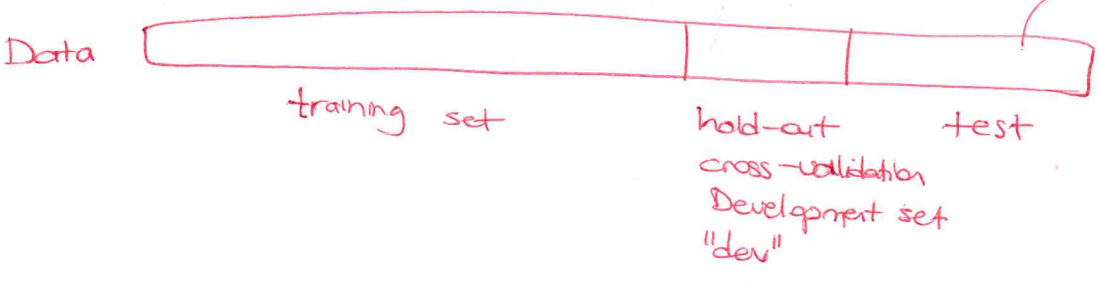
COURSE 2
WEEK 1

Train / Dev / Test sets

Applied ML is a highly iterative process

#layers
#hidden units
learning rates
activation functions
...



The goal of the test set is to give you a unbiased estimate of the performance of your final network.

* The goal is to see which of many different models performs best on your dev set

Mismatched train/test distribution

Training set:
Cat pictures from webpages
- high resolution
- very professional
- very nicely framed pictures

Dev/test sets:
Cat pictures from users using your app
- blurrier
- low resolution.

=> So these two distributions of data may be different

Rule:
Make sure dev and test sets come from the same distribution

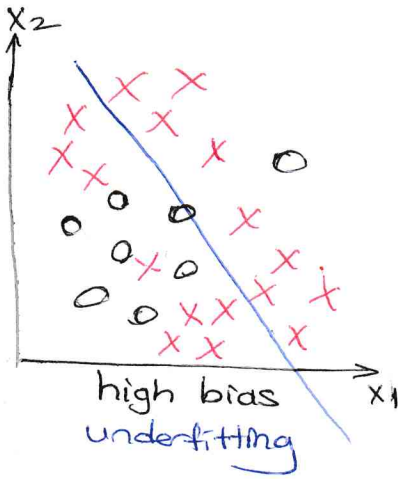
→ Not having a test set might be okay (Only dev set)

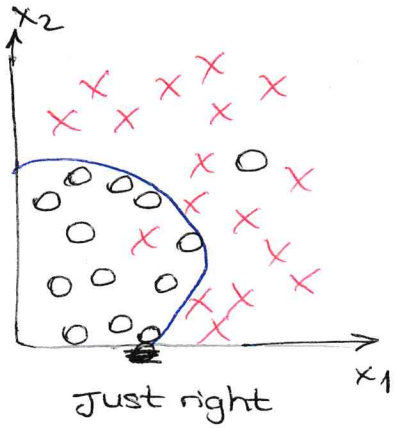
train / development set would be more correct terminology

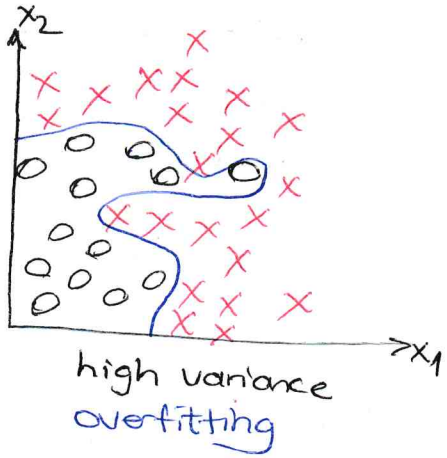
So: Having set up a train, dev and test set will allow you to integrate more quickly.

It will also allow you to more efficiently measure the bias and variance of your algorithm. So ways can be selected more efficiently to improve algorithms.

Bias / Variance







For Example : CAT CLASSIFICATION

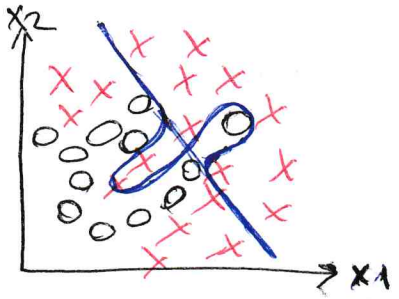
Train set error :	1 %	15 %	15 %	0.5 %
Dev set error :	11 %	16 %	30 %	1 %
	high variance	high bias	high bias & high variance WORST	low bias low variance

Human : $\approx$ 0%

Optimal (Base) error : $\approx$ 0%

By looking at training set error, you can get a sense of how well you are fitting. So that it tells you have a bias problem

With dev set error, that should give you a sense of how bad is the variance problem.



high bias (too general)
high variance (too much flexibility)

Basic Recipe For Machine Learning

high bias ? (training data performance)
- Y → Bigger network (more hidden layers, hidden units) (you could try)
  - Train longer or try some more advanced optimization algorithms
  - (NN architecture search)
- N ↓

High variance ? (dev set performance)
- Y → more data
  - Regularization
  - (NN architecture search)
- N ↓

Done

MAYBE

"bias variance tradeoff"

* Training a bigger network almost never hurts. And the main cost of training a neural network that's too big is just computational time, so long as you're regularized

Regularization is very useful technique while reducing the variance.
There is a little bit of a bias variance tradeoff when you're using regularization

Regularization

Overfitting → high variance problem

Logistic Reg:

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

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

→ penalizes weight matrix from being too much

omit

L2 regularization (most common type) $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

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

λ = regularization parameter

"lambd"

Neural Network:

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

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

"Frobenius norm" means the sum of squares of elements of a matrix)

L2 regularization is sometimes also called "weight decay"

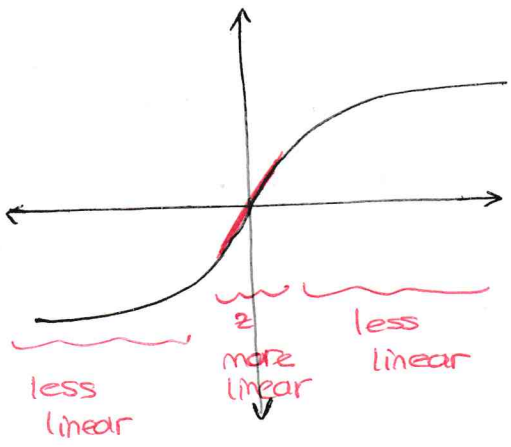
$$w^{[l]} := w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

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

$$\left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]}$$

So you are multiplying the weight metrics by a number of slightly less than 1.

Why Regularization reduces overfitting?



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

If λ is too high then w would be ≈ 0

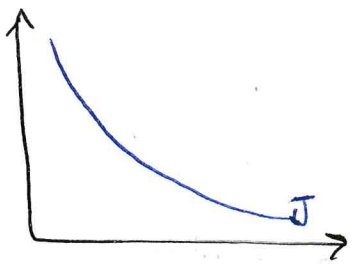
It would make the some hidden units smaller and gets the smaller neural network.

$\lambda \uparrow \quad w^{[l]} \downarrow \quad z^{[l]} = w^{[l]} \cdot a^{[l-1]} + b^{[l]}$

↓

will look like a simpler model like linear regression

$$J(\text{---}) = \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_l \|w^{[l]}\|_F^2$$



Dropout Regularization

Implementing dropout ("Inverted Dropout")

most common dropout implementation

Illustrate with layer=3 keep_prob = 0.8

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

50 units ⇝ 10 units shut off

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

↑ This will be reduced by %20

Making Predictions at Test Time:

No dropout

If you implement test time, it just add noise to your predictions.

Understanding Dropout

Why does drop-out work?

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

shrink weights similar L2 reg.

Dropout has a similar effect to L2 regularization

more adaptive to the scale of different inputs.

It is also feasible to vary keep_prop by layer.

For the layer with a lot of parameters, keep_prop can be set smaller to apply a more powerful form of dropout.

Many of the first successful implementations of dropouts were to computer vision

In computer vision input size is so big, inputting all these pixels that you almost never have enough data. So dropout is very frequently used.

Other Regularization Methods

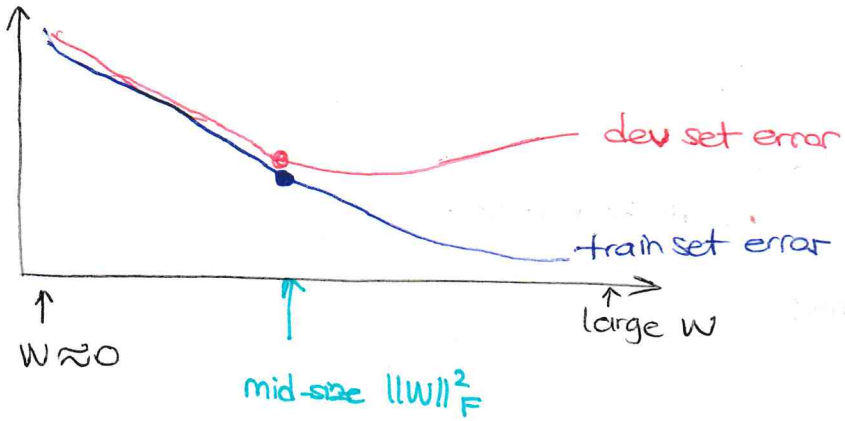
Data Augmentation:

Image of cat can be rotated or randomly zoomed in order to get more data.

This can be an inexpensive way to give your algorithm more data.

So data augmentation can be used as a regularization technique

Early Stopping:

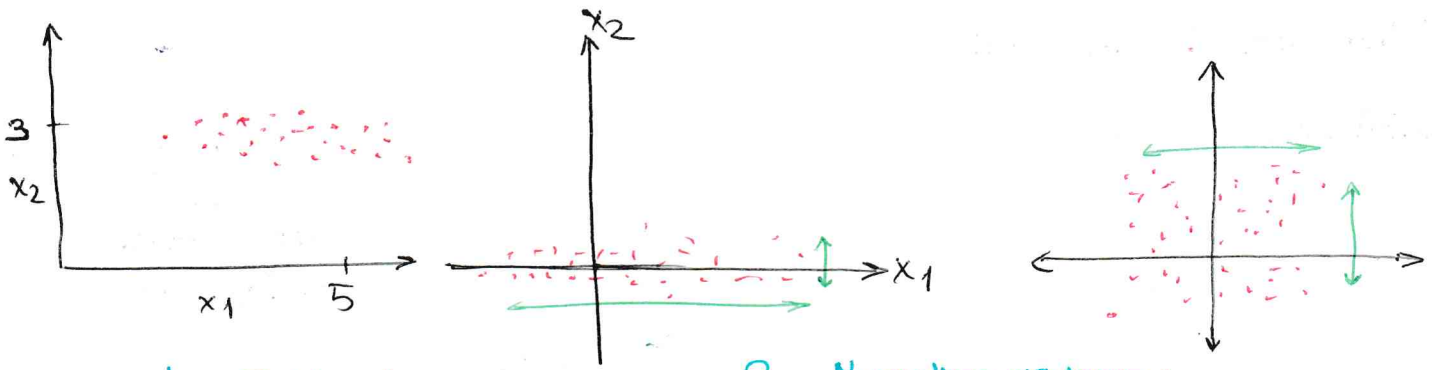


Orthogonalization

- Optimize the cost function J
  Gradient Descent, momentum, RMS prop, Atom ...
- Not overfit
  Regularization, getting more data ...

Early stopping does let you get similar effect without needing to explicitly try lots of different values of lambda.

Normalizing Inputs



1. Subtract mean:

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

$$x := x - \mu$$

2. Normalize variance:

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

** element-wise

$$x /= \sigma^2$$

For train and test set, same steps should be used.

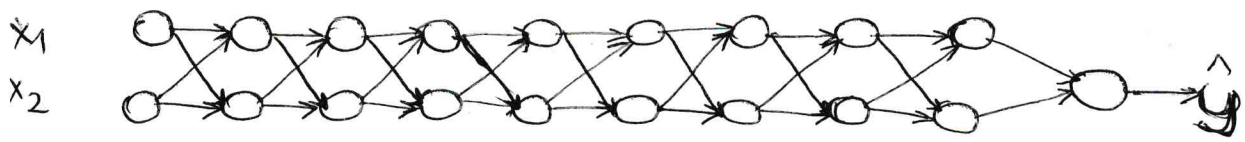
If you don't normalize, you should keep small the learning rate. But when inputs are normalized, learning-rate can be larger.

Normalizing makes your cost function J easier and faster to optimize.

All your features on a similar scale will usually help your Learning algorithm run faster.

Vanishing / exploding Gradients

One of the problems of training neural network especially very deep neural networks, is data vanishing and exploding gradients.



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

$\hat{y} = W^{[l]} W^{[l-1]} \dots W^{[3]} W^{[2]} W^{[1]} X$

$z^{[1]} = W^{[1]} x$

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

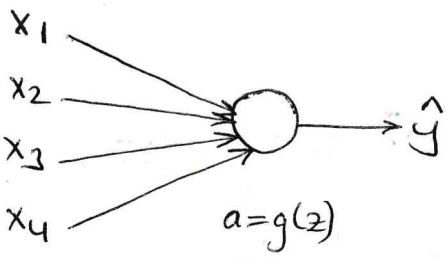
$a^{[2]} = g(z^{[2]}) = g(W^{[2]} a^{[1]})$

$W^{[l]} > I \Rightarrow$ exploding

$W^{[l]} < I \Rightarrow$ vanishing

If your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small. And this makes training difficult, especially if your gradients are exponentially smaller than L then gradient descent will take tiny little steps. It will take a long time for gradient descent to learn anything.

Weight Initialization for Deep Networks



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

large $n \rightarrow$ smaller w_i

$\text{Var}(w_i) = \frac{1}{n}$ → for ReLU

$\text{Var}(w_i) = \frac{2}{n}$ → for ReLU

$W^{[l]} = \text{np.random.rand(shape)} * \sqrt{\frac{2}{n^{[l-1]}}}$ ← for ReLU

Variance parameter could be another thing that you could tune of your hyperparameters.

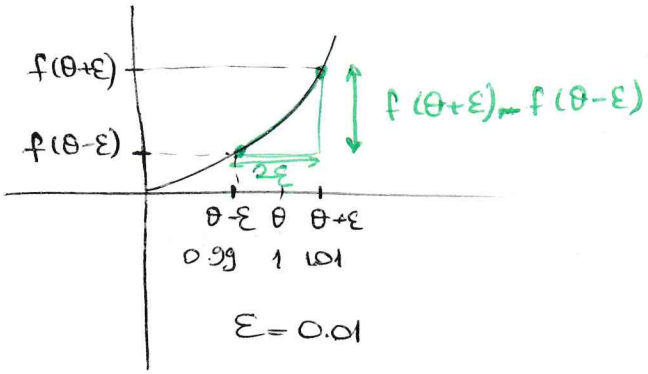
Other variants:

tanh = $\sqrt{\frac{1}{n^{[l-1]}}}$ called "Xavier"

$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$ → by Yoshua Bengio and colleagues

With these functions $W^{[l]}$ won't be too much bigger than 1 and too much less 1. So it doesn't explode or vanish too quickly.

Numerical Approximation Of Gradients



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

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

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

approximation error = 0.0001

(prev. slide: 3.0301 error: 0.03)

Note: $f'(\theta) = \lim_{\varepsilon\to 0} \frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$ → $O(\varepsilon^2)$ — more accurate

0.01
0.0001

$\lim_{\varepsilon\to 0} \frac{f(\theta+\varepsilon)-f(\theta)}{\varepsilon}$ error: $O(\varepsilon)$ — less accurate

0.01

You can numerically verify whether or not a function g, $g(\theta)$ is a correct implementation of the $f'(\theta)$

Gradient Checking

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

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

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

It is used to find bugs in the implementation of backpropagation.

for each i:

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

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

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

$\approx$ 10^{-7} – great

10^{-5}

10^{-3} – worry

Gradient Checking Implementation Notes

- Don't use in training – only to debug
- If algorithm fails grad check, look at components to try to identify bug.
- Remember regularization — $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l}\|W^{[l]}\|_F^2$
- Doesn't work with dropout — $d\theta$ is gradient of with respect to θ
- Run at random initialization; perhaps again after some training.

Mini-Batch Gradient Descent

Batch vs mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.

baby training sets = mini-batches

$$X = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)}\ \dots\ x^{(1000)}}_{X^{\{1\}}\ (n_x, 1000)}\ |\ \underbrace{x^{(1001)}\ \dots\ x^{(2000)}}_{X^{\{2\}}\ (n_x, 1000)}\ |\ \dots\ |\ \underbrace{\dots\ x^{(m)}}_{X^{\{5000\}}\ (n_x, 1000)}]$$

(n_x, m)

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

what if $m = 5000000$?

Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$

for $t = 1, \dots, 5000$

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

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

} Vectorized Implementation

1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$ (as if $m = 1000$)

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

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

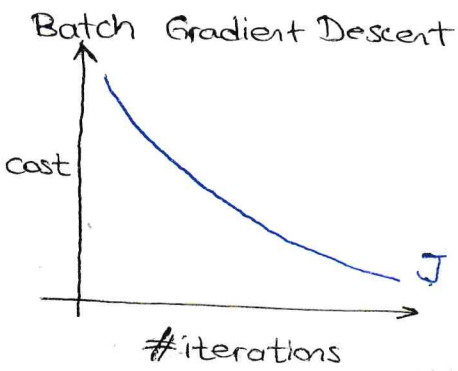
$$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

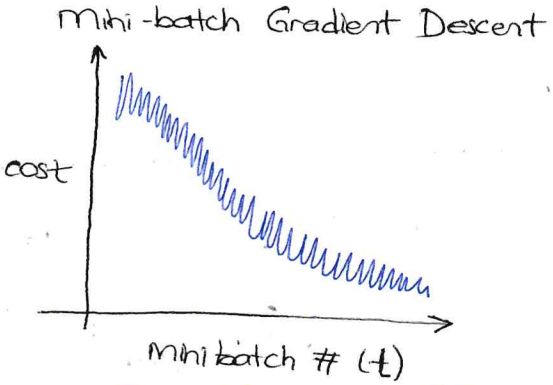
"1 epoch" $\Rightarrow$ 1 single pass through training set.

! Whereas with batch gradient descent, a single pass through the training allows you to take only one gradient descent step, with mini-batch gradient descent, a single pass through the training set, one epoch, allows you to take 5000 gradient descent steps.

Understanding Mini-batch Gradient Descent

Training With Mini Batch Gradient Descent



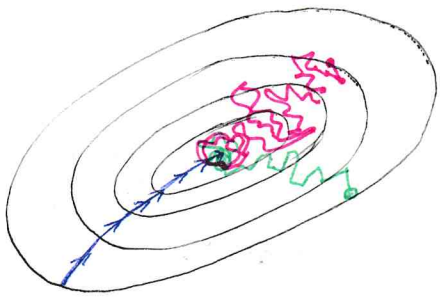


Plot $J^{\{t\}}$ using $X^{\{t\}}, y^{\{t\}}$

Choosing your Mini-batch Size

If mini-batch size = m : Batch gradient descent $(X^{\{1\}}, y^{\{1\}}) = (X, y)$

If mini-batch size = 1 : Stochastic Gradient Descent. Every example is its own $(X^{\{1\}}, y^{\{1\}}) = (X^{(1)}, y^{(1)}) \dots (X^{(2)}, y^{(2)})$



In practise : Somewhere between 1 and m

Disadv. of stochastic gradient descent
↓
lose almost all your speed up from vectorization
The way you process each example is going to be very inefficient

In-between (mini-batch size not too big/small)
↓
Fastest learning
- vectorization (a lot of)
- make progress without needing to wait till you process the entire training set.

Disadv. of batch gradient descent (mini-batch size = m)
↓
Too long per iteration

If small training set : Use batch gradient descent ($m \leq 2000$)

Typical mini-batch sizes :

64 , 128 , 256 , 512 , 1024
(2^6 , 2^7 ... 2^8 , 2^9 , 2^{10})
more common

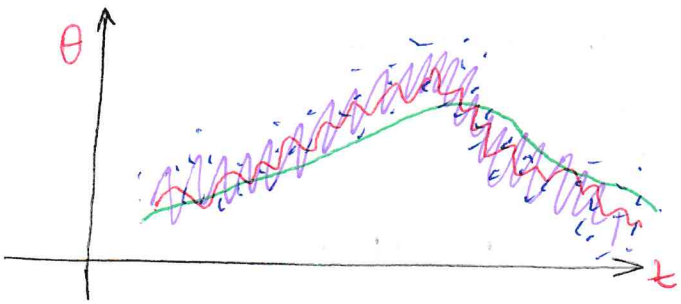
Make sure mini-batch fit in CPU/GPU memory
$X^{\{t\}}, y^{\{t\}}$

Exponentially Weighted (moving) Averages

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

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

- $\beta = 0.9$: $\approx$ 10 days' temperature
- $\beta = 0.98$: $\approx$ 50 days
- $\beta = 0.5$: $\approx$ 2 days



Understanding Exponentially Weighted Averages

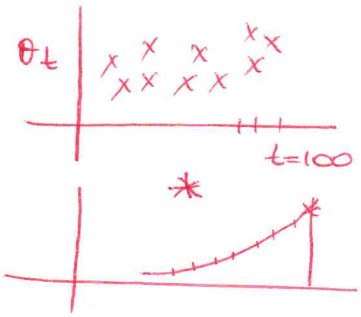
$V_t = \beta V_{t-1} + (1-\beta)\theta_t$

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

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

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

...



$\approx \frac{1}{1-\beta}$

$\varepsilon = 1-\beta$

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

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

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $\quad (1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$ $\quad \underbrace{}_{0.9}$

$\varepsilon = 0.02 \rightarrow 0.98^{50} \approx \frac{1}{e}$

$V_\theta := 0$

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

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

⋮

$V_\theta = 0$

Repeat {

Get next θ_t

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

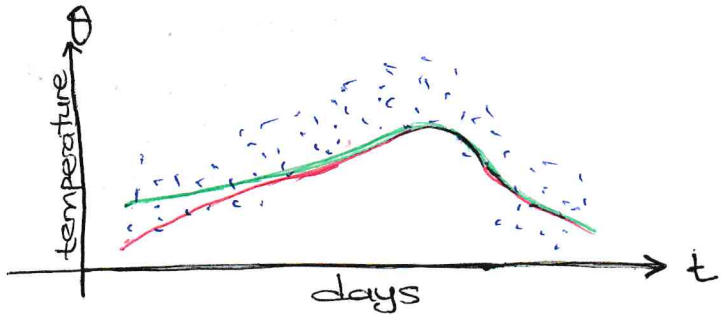
}

Advantage: - It takes very little memory
- One line of code

If you were to compute a moving ~~forward~~ window, where you explicitly sum over the last 10 days, last 50 days temperature and just divide by 10 or divide by 50 that gives you better estimate. But disadvantage of that of explicitly keeping all the temperatures around and sum of the last 10 days requires more memory. More complicated to implement and more expensive (computationally).

Bias Correction in Exponentially Weighted Average

Bias correction makes your computation of these averages more accurately,



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

$V_0 = 0$

$V_1 = 0.98 V_0 + 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2$

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

$= 0.0196\theta_1 + 0.02\theta_2$

BIAS CORRECTION

$$V_t^{\text{corrected}} = \frac{V_t}{1-\beta^t}$$

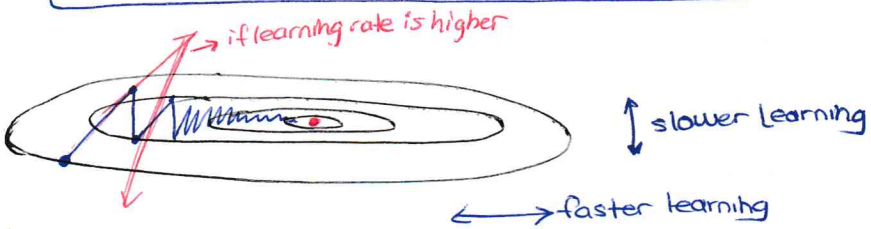
$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

$$\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

It helps you go from red line to green line

But we are concerned about the bias during this initial phase,
Then bias correction can help you get a better estimate early on.

Gradient Descent With Momentum



Implement gradient descent with momentum

Momentum:

On iteration t:

Compute dW, db on current mini-batch

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

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

friction ↑ velocity

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

So what this does is smooth out the steps of gradient descent

Implementation details

$V_{dw} = 0, \quad V_{db} = 0$

On iteration t:

Compute dW, db on the current mini-batch

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

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

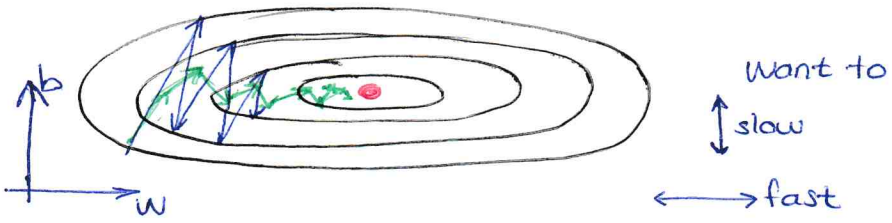
$W = W - \alpha V_{dw}, \quad b = b - \alpha V_{db}$

Hyperparameters: α, β $\beta = 0.9$

In practise, I don't really see people bothering with bias correction when implementing gradient descent or momentum.

$\frac{V_{dw}}{1-\beta^t}$

RMSprop (Root Mean Square Prop)



On iteration t:

Compute dW, db on current mini-batch

$S_{dW} = \beta S_{dW} + (1-\beta)\, dW^2$ ← element wise

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

small → large

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

if S_{dW} or S_{db} is 0 → $\varepsilon = 10^{-8}$

You can use larger learning rate (α) and get faster learning without diverging in the vertical direction.

Adam Optimization Algorithm

Adam optimization algorithm is basically taking momentum and rms prop and putting them together.

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

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

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

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

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

$$W := W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \varepsilon}, \quad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

In the typical implementation of Adam, you do implement bias correction.

This is commonly used learning algorithm that is proven to be very effective for many different neural networks of very wide variety of architectures.

Hyperparameters

α : need to be tune

β_1 : 0.9 → (dW) → is computing mean of the derivatives

β_2 : 0.999 → (dW^2) → is used to compute exponentially weighted average of dW^2

ε : 10^{-8} (doesn't need)

ADAM (Adaptive Moment Estimation

Learning Rate Decay

Slowly reduce α

1 epoch = 1 pass through data

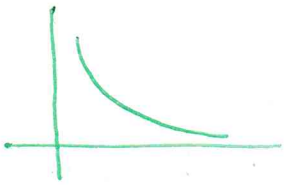
$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_num}} \cdot \alpha_0$$

↙ another hyperparameter

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

$\alpha_0 = 0.2$

decay-rate = 1

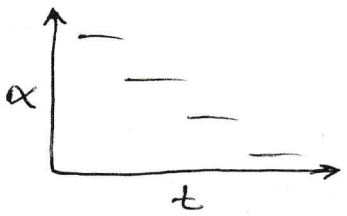


Other Learning Rate Decay Methods

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

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

↙ mini-batch number



discrete staircase

manual decay → this works only if you're training only a small number of models

The Problem Of Local Optima

derivative = 0 that point is called a saddle point

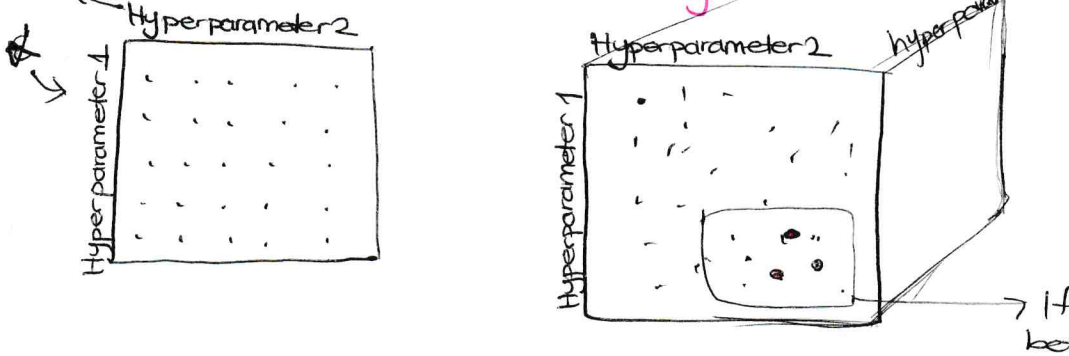
- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

HYPERPARAMETER TUNING

Tuning Process

- α → most important
- β
  - second important
  - third important
- $\beta_1, \beta_2, \varepsilon$
- # layers
- # hidden units
- Learning rate decay
- mini batch size

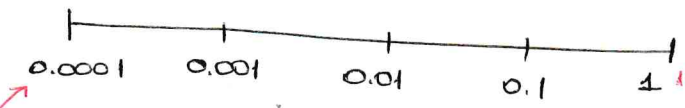
Try Random Values : Don't use grid



→ If this coarse gives better results, you can focus on this area.

- use random sampling
- adequate search
- (optionally) consider implementing a coarse to fine search process.

Using an Appropriate Scale to Pick Hyperparameters



10^a $a = \log_{10} 0.0001 = -4$

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

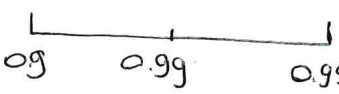
$\alpha = 10^r$! ← $10^{-4} \dots 10^0$

10^b $b = \log_{10} 1 = 0$

$10^a \dots 10^b$ $r \in [a, b]$ $[-4, 0]$

→ It doesn't make sense to sample on the linear scale

$1 - \beta = 0.1 \dots 0.001$



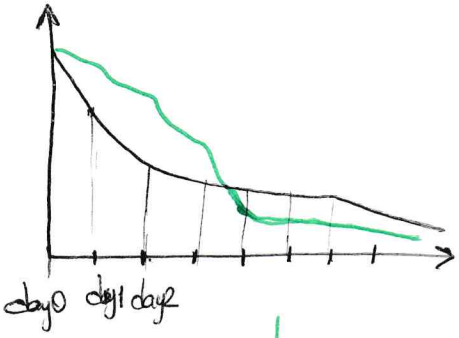
$r \in [-3, -1]$

$1 - \beta = 10^r$

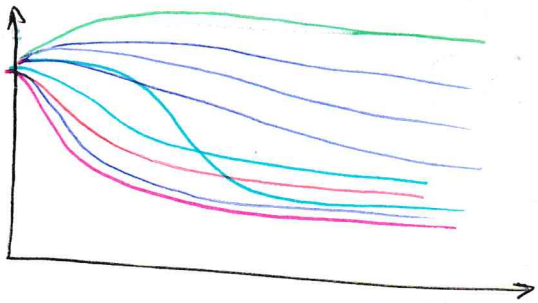
$\beta = 1 - 10^r$

Hyperparameters Tuning in Practice: Pandas vs Caviar

Babysitting One Model



Training Many Models in Parallel



The way to choose between these two approaches is a function of how much computational resources you have;

For example:

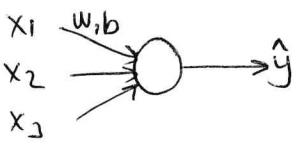
Some online advertising settings as well as in some computer vision applications where there's just so much data, models are so big that it's difficult to train a lot of models at the same time.

If you have enough computers to train a lot of models in parallel

Normalizing Activations in a Network

Batch normalization makes your hyperparameter search problem much easier, makes your neural network much more robust. Also will enable you to much more easily train even very deep networks.

Created by two researchers, Sergey Ioffe and Christian Szegedy.



↳ normalizing x_1, x_2, x_3 maybe helps you train w and b more efficiently

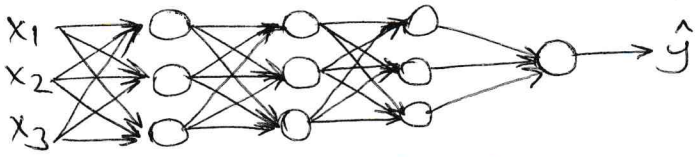
$$\mu = \frac{1}{m}\sum_i x^{(i)}$$

$$X = X - \mu$$

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

$$X = X/\sigma^2$$

Wouldn't it be nice if you can normalize the mean and variance of $a^{[2]}$ to make the training of $w^{[3]}, b^{[3]}$ more efficient?



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

Implementing Batch Norm:

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

1. $\mu = \frac{1}{m}\sum_i z^{(i)}$
2. $\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$
3. $z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$

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

γ, β: learnable parameters of model

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

$\beta = \mu$ is true

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

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

tilda

With the parameters gamma and β, you can make that your $z^{(i)}$ values have the range of values that you want.

What it really does is it normalizes in mean and variance of these hidden units values to have some fixed mean and variance.

could be 0 and 1 or other value

It's controlled by γ and β

Fitting Batch Norm into a Neural Network

$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm. (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow a^{[2]} \longrightarrow \cdots$

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots, W^{[L]}, b^{[L]}$

$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots, \beta^{[L]}, \gamma^{[L]}$ } $d\beta^{[l]}$ $\quad \beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$

this β is different than the β in Adam, RMS-prop or momentum. !

tf.nn_batch-normalization

In practice, Batch norm is usually applied with mini batches of your training set.

Adding any constant to all of the examples in the minibatch, it doesn't change anything. Because any constant you add will get cancelled out by the mean subtracting step. So if you're using batch norm, you can eliminate $b^{[l]}$

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

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

$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$ → is a parameter that controls that ends up affecting the shift or the biased terms.

Parameters: $w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$ → are used to scale the mean and variance of ~~each~~ the hidden units to whatever the networks wants to set them to.

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

Implementing Gradient Descent:

for $t = 1 \ldots$ numMiniBatches

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

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

Use backprop to compute $dW^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$

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

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

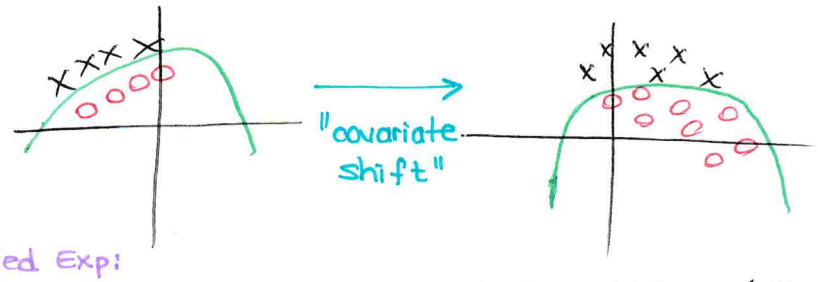
$\gamma^{[l]} := \ldots$

Works for all optimization algorithms

Why Does Batch Norm Work?

One intuition behind why batch norm works is, this is doing the similar thing for the values in your hidden units and not just for your input layer.

∝ Learning on Shifting Input Distribution:



Detailed Exp:

Batch norm limits the amount to which updating the parameters in the earlier layers can affect the distribution of values.

So, batch norm reduces the problem of the input values changing, it really causes these values to become more stable, so that the later layers of the neural network has more firm ground to stand on. And so it allows each layer of network to learn by itself, a little bit more independently of other layers and this is the effect of speeding up of learning in the whole network.

The earlier layers don't get to shift around as much, because they're constrained to have the same mean and variance and so this makes the job of learning on later layers easier.

∝ Batch Norm as Regularization:

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations. (64, 128, 256)
- This has a slight regularization effect. (the thinks unintended affect)

→ μ and σ^2 are noisy

small affect (not huge)
so you might choose to use batch norm together with dropout

Using with dropout makes more powerful regularization affect

By using LARGER mini batch size, you're reducing this noise and therefore also reducing this regularization affect.

mini batch : 64 → 512

Don't turn to batch norm as a regularization.

Use it as a way to normalize your hidden units activations and therefore speed up learning.

Batch Norm At Test Time

Batch norm processes your data one mini batch at a time, but the test time you might need to process the examples one at a time.

TRAIN

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

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

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

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

TEST

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

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

$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]} \longrightarrow \mu$

$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]} \longrightarrow \sigma^2$

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

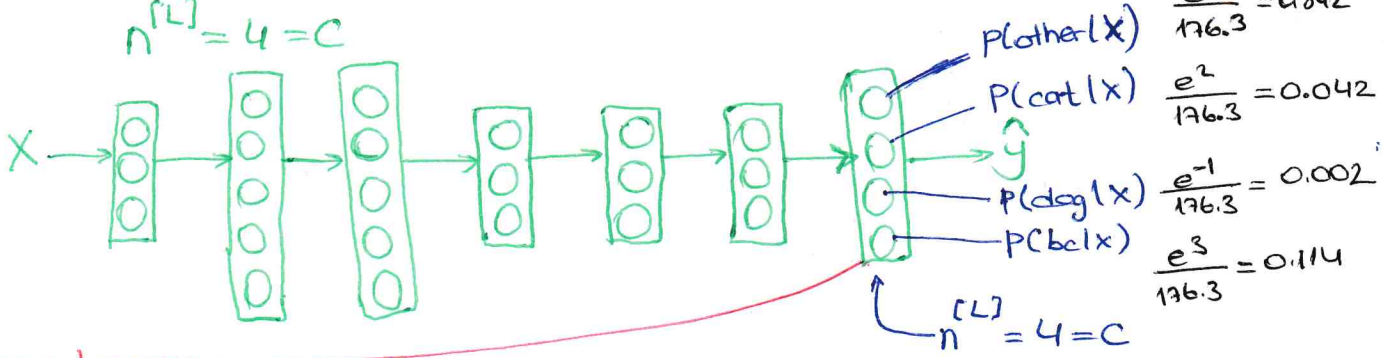
$$\tilde{z} = \gamma z_{norm} + \beta$$

γ, β: Learned during training

Softmax Regression

There's a generalization of logistic regression called Softmax Regression.

C = #classes = 4 (0, ..., 3)



Softmax Layer:

$$z^{[L]} = W^{[L]} \cdot a^{[L-1]} + b^{[L]} \quad \text{dimension} \rightarrow (4,1)$$

Activation Function:

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

temporary variable

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

$\hat{y}$

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

$(4,1)$ $\quad$ $(4,1)$

The unusual thing about the Softmax activation function is it takes an input (4,1) vector, and outputs a (4,1) vector

Ex:

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

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

$$\sum_{j=1}^{4} t_j = 176.3$$

$$a^{[L]} = \frac{t}{176.3}$$

Training a Softmax Classifier

Understanding Softmax

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

"softmax"

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

"hard max"

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

Softmax regression generalizes logistic regression to C classes.

If C = 2, softmax reduces to logistic regression

Loss function

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

$y_2 = 1$

$y_1 = y_3 = y_4 = 0$

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

C = 4

$$-y_2 \log y_2 = -\log \hat{y}_2$$

(make $\hat{y}_2$ big)

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

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

$$\hat{Y} = [\hat{y}^{(1)}, \ldots, \hat{y}^{(m)}] = \begin{bmatrix} 0.3 & \cdots \\ 0.2 & \\ 0.1 & \\ 0.4 & \end{bmatrix}$$

More generally, what this loss function does is it looks at whatever is the ground true class in your training set, it tries to make the corresponding probability of that class as high as possible.

Gradient Descent with Softmax

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

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

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

```
import numpy as np
import tensorflow as tf
w = tf.Variable(0, dtype=tf.float32)
cost = w**2 - 10*w + 25   # tf.add(tf.add(w**2, tf.multiply(-10., w)), 25)
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
                                          # 0.01: learning rate
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
session.run(w)
```

Backprop is already built in add, multiply and square function in Tensorflow.

```
w = tf.Variable(0, dtype=tf.float32)
cost = w**2 -10*w + 25
train = . . . .
init = . . . .
session = . . . .
session.run(train)
```

```
coefficients = np.array([[1.], [-20], [100]])
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]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(train, feed_dict={x:coefficients})
```

7

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

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