Setting up your ML application

DL spec \\ Course II \\ week 1

→ Train / dev / test sets

→ highly iterative process

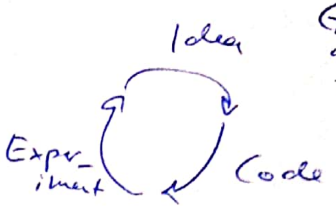
Hyperparameters

# layers

# hidden units

# learning rates

# types of activation functions



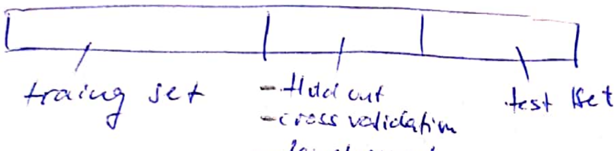
Go fast around this cycle

DL → NLP, Comp Vision, Speech, Structured data — Ads, web search, Security, Logistics → Intuition doesnt transfer easily

↳ people can't just transfer knowledge from ads to security for example

Set datasets up well increases efficiency

Training data



⇒ workflow: train algorithms on training set, use "dev" set to see which of different models performs best, after that evaluate best model on test set to get unbiased estimate

(Previous era: 70/30 60/20/20)
% % or % % %

Modern era : if m = 1'000'000 → 10'000 in dev set is enough

↳ $\boxed{\begin{matrix}\text{train} & \text{dev} & \text{test}\\ 98\% / & 1\% / & 1\%\end{matrix}}$

10'000 also enough for test set

Mismatched train/test distribution

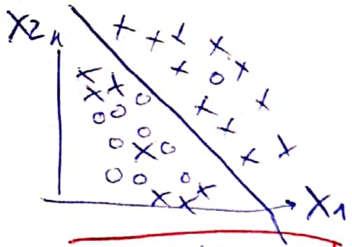
training set: Cat pictures from webpages ←different quality (different distributions)→ dev/test sets: Cat pics from users using app

! → Make sure that dev and test sets come from same distribution (as training set)

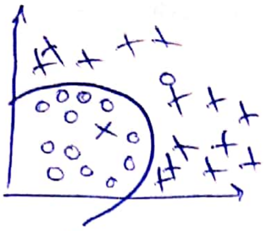
→ Not having a test set might be OK. Must have dev set!

! ↳ If only a test and no dev set is present, algorithm might be overfitted to test set (because adjusted to results only)

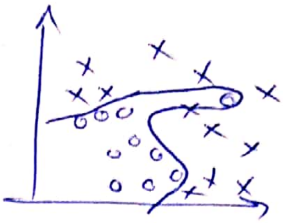
Bias/Variance



high bias
→ underfitting



→ just right
→ misses a few but that is OK



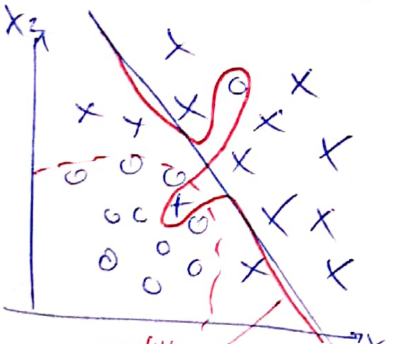
high variance
→ overfitting

→ In practice can't plot data from higher dimensions
↳ so other metrics: Bias and Variance

→ cat classification cat (y = 1) non-cat (y = 0)

train set error	1%	15% → can't even get training set right	15% → high bias	0.5%
Dev set error	11% (compared to 0% human error)	16%	30%	1%
	"high variance"	"high bias" as underfitting	high bias and high variance	low bias, low variance

based on assumption: human ≈0% error
(↳ generally: optimal error (Bayes) error: ≈0%
and that sets are drawn from the same distribution



high: → high variance
→ bias: because needed a quadratic function (---) but was a straight line
Variance: too much flexibility also capturing outliers

DL spec Course II | week 1 |

Basic recipe for machine learning

→ Does it have high bias? (training data performance) —yes→ → try bigger network (L, n_h) → train longer (GD longer or advanced opt. algorithms ADAM) (underfitting?) (NN architecture search)

↓ no

high variance? (dev set performance?) —yes→ → Get more data → Regularization (reduce overfitting) (NN architecture search)

↓ no

done ☺

training a bigger network almost never hurts

usually use train and dev set to diagnose and then select the appropriate measure to try

→ If high bias problem, getting more training data won't help

→ "Bias variance trade off" ← could improve only one in old DL era.

→ new DL era allows to optimize both

Regularization

on example of Logistic Regression

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

λ = regularization parameter ← set using dev-set! → In Python "lambd" so not to clash with lambda internal function

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

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

weight!

usually omitted as w is of higher dimension

L_2 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ (Euklidean norm)

L_1 regularization: $\ldots + \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$ → w will be sparse ↳ vector will have a lot of zeros

how abouth in Neural network

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

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

how do GD now?

↳ $dw^{[l]} = \text{(from BP)} \; \frac{\partial J}{\partial w} + \frac{\lambda}{m} w^{[l]}$

L_2 regularization is sometimes called: "weight decay"

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

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

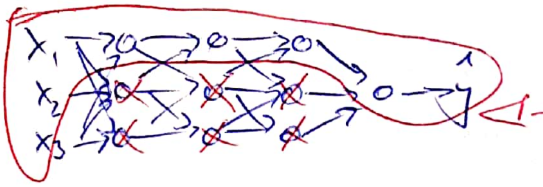
$w^{[l]} = \underbrace{w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]}}_{↳ \, w^{[l]} \cdot (1 - \frac{\alpha\lambda}{m})} - \alpha \, \text{(from BP)}$

Why regularization L_2 reduces overfitting?

How does it prevent overfitting?

in case of overfitting

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

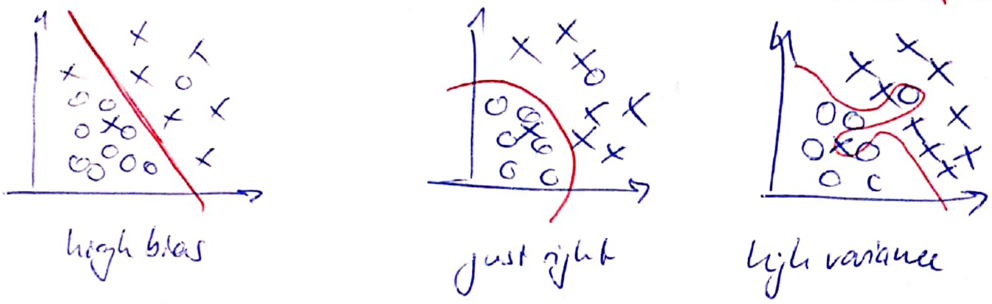


if λ is really big

set $W^{[l]} \approx 0$ so that it zeros out a lot of hidden units' impact

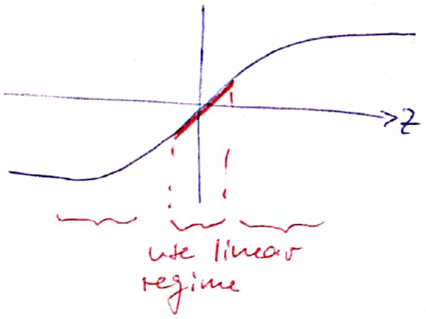
↳ will rather lead to high bias

↳ might enable to find some intermediate λ to be "just right"



Intuition of zeroing out a bunch of hidden units isn't quite right... what actually happens is they are still there but with less effect

⇒ Regularization ⇒ Variance reduction



tanh $g(z): \tanh(z)$

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

If λ is large, parameters are small because penalized for being large

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

if $z^{[l]}$ takes small values (as in liner regime)

$g(z^{(l)})$ will also be roughly linear

$\boxed{\text{↳ Every layer} \approx \text{linear}}$

↳ $\boxed{\text{whole network will be linear}}$

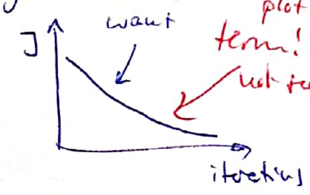
↳ can only compute linear function ☹
(and thus not overfit)

*remember from course I: linear activation functions ☹

Implementational tip

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

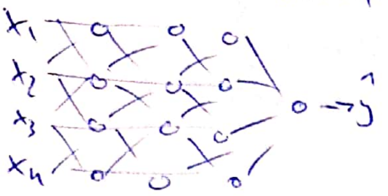
Gradient Descent



plot the entire term! Otherwise might not see decrease!

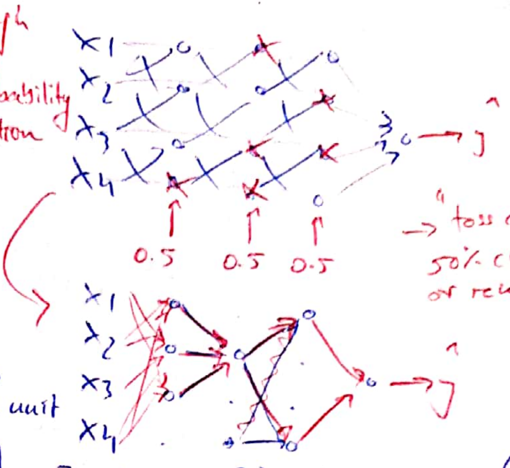
Dropout Regularization

1st NN (overfitting) $\xrightarrow{\text{create copy}}$ Copy of NN



go through all layers add some probability of node elimination

"cutting out nodes at random"



→ "toss coin and get 50% chance of keeping or removing node"

Implement dropout ("Inverted dropout")

Illustrate with layer l=3 keep-prop = 0.8 probability that a hidden unit will be kept

d3 = np. random. rand (a3. shape[0], a3. shape[1]) < keep-prob (0.2 → 20% chance of elimination)

dropout vector of layer 3

↳ generates a random matrix d3

a3 = np. multiply (a3, d3) elementwise multiplication | technically d3 is boolean matrix

$\boxed{\text{a3/= 0.8 (keep-prop)}}$ inverted dropout technique → by dividing by "keep-prop" the expected activation is corrected!!!

<u>e.g.</u> 50 neurons in 3rd layer → average 10 units shut off

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

reduced by 20%

↳ /= 0.8

expected value of a3 is not changed

→ makes test time easier because less of a scaling problem

$\boxed{\text{→ make multiple passes through training set!}}$

Making predictions at test time

$a^{[0]} = X$

No drop out! during test time!

↓ just adds noise! to prediction

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$
$$\vdots$$
$$\downarrow$$
$$\hat{y}$$

/= keep-prop

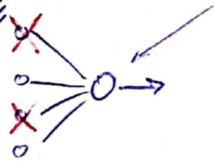
→ effect is to ensure that even if we don't use drop-out at test time that the scaling of the expected activation values does not change so do not need to add in another scaling factor

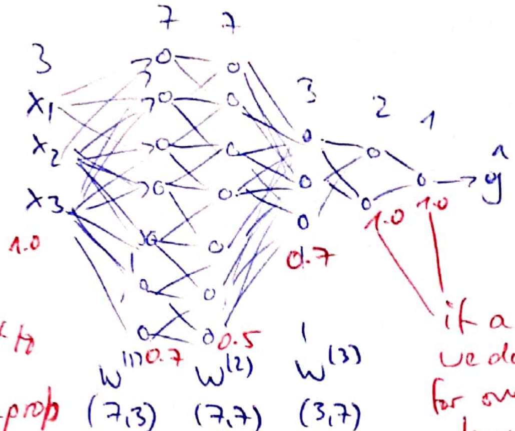
Understanding drop-out

Intuition: Can't rely on any one feature, so have to spread out weights

↓

shrinking squared norm of weights L_2





might want to vary keep-prob per layer!

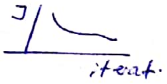
biggest matrix!! } have lower keep-prob in this layer!

if a layer's weight we do not worry about for overfitting we can have keep-prob = 1.0 } regularization where ~~necessary~~ ~~necessary~~ necessary

→ often used in computer vision

downside: Cost J is not any more well defined

↳ lose "debugging tool" to plot



so turn off keep-prob off by setting = 1

Other regularization techniques

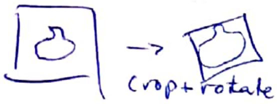
exp. cat classifier. If overfitting, more data could help but:

↳ Use data augmentation:

flip
horizontally

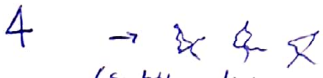
↳ 2 times as many data (but is a bit redundant of course)

also use random crops



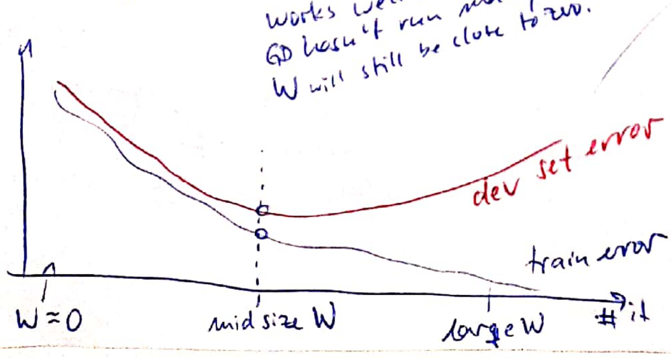
crop + rotate or randomly zoom in

or



(subtle distortions are OK)

Early stopping



works well because if GD hasn't run many iterations W will still be close to zero.

similar to L_2 norm $\|w\|_F^2$

1st task
- Want Optimize J (GD...) } $J(w,b)$
- Not overfit (Regularization...) } different task not to overfit

↳ one task at a time!

Orthogonalization!

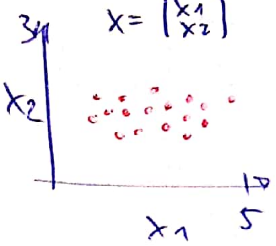
downside: Early stopping is coupling optimizing J and trying not to overfit

Rather use L_2

Normalizing Inputs

Speed up Optimization

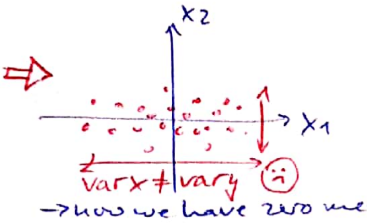
$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



1st step: Subtract mean

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

$x := x - \mu$



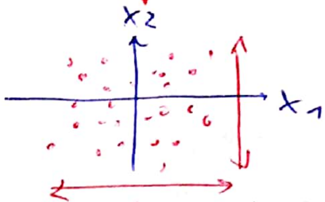
→ now we have zero mean

→ 2nd: Normalize variance

$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)2}$ | Python: ($x^{(i)}$ ** 2) element wise

$x /= \sigma$ (not squared!)

⇓



variancex = vary = 1

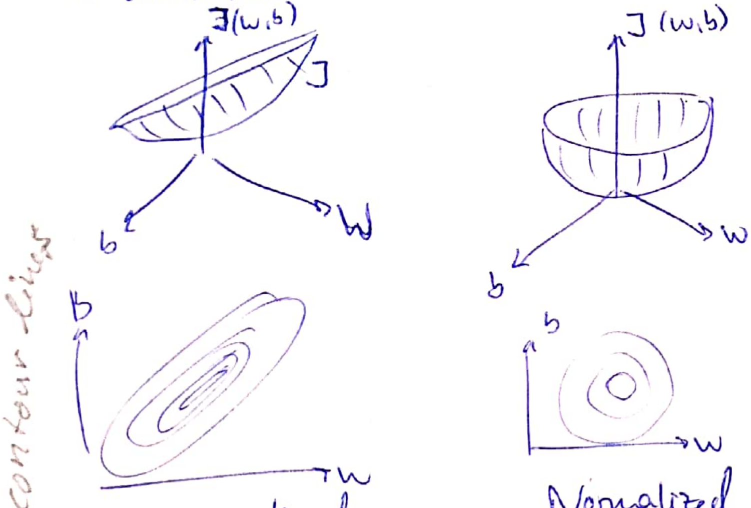
Use same μ and δ^2 to nomalize test set!!!

↳ [redacted] set should be normalized equally!

Why nomalize features?

→ If not, cost function might be distorted!

x_1, x_2 should have similar ranges (don't have to be exactly equal)



Unnormalized

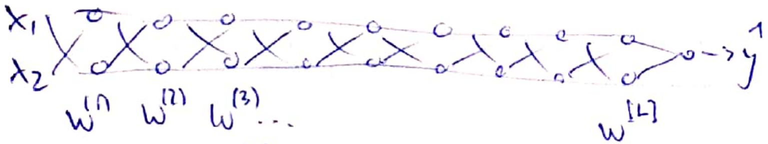
Normalized

Gradient Descent can take much larger steps!

do normalization anyway!

Vanishing or exploding gradients

Comes from poor initialization!



$g(z) = z \quad b^{[l]} = 0$

$y = w^{[L]} \cdot w^{(L-1)} \dots w^{(L-2)} \dots w^{[3]}, w^{(2)} \underbrace{w^{(1)} \cdot x}$

$z^{(1)} = w^{(1)} \cdot x + 0$

$\underbrace{a^{(1)} = g(z^{(1)}) = z^{(1)}}$

$a^{(2)} = g(z^{(2)}) = g(w^{(2)} \cdot a^{(1)} + 0)$

Identity matrix

$w^{[l]} > I$ → ~~de~~ increase exponentially

$w^{[l]} < I$ → decrease exponentially

→ if gradients are small GD will be slowed down heavily!

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

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

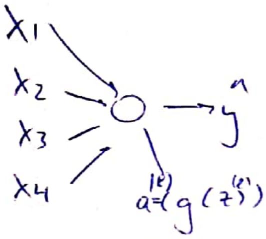
$\hat{y} = 1.5^{(L-1)} \cdot x$

→ $\hat{y}$ will exponentially grow when deep neural network

when $\begin{bmatrix} 0{,}5 & 0 \\ 0 & 0{,}5 \end{bmatrix}$

→ $\hat{y}$ will decrease exponentially

Weight initialization for deep networks



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

Variance parameter

ReLU → ... * np.sqrt$\left(\frac{2}{n^{[l-1]}}\right)$

He initialization ↳ $Var(W_i) = \frac{2}{n}$

$z = w_1x_1 + w_2x_2 + \dots w_nx_n \mid b=0$

$n \uparrow$ $w_i \downarrow$ smaller w_i with larger n

$Var(W_i) = \frac{1}{n}$ HERE n IS # OF INPUT FEATURES GOING IN A NEURON

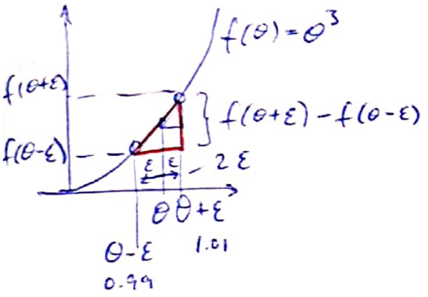
tanh → $\sqrt{\left(\frac{1}{n^{[l-1]}}\right)}$ ↳

Xavier initialization

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

Variance parameter can be seen as hyperparameter and also be tuned

Numerical approximation of gradients



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

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

$$\underset{\theta=1}{g(\theta)} = \underset{3\cdot1^2}{3\theta^2} = 3$$

approx. error: 0,0001

Gradient Checking

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

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

In Python this is solved via dictionaries → functions convert to vectors and vice versa

↳ Is $d\theta$ the gradient of J? | now $J(\theta)$

for each i:

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

↳ $d\theta_{approx}$

↳ $d\theta$ $\overset{?}{\approx}$ Is this roughly equal? Difference ¿ In practice $\varepsilon \approx 10^{-7}$ threshold

Check euclidean distance between vectors $\boxed{\frac{\|d\theta_{approx}-d\theta\|_2}{\|d\theta_{approx}\|_2+\|d\theta\|_2}}$

if $\approx 10^{-7}$ great! ☺

if $\approx 10^{-5}$ maybe OK

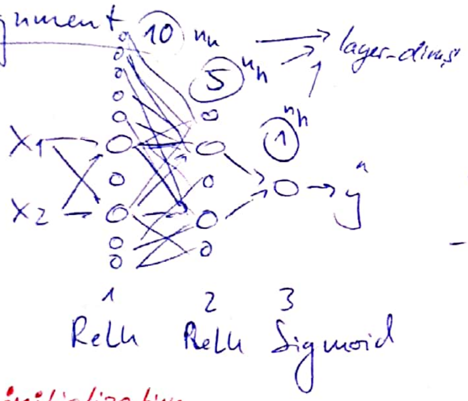
if $\approx 10^{-3}$ not good ☹ bug somewhere

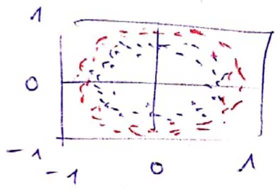
Gradient Checking Implementation notes

- Don't use in training - only to debug
- If algorithm fail grad check, look at components to try to identify bug.
- Remember regularization.
- Doesn't work with drop out! (keep_prob = 1.0)
- Run at random initialization, perhaps again after some training

recommendation:
- turn off drop out
- use grad check
- turn on drop out

1st Assignment





⇒ want to find a decision boundary

$y = $ 0 for red
1 for blue

$\underline{X} \in \mathbb{R}^{(n_x, m)}$ $y \in \mathbb{R}^{(1, m)}$

zero initialization

⇒ The weights $W^{[l]}$ should be initialized randomly to break symmetry

⇒ It is OK to initialize biases $b^{[l]}$ to zero. Symmetry is still broken when $W^{[l]}$ is initialized randomly

random initialization (large values * 10)

⇒ Initializing $W^{[l]}$ to large random values does not work well

⇒ Need to see how small values need to be...

He initialization (He et. al, 2015)

⇒ works well

Take aways :

⇒ Different initializations lead to different results

⇒ Random initialization is used to <u>break symmetry</u> and make sure different hidden units can learn different things

⇒ Don't initialize to values that are too large

⇒ He initialization works well for networks with ReLu activations.

2nd Assignment: Regularization + Dropout

Regularization

→ λ is a hyperparameter to tune using dev set

→ L2 Regularization makes the decision boundary smoother

↳ if λ is too large it can be "oversmooth" resulting in high bias

↳ L_2: assumption that model with small weights is simpler
Thus, by penalizing the square value of the weights in the cost function you drive all the weights to smaller values.
This is too costly for cost to have large weights!
This leads to a smoother model in which outputs changes more slowly as the input changes.

⇒ Regularization adds term to cost function

⇒ Backpropagation: There are extra terms in the gradients w.r.t. $W^{[l]}$

$$\frac{\partial}{\partial W}\left(\frac{1}{2}\frac{\lambda}{m}W^2\right) = \underbrace{\frac{\lambda}{m}W^{[l]}}_{\text{term to add}}$$

⇒ Weights end up smaller ("weight decay")

Dropout: (Deep learning specific regularization method).

⇒ Dropout is a regularization technique

⇒ Only apply during training, don't use during testing

⇒ Apply Dropout during Forward Propagation and Backpropagation

⇒ During training time, divide each drop-out layer by keep_prob to keep the same expected value for the activations. If keep_prob value is eg. 0.5 we will on average drop 50% of the neurons per layer so affecting the output as well, since only the remaining neurons are contributing. Dividing by keep-prop (0,5) is equal to multiplying by 2. Now the outcome has the expected value.

Gradient Checking

⇒ GC verifies closeness between gradients from backpropagation and the numerical approximation of the gradient (computed using FP).

⇒ GC is slow, so we don't run it in every iteration of training.
Just run to make sure code is correct, then turn off and use BP for actual learning process