.
Dev/Test sets comes from your phone.

★ Rule of Thumb. Dev and test comes from the same distribution.

It is OK if you only have dev.

Bias/Variance.



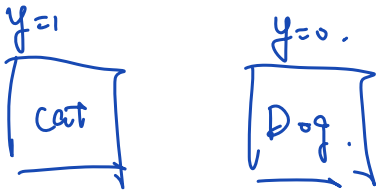
High Bias
Underfitting

Just Right

High Variance.
Overfitting.

E.G:

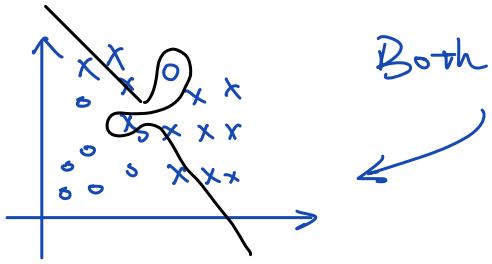
Cat classification:



Assumption: Train and Dev comes from the same distribution

Trainset Error.	1%	15%	15%	0.5%
Devset Error:	11%	16%	30%.	1%
	high variance (Overfitting)	high bias. (underfitting).	high bias & high variance.	low bias & low variance.

Crucial point: Optal (Bayes) Error: e.g. human classifier error.



Both high bias & variance.

Basic "Recipe" for machine learning.

① High Bias? (Train data problem) → Yes. →
- Bigger Network.
- Train longer.

② High Variance? (Dev set problem). → Yes. →
- More data.
- Regularization.

↓

Fixed ①②. Done.

* High bias problem. "More data" won't help.

⇓

* Bias Variance Tradeoff. its not accurate. We can drive down one thing without hurting another for just deep learning.

Regularization

e.g. Logistic regression.

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

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

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

L1 regularization. $\frac{\lambda}{m}\sum_{i=1}^{n_x}|w| \rightarrow$ w will be sparse. because some w_i will be 0.

λ: regularization param. determined by cross-validation.

↳ is a hyperparam need to be Tuned.

- Neural Network.

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

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2 \quad \text{"Frobenius norm".}$$

$dw^{[l]}$ from backprop. $\frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$.

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

Adding L2 regularization. $dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$

We call this "weight decay". $w^{[l]} = w^{[l]} - \alpha\left[(\text{from back}) + \frac{\lambda}{m} w^{[l]}\right]$

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

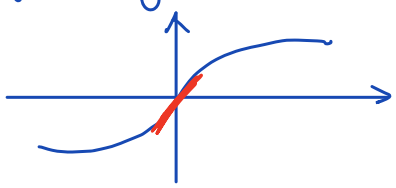
↓

Now you are multiply $w^{[l]}$ by $(1 - \frac{\alpha\lambda}{m})$

Why regularization prevent overfitting

- Intuition: when $\lambda \rightarrow \infty$. we are setting some $w \rightarrow 0$. So the network actually become simpler. "Smaller Network".

- e.g. Say we have a $\tanh(z)$. activation function.



$\Rightarrow \lambda \uparrow$. $w^{[e]} \downarrow$. $z^{[e]} = w^{[e]} a^{[e-1]} + b^{[e]}$. $\downarrow$

So z will concentrate with the linear area.

Which we kind simplifies the network.

Dropout Regularization.

We we go through each layer and each node. We have a probability drop some nodes.

Remove all the connections to those nodes.

Do Backprop.

Next iteration (New example input).

Still using the probability to drop some of the nodes.

- Implementing Dropout.

Illustrate with layers $l=3$. keep-prob = 0.8. → 0.2 prob to drop.

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

↓ Boolean matrix. 0 or 1.

3: represents the 3rd layer.

a3 = np.multiply(a3, d3).

a3 /= keep-prob.

↓

explanation: Say you have 50 units. On average, 10 units will get shut dow

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

↑ value will be reduced by 20%.

So dividing 80%, we inverted the dropout effect.

- Making prediction at test time.

$a^{[0]} = x$ input.

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

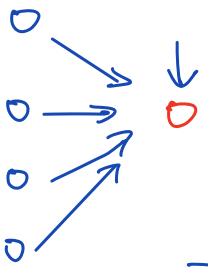
$a^{[i]} = g^{[i]}(z^{[i]})$.

⋮

In test time. we don't want the dropout layer.

Understanding Dropout.

Intuition: Can't rely on any one feature. so have to spread out weight.



Since some blue node might go away. so the red node can't rely on single blue node. So it have to spread out its weight. "Shrink weight" for each blue node.

This looks like L_2 regularization.

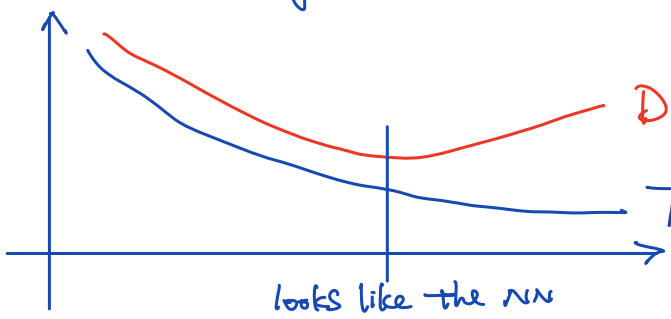
Remember. no overfitting, no dropout.

Other regularization technique.

- Data augmentation.

e.g. flipping images. rotating pics. zoom in/out pics.
rotation, distortion for digits images.

- Early stopping.



Stop half way with mid-size weights $\|w\|_F^2$.

Orthogonalization

By doing early-stopping. You cannot do both-thing independently.

Like doing both things with one tool.

* for DNN. Optimizing J.
  - Gradient descent.

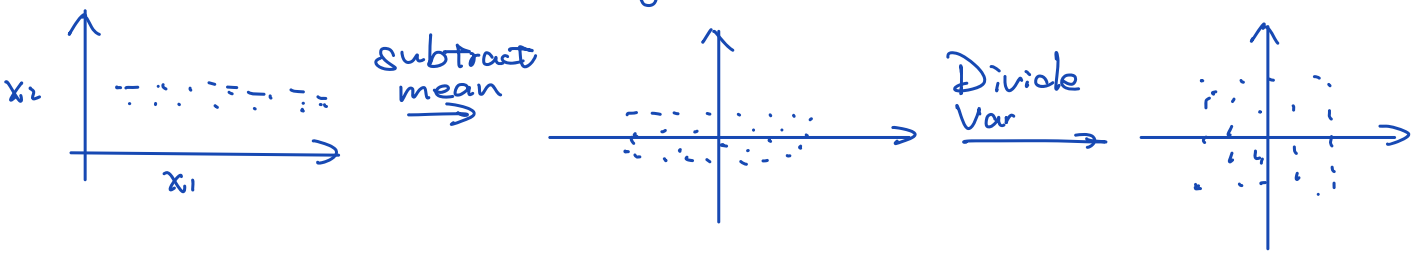
Not overfit.

— Regularization.

Normalizing Inputs.

Normalizing traing Set. Subtracting mean ⇒ Zero mean.

$\sigma^2 = \frac{1}{m}\sum x^{(i)2}$. Divided by σ^2. Variance bothe equal to 1.

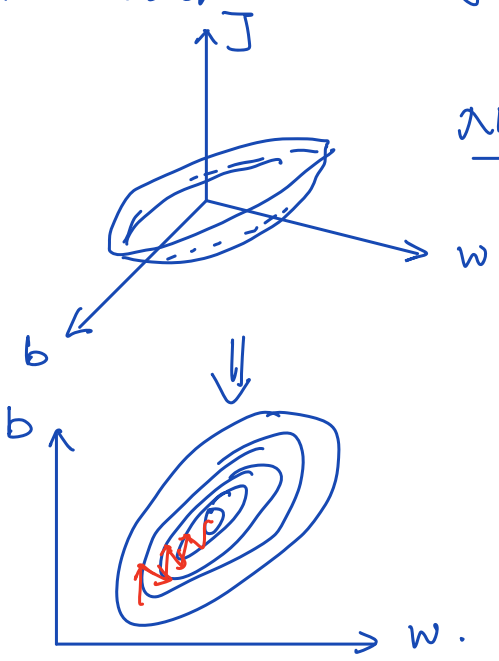


Use same μ and σ^2 to rescale the test set!

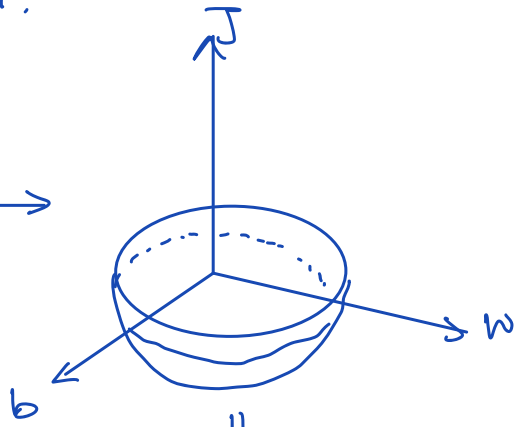
- Why normalize features?

Rough intuition.

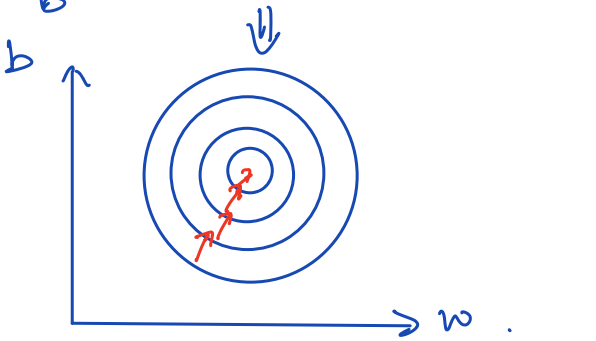
Unnormalized:



Normalize



The gradient might be going squizy.

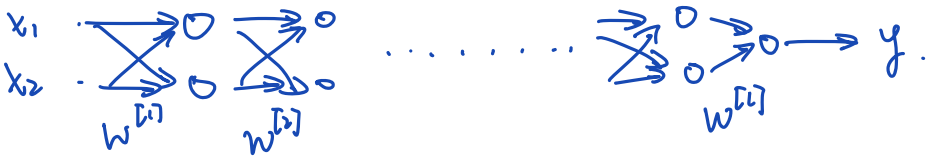


The gradient can go straight

Your features are in same scale.

Vanishing / Exploding gradient.

Suppose you are training a really deep NN.



$g(z) = z$ as activation func. $b^{[l]} = 0$.

So $\hat{y} = w^{[l]} w^{[l-1]} w^{[l-2]} \cdots w^{[2]} w^{[1]} x$

$z^{[1]} = w^{[1]} x$.

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

$a^{[2]} = g(z^{[2]}) = g(w^{[2]} w^{[1]})$

Suppose we have. $w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ $\hat{y} = w^{[l]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{l-1} \cdot x$

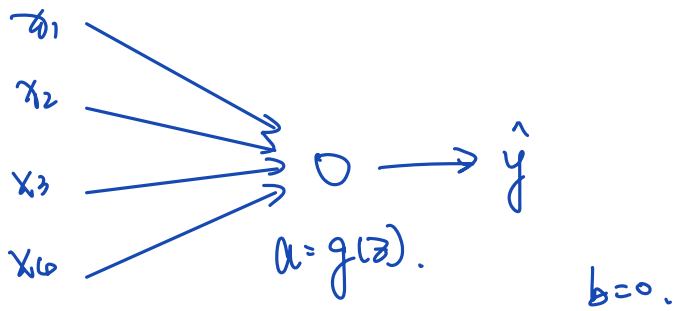
So if $l \to \infty$. we know this will explode. (exponential). $\hat{y}$ really big.

if $w^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$. Similarly. with $l \to \infty$. $\hat{y} \to 0$.

So. with really deep neural network. it will take really long time to train

Weight initialization in a deep network.

Single neuron example:



$b = 0$.

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

We don't want z blow up or too small.

So we want large n and small w_i.

One reasonable thing to do:

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

n: Number of features fed into one neuron.

So we can set: $w^{[l]} = np.random.rand(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

* If use ReLU. $\left(\frac{2}{n^{[l-1]}}\right)$ sometimes perform better.

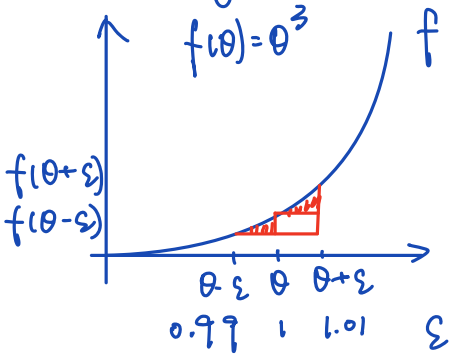
e.g. Xavier initialization. Random uniform distribution between:

$\pm\sqrt{\frac{6}{n^{[i]} + n^{[i+1]}}}$. $n^{[i]}$: connections into the neuron.

$n^{[i+1]}$ connection going out of the neuron.

Numerical Approximation of Gradient.

Checking your derivative computation.



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

$$\frac{(1.01)^3-(0.99)^3}{0.02} = 3.0001.$$

$g(\theta) = 3\theta^2 = 3$. Approximation err: 0.0001.

Gradient Checking. (Don't use in training. Only for debug).

Take $w^{[1]}, b^{[1]}, \cdots, w^{[l]}, b^{[l]}$ and reshape it into big vector θ.

$J(w^{[1]}, b^{[1]}, \cdots, w^{[l]}, b^{[l]}) = J(\theta)$.

Take $dw^{[1]}, db^{[1]}, \cdots$ and reshape it into big vector $d\theta$.

Q: is $d\theta$ the gradient of $J(\theta)$. $J(\theta) = J(\theta_1, \theta_2, \theta_3, \cdots)$.

for each i:

$$d\theta_{approx}^{[i]} = \frac{J(\theta_1, \theta_2, \cdots, \theta_i+\varepsilon, \cdots) - J(\theta_1, \theta_2, \cdots, \theta_i-\varepsilon, \cdots)}{2\varepsilon}. \quad \varepsilon = 10^{-7}$$

We know this should approximately equal to $d\theta^{[i]} = \frac{\partial J}{\partial \theta^{[i]}}$

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ $\approx 10^{-7}$ Great!

$\approx 10^{-5}$

10^{-3} worry.

Remember Regularization

Doesn't work with dropout

Run at random init.

Again after some training

Mini-Batch Gradient Descent. Speed up.

Vectorization allows you to efficiently compute on m examples.

$X = [x^{[1]}, x^{[2]}, \cdots, x^{[m]}]$.

$(n \times m)$

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

$Y = [y^{[1]}, y^{[2]}, \cdots, y^{[m]}]$

$(1 \times m)$

Split examples into "baby" training set. e.g. mini-batches of 1,000 each.

e.g. $X^{\{1\}} = [x^{[1]}, \cdots, x^{[1000]}]$. $X^{\{2\}} = [x^{[1001]}, \cdots, x^{[2000]}]$.

We have up to $X^{\{5000\}}$.

Split y accordingly.

Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$. Dimension of $X^{\{t\}}$: $(n \times 1000)$. $y^{\{t\}}$: $\{1 \times 1000\}$

for $t = 1, \cdots, 5000$. {

1 step of gradient descent using $X^{\{t\}}, y^{\{t\}}$.

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

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

You can vectorize this.

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

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

Update weights: $W^{[l]} = W^{[l]} - \alpha \cdot dW^{[l]}$. $b^{[l]} = b^{[l]} - \alpha \cdot db^{[l]}$.

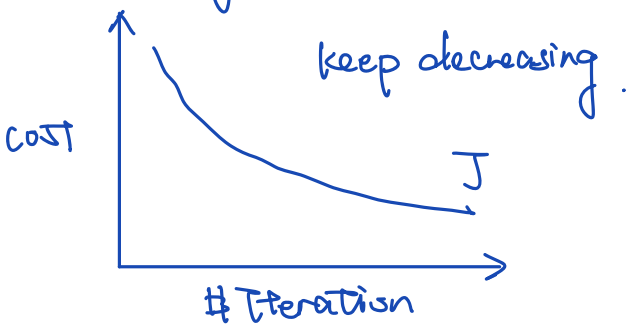
} → This entire for loop finish is 1 epoch of training.

"Single pass through the training set".

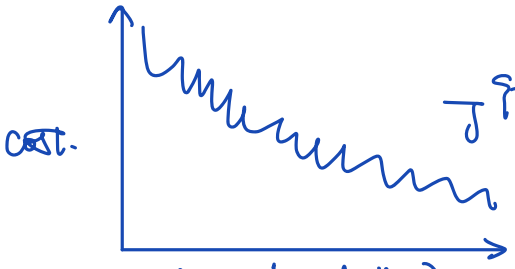
e.g. 100 epochs.: Repeat 100 times for loop.

Understand Mini-batch Gradient Descent.

Batch gradient descent.



Trend: Going down.



Plot $J^{(t)}$ using $x^{\{t\}}$ $y^{\{t\}}$.

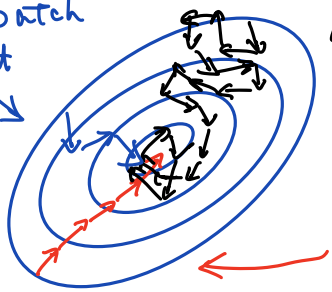
Each time you are throwing a different training

Choosing your size:

① if mini-batch size = m. Batch gradient descent.

② if ... size = 1. Stochastic gradient descent:
Every-example itself is a mini batch.

Mini-batch gradient des.



Stochastic Gradient Des.
Noise may be reduced by decreasing learning rate. Loosing adv. for vectorization.

Batch gradient des.
Processing a huge dataset in every iteration.
Time of each iteration is long.

So we want to take mini-batch size not too large nor too small.

- How to choose batch-size?

if small training set. (e.g. ≤ 2000). Use batch gradient descent.

Typical mini-batch size: 64. 128. 256. 512.

(Reason: the structure of memory is binary. Maybe faster).

Make sure minibatch fit the CPU/GPU memory.

Exponentially weighted averages.

e.g. Temperature in London.

$\theta_1 = 40°F$. $4°C$.

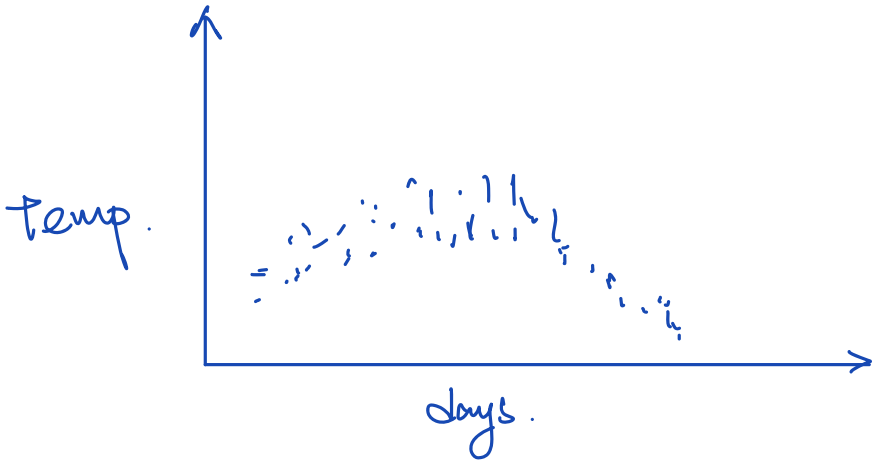
$\theta_2 = 49°F$. $9°C$.

$\theta_3 = 45°F$

$\vdots$

$\theta_{180} = 60°F$.

$\theta_{181} = 56°F$.



$V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

$V_3 = 0.9 V_2 + 0.1 \theta_3$.

$\vdots$ $\vdots$

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

Weighted average:

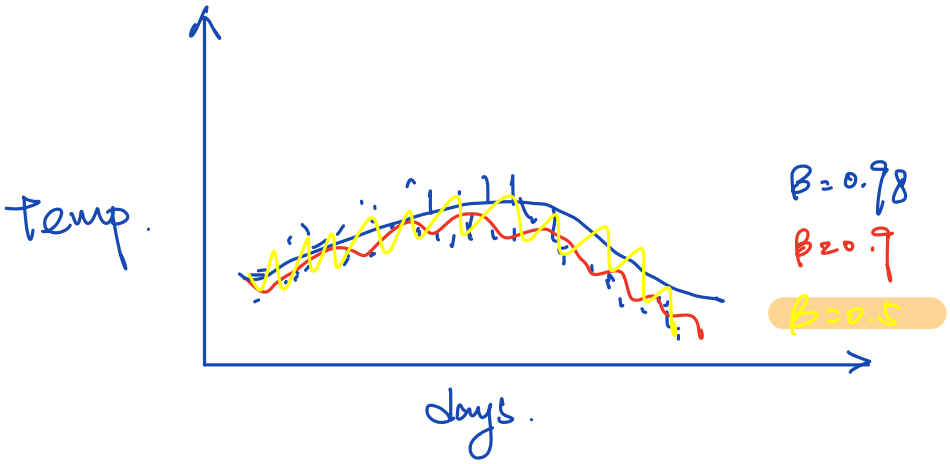
$V_t = \beta V_{t-1} + (1-\beta)\theta_t$.

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

$\beta = 0.9$: $\approx$ 10 day temperature.

$\beta = 0.98$ $\approx$ 50 days temperature.

$\beta = 0.5$ $\approx$ 2 days temperature



Understanding exponentially weighted averages

Exponentially weighted average:

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

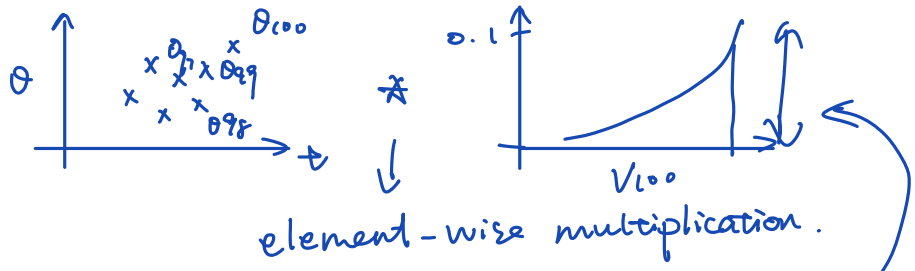
Take an example:

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

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

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

Suppose we have:



Wonder: how many days you are averaging?

Notice $0.9^{10} \approx \frac{1}{e}$. Generally: $(1-\varepsilon)^{\frac{1}{\varepsilon}} = \frac{1}{e}$. when $\varepsilon \to 0$. $\varepsilon = 1-\beta$.

So in here. we take 10 days height of this decrease $\frac{2}{3}$

After 10 days. the weight decrease to $\frac{1}{3}$ approx.

Other e.g. $\varepsilon = 0.2$. $0.98^{50} \approx \frac{1}{e}$. Averaging over 50 days.

Bias correction in exponentially weighted average.

We implement $V_0 = 0$. $V_1 = 0.98 V_0 + 0.02\theta_1 = 0.02\theta_1$.

$V_2 = 0.98 V_1 + 0.02\theta_2$

$= 0.0196\theta_1 + 0.02\theta_2$

This is not a good estimate of the first day temperature.

In general. lower estimate of early days.

Better estimate: $\frac{V_t}{1-\beta^t}$

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

New estimate: $V_2 / 0.0396 = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$.

Remove the bias.

When t goes large. the correction came to 1.

Gradient Descent with Momentum.

GD e.g.

keep ocelating.

But we cannot increase learning rate because it will over shot.

Vertically: slower. Horizontally: faster.

Momentum:

On iteration t:

Compute dW. db on current minibatch.

$V_{dw} = \beta V_{dw} + (1-\beta) dw$. (Similar to $V_\theta = \beta V_\theta + (1-\beta)\theta_t$)

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

(V_{dw}, V_{db}: velocity; dw, db: acceleration.)

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

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

Hyperparam: α. β.

Normally: $\beta = 0.9$. Average across past 10 iteration.

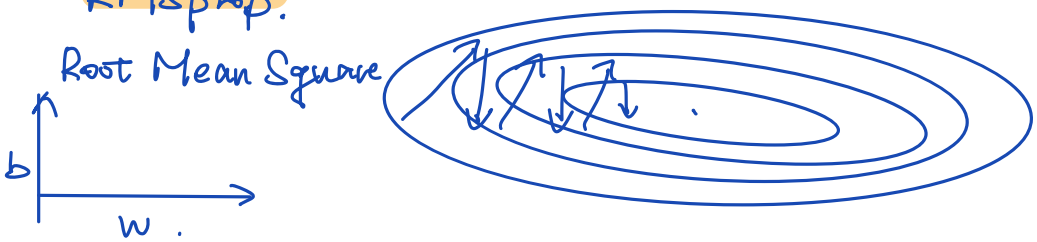
Now say we are doing descent: 

Vertical average will goes to the middle.

Horizontally. we kept going to the right. So we'll go much quicker.

RMSprop.

Root Mean Square



in practical:

some W_i will be vertical

some W_j will be horizontal

Say b: speed of learning vertically.

W: . . . . . horizontally

On iteration t:

Compute dW. db on current mini-batch.

$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$ → elementwise square.

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

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

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

Notice: S_{dw} relative small. and S_{db} is relative large.

So dw divide by small number and db divide by large value.

Note. we don't want S_{db} be zero. So we add some epsilon to bottom.

Adam Optimization Algorithm.

Adam: Adapted moment estimation.

Initialize: $V_{dw} = 0$. $S_{dw} = 0$. $V_{db} = 0$. $S_{db} = 0$.

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.

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

Bias Correction.

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

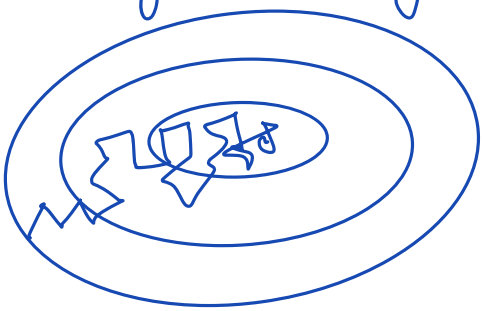
$w = w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon}$ $b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$.

Hyperparam Choice:

α: Need to be tune. (learning rate).

β_1: 0.9. β_2: 0.999. ε: 10^{-8}. (Default value).

Learning Rate Decay.



Say your learning at the end just wander around
Maybe just reduce α.
"Beginning, take large steps. later smaller step"

1 epoch = 1 pass through data.

Set $\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch-num}} \alpha_0$. Decay-rate need to be tuned.

$\alpha_0 = 0.2$. Decay-rate : 1.

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

Other learning rate decay method:

$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0$. - exp. decay.

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

Hyperparam Tuning.

Hyperparameters: ① α : learning rate. ② β : Momentum. β_1. β_2. ε.
0.9 0.999. 10^{-8}.

Barely tuned

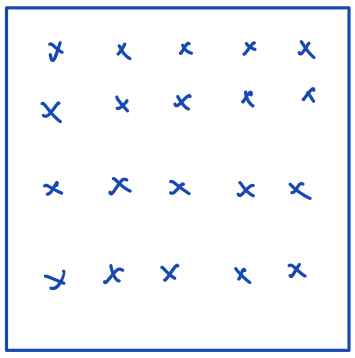
② ① ③ ②
layers. # hidden units. learning rate decay. Mini-batch size.

Try random values: Don't use a grid. Use random sampling.

e.g. Hyperparam ①

Hyper ②

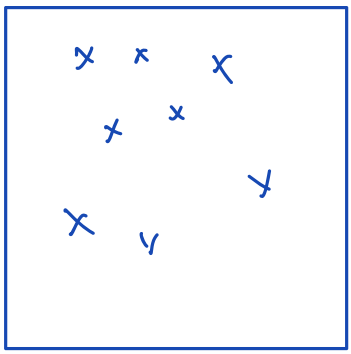
Hyper ③



Grid Sample.

e.g. ②. Hyper ①

Hyper ②



Randomly Sample.

- Coarse to fine.

When sampling. Shrink sample range. Sample more densely.

Use an appropriate scale to pick hyperparams.

Picking HP at random.

$n^{[l]} = 50 \sim 100$ Number of hidden units.

You think if is good to pick 10 - 100 units.

layers: 2-4.

Appropriate scale for HP.

Say for $\alpha = 0.0001 \sim 1$. If sample uniformly. 90% will between 0.1 to 1. Only 10% in 0.0001 to 0.1.

Instead. you should do log.

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

$\alpha = 10^r$: $10^{-4}, \cdots, 10^0$.

HP for exp. weighted averages.

$\beta = 0.9, 0.99, 0.999$.

Averaging: $\downarrow$ 10. $\downarrow$ 100 $\downarrow$ 1000

So it is un-reasonable to search from 0.9 to 0.999 linearly.

We can use the previous method to sample on

10^{-1} 10^{-3}

- Reason behind this.

$r \in [-3, -1]$. $1-\beta = 10^r$. $\beta = 1 - 10^r$.

When β change from 0.9 to 0.9005. (Average change in 10 values).

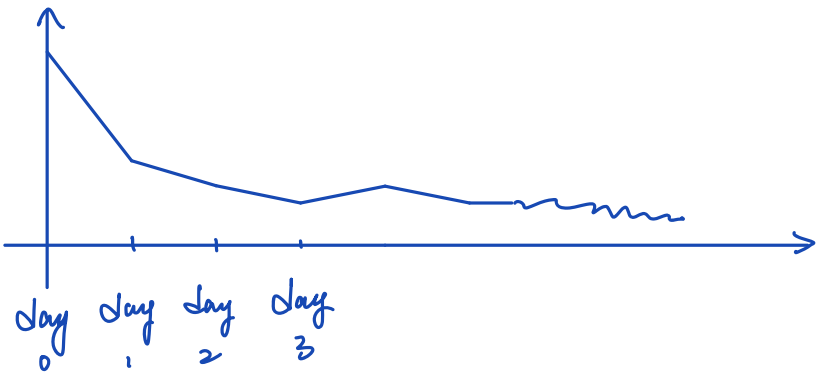
Not a big deal.

When β change from 0.9990 to 0.9995. (Average change 1000 values).

It is a huge change.

HP tuning in practice: Pandas VS Caviar.

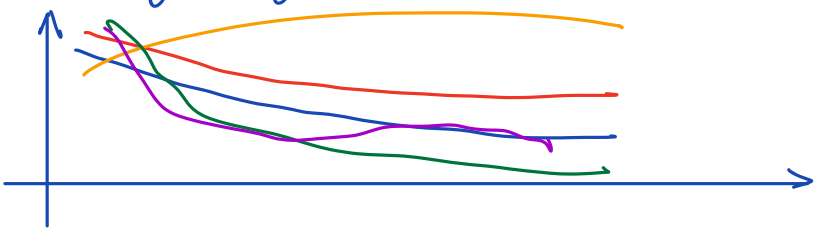
Baby-sitting one model.



Panda model.

Only one baby. Try lots of thing.

Training many models in parallel.



Basically trying a lot of models.

Caviar. fishing. Didn't pay a lot of attention on specific one.

Normalizing activations in a network.

Normalizing inputs can speed up learning.

 for gradient descent

What about multiple layers.

Can we normalize $a^{[i]}$ so to train $w^{[i+1]}$ $b^{[i+1]}$ faster.

We normalizing $z^{[i]}$. (There are debate whether normalizing before or after activation function)
(z) (a).

- Implementing Batch-Norm.

Given some intermediate value in NN.

Hidden units: $z^{(1)}, \cdots, z^{(m)}$. $\longrightarrow$ ($z^{[l](i)}$.)

which layer.

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

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

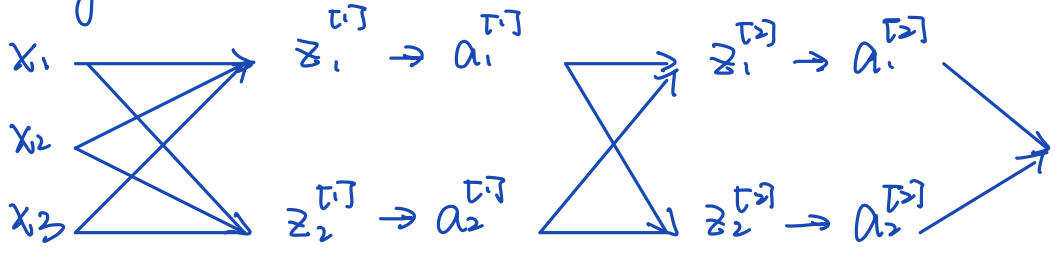
$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta. \quad (\gamma \text{ and } \beta \text{ are learnable params})$$

This allow you to set mean and variance of $z^{(i)}_{norm}$

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

Fitting BN into a NN.



$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch-Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \to a^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \to a^{[2]}$

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \cdots, W^{[L]}, b^{[L]}$.
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \cdots, \beta^{[L]}, \gamma^{[L]}$.

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

e.g. tensorflow: tf.nn.batch-normalization. (One-line code).

- Working with mini-batch.

$X^{\{1\}} \xrightarrow{W^{[1]}, b^{[1]}} z^{\{1\}} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{\{1\}} \cdots$.

Normalizing only inside the mini-batch.

Params: $W^{[L]}, b^{[L]}, \beta^{[L]}, \gamma^{[L]}$.

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

When compute the norm of $z^{[L]}$. We actually cancel the constant $b^{[L]}$.

So we only do $z^{[L]} = W^{[L]} a^{[L-1]}$. then compute $z^{[L]}_{norm}$.

$\Rightarrow \tilde{z}^{[L]} = \gamma^{[L]} z^{[L]}_{norm} + \beta^{[L]}$. ($\beta^{[L]}$ is actually performing the "bias")

Implement GD.

for $t = 1, \cdots,$ # 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 $dW^{[L]}, d\beta^{[L]}, d\gamma^{[L]}$.

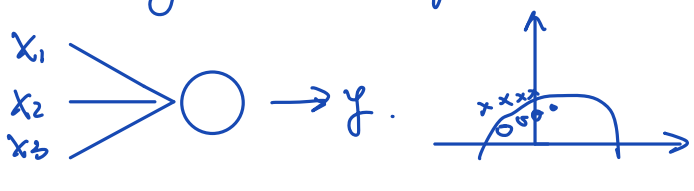
Update: $W^{[L]} = W^{[L]} - \alpha dW^{[L]}$.

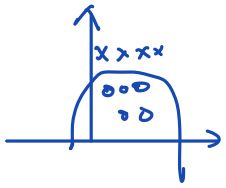
$\beta^{[L]} \cdots, \gamma^{[L]} = \cdots$

Work with momentum. RMSprop. Adam. $\cdots$

Why does BN work?

learning on shifting input distribution.





o: Pos e.g.
x: Neg e.g.

e.g. Cat. Non-Cat. Cat Non-Cat.
$y=1$ $y=0$. $y=1$ $y=0$

So your classifier won't work really well.

Black cats. ⟶ Colored-Cat

"Covariate Shift".

Why this is a prob. for NN?

e.g. for $w^{[3]}, b^{[3]}$.
$w^{[3]}, b^{[3]}$. $w^{[4]}$ $b^{[4]}$

$a^{[2]}$.
$a_2^{[2]}$ ⇒ ◯ ⇒ $\hat{y}$.
$a_3^{[2]}$
$a_4^{[2]}$

x_1
x_2 ⇒0⇒
x_3

As $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ change.
$a^{[2]}$ keep changing in $w^{[3]}, b^{[3]}$ perspective.
Distribution

What BN does. is "fixing" ~~the shift of~~ the hidden units distribution.
No matter how it changes. Mean and variance governed by γ and β.
Make hidden units value more STABLE. (Learn indep. for each layer)

- BN as regularization.
  - Each mini-batch is scaled by the mean/var computed on just that minibatch. (which has some noise ↓) Similar for β and γ.
  - This adds some noise to the values $z^{[l]}$ within that minibatch. Some similar to dropout. it adds some noise to each hidden layer's activation.

- This has a slight regularization effect. (unintended effect).

Batch Norm in Test time. in test time, you may not have "mini-batch".

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

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

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

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

μ, σ^2: estimated using exp. weighted average (across mini-batch).

$x^{\{1\}}, x^{\{2\}} \cdots$

$\mu^{[l]\{1\}} \rightarrow \mu^{[l]\{2\}} \cdots$

Compute $\theta_1, \theta_2 \cdots$ using exp. weight average. And finally get the μ.

Keep a running average. Similar for σ^2.

Compute $z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$. $\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$. Use these μ, σ^2 for the hidden units in your network during test time.