

Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

Week 1

Setting up your Machine Learning Application

Train/dev/test

Applied ML is highly iterative process

- layers
- hidden units
- learning rates
- activation functions

Idea -> Code -> Experiment -> Idea...

Train/dev/test sets

- Previous era 70/20/10 splitting
- Modern era, million of examples 98/1/1 is more reasonable

Mismatched train/test distribution

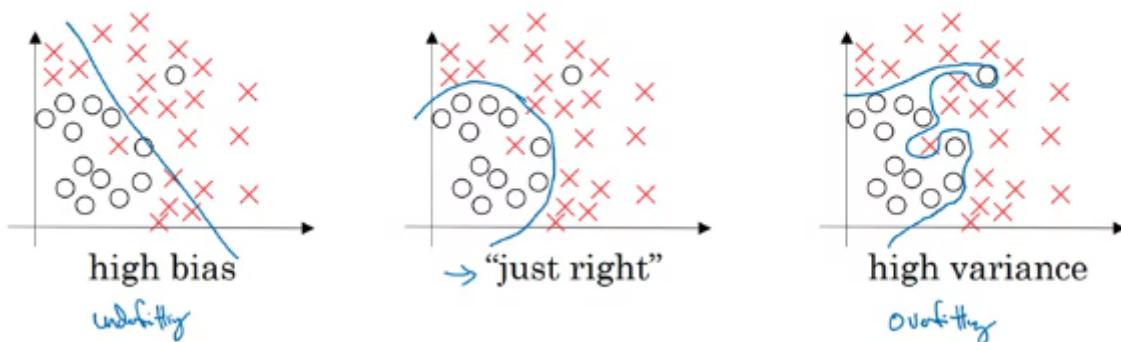
Training set: Cat from webpages Dev/test set: Cat pictures from users

-> Make sure that train/test come from the same distribution

Bias / Variance

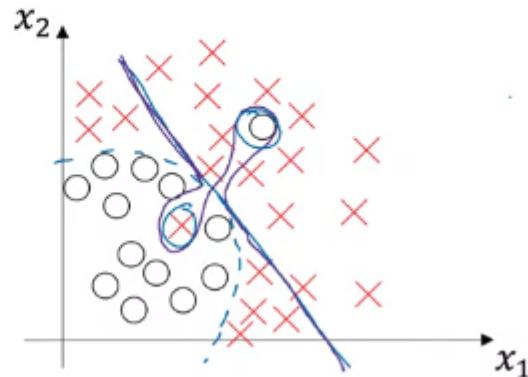
 Share

Bias and Variance



train/dev error: %1/%11 -> high variance
 trian/dev error: 15%/16% -> high bias (human level is around %0-1 error)
 trian/dev error: 15%/30% -> high bias and high variance

High bias and high variance



train/dev error: 0.5%/1% -> low/low

All under the assumption that base error is quite small and same distribution of train/dev

Basic Recipe for ML

High bias?

Try:

- bigger network
- Train longer
- Different archs

High Variance?

Try:

- More data
- Regularization
- Different archs

In the past it was a tradeoff, in the modern big dat era you can deal both without hurting the other.

Regularization your NN

Regularization

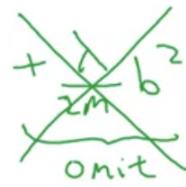
Logistic regression

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

$$\underline{w \in \mathbb{R}^n}, \underline{b \in \mathbb{R}}$$

λ = regularization parameter
 lambda lambd.

$$J(w, b) = \underbrace{\frac{1}{m} \sum_{i=1}^m l(y^{(i)}, \hat{y}^{(i)})}_{L_2 \text{ regularization}} + \frac{\lambda}{2m} \|w\|_2^2$$



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

$$L_1 \text{ regularization} \quad \frac{\lambda}{2m} \sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m} \|w\|_1$$

w will be sparse

- L2 is the most common regularization
- Usually doesn't regulate bias which is low dim
- L1 regularization cause the sparse w (many zeros) can be used to compress model
- lambda = regularization parameter

Neural network

$$J(w^{(0)}, b^{(0)}, \dots, w^{(L)}, b^{(L)}) = \underbrace{\frac{1}{m} \sum_{i=1}^m l(y^{(i)}, \hat{y}^{(i)})}_{\text{loss function}} + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{(l)}\|_F^2$$

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

$$w: (n^{(l+1)} \ n^{(l)}).$$

$$\begin{matrix} \text{"Frobenius norm"} & \| \cdot \|_2^2 \\ \downarrow & \uparrow \\ \| \cdot \|_F^2 \end{matrix}$$

$$\begin{aligned} \frac{\partial J}{\partial w^{(l)}} &= (\text{from backprop}) + \frac{\lambda}{m} w^{(l)} \\ \rightarrow w^{(l)} &:= w^{(l)} - \alpha \frac{\partial J}{\partial w^{(l)}} \end{aligned}$$

$$\frac{\partial J}{\partial w^{(l)}} = \frac{\partial J}{\partial w^{(l)}}$$

"Weight decay"

$$\begin{aligned} w^{(l)} &:= w^{(l)} - \alpha \left[(\text{from backprop}) + \frac{\lambda}{m} w^{(l)} \right] \\ (1 - \frac{\alpha \lambda}{m}) w^{(l)} &= w^{(l)} - \alpha \left[(\text{from backprop}) + \frac{\lambda}{m} w^{(l)} \right] \end{aligned}$$

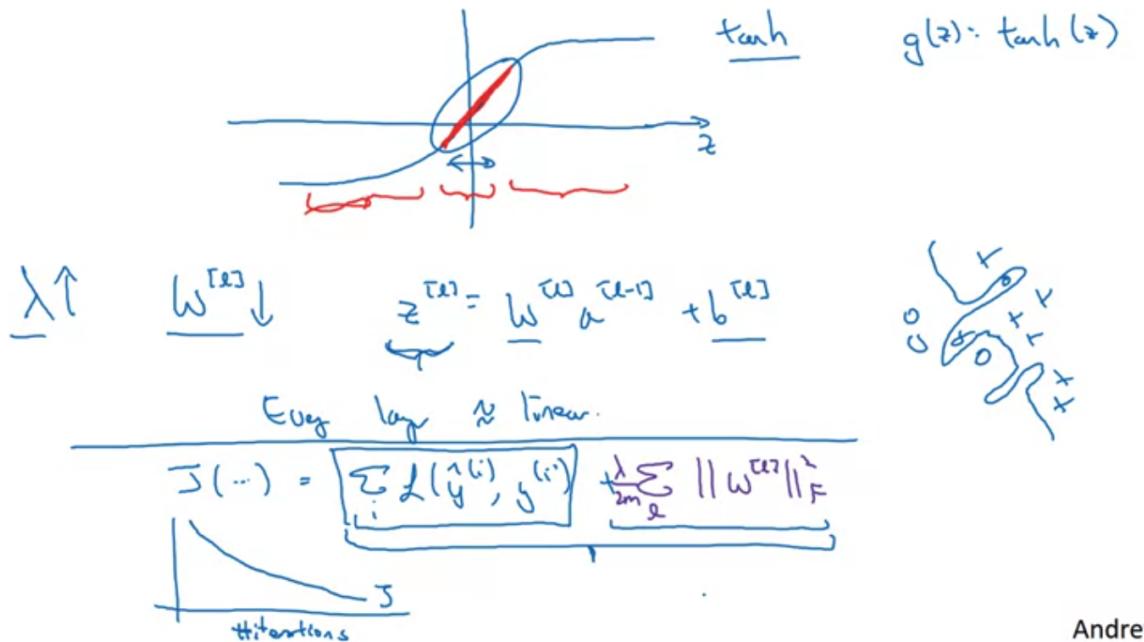
Andrew Ng

- backprop changed with extra regularization term
- L2 called "Weight Decay", see bottom of slide to understand why

Why regularization reduces overfitting?

- Intuition: many parameters close to zero with no impact and cause to simpler network

How does regularization prevent overfitting?



Andrew Ng

- tanh activation: small values of W cause linear output and prevent overfitting
- Tip: when plotting J over iteration for debug don't forget the regularization term (else might not see monotonically decrease)

Dropout regularization

- drop several nodes connection on each forward stage
- less nodes active on each such iteration and because of that eliminate overfitting

Inverted dropout

Implementing dropout ("Inverted dropout")

Illustrate with layer $l=3$. $\text{keep-prob} = \frac{0.8}{x}$ $\underline{0.2}$

$$\rightarrow d_3 = \text{np.random.rand}(a_3.shape[0], a_3.shape[1]) < \text{keep-prob}$$

$$a_3 = \text{np.multiply}(a_3, d_3) \quad \# a_3 * d_3.$$

$$\rightarrow a_3 /= \cancel{\text{keep-prob}} \leftarrow$$

\uparrow 50 units. \rightsquigarrow 10 units shut off

$$z^{[4]} = w^{[4]} \cdot \frac{a^{[3]}}{\cancel{10}} + b^{[4]}$$

$\cancel{10}$ reduced by $\underline{20\%}$.

Test

$$/ = \underline{0.8}$$

Making prediction at test time

- No dropout (will cause noise)
- do not divide by keep_prob on test (only train), else will get increased weight

Why does drop-out work?

- Intuition: single unit can't rely on any one feature, so have to spread out weights -> shrinking weights
- it is possible to provide different keep_prob per layer (bigger layers needs smaller keep_prob value)
- to input layer recommended to be with keep_prob = 1 or very high value such 0.9

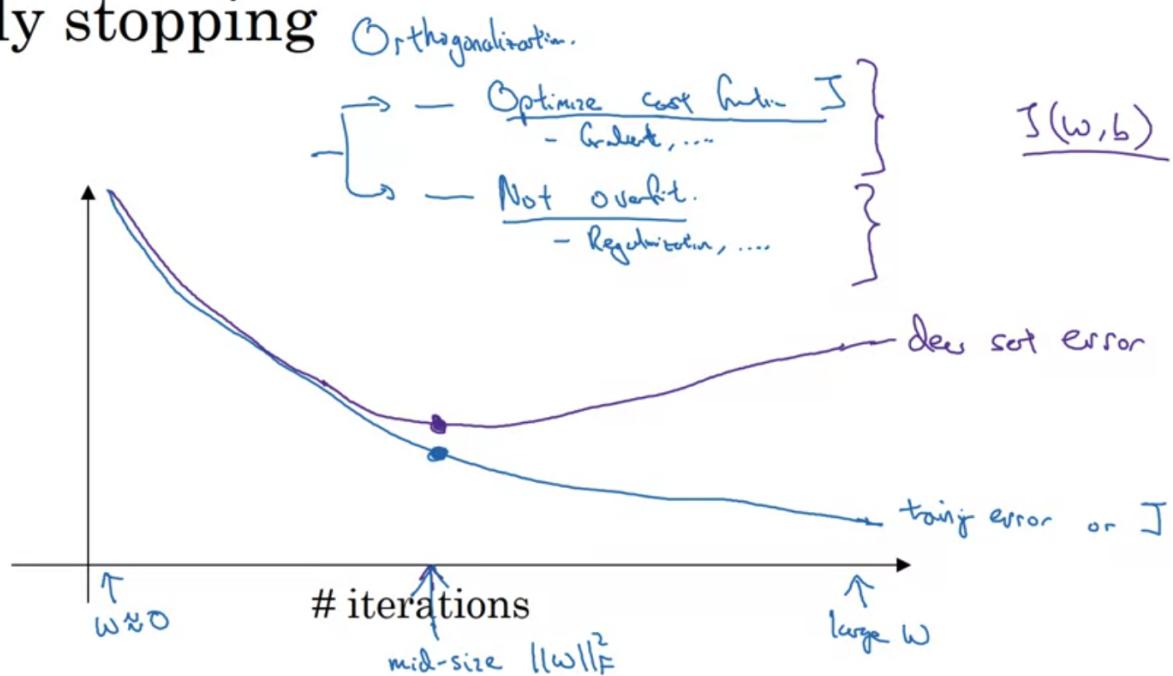
Other regularization method

Data augmentation

for example flip, crop, rotate, zooming image train inputs

Early stopping

Early stopping



- plots dev/train set error over iterations.
- find iteration that the errors starts to move away from each other and train again up to that point.
- Andrew doesn't recommend because you also stop J to arrive its optimal value

Setting up your optimization problem

Normalizing inputs

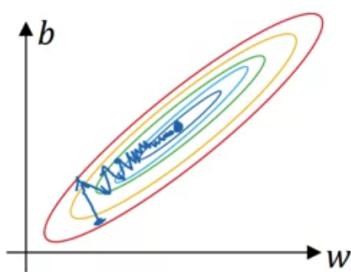
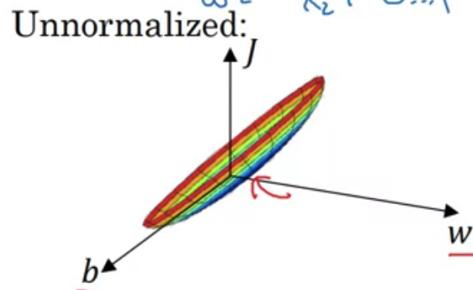
Steps

- Subtract the mean
- Normalize the variance

Why normalize inputs?

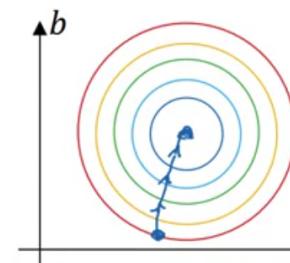
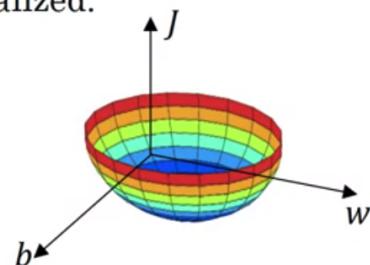
Why normalize inputs?

$w_1 \quad x_1: 1 \dots 1000$
 $w_2 \quad x_2: 0 \dots 1$



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

Normalized:

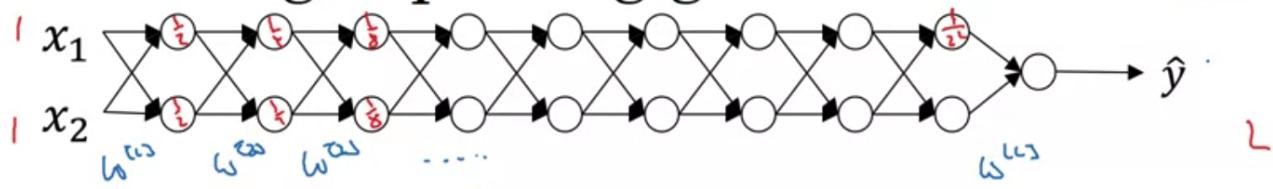


Andrew Ng

Vanishing / Exploding gradients

Vanishing/exploding gradients

$L=150$



$$\frac{\partial g(z)}{\partial z} \cdot z \cdot \frac{\partial b^{[L]}}{\partial z}$$

$$\hat{y} = w^{[L]} \underbrace{w^{[L-1]} \circ w^{[L-2]} \circ \dots \circ}_{\text{Input}} \underbrace{w^{[1]} \circ w^{[2]} \circ \dots \circ}_{\text{Input}} x$$

$$w^{[1]} > I$$

$$w^{[2]} < I \quad [0.9 \quad 0.9]$$

$$w^{[2]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \\ 0 & 0 \end{bmatrix}$$

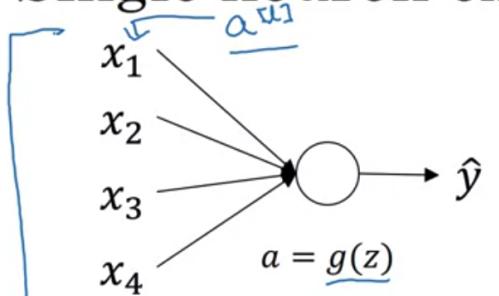
$$\begin{aligned} z^{[1]} &= w^{[1]} x \\ a^{[1]} &= g(z^{[1]}) = z^{[1]} \\ a^{[2]} &= g(z^{[2]}) = g(w^{[2]} a^{[1]}) \\ z^{[2]} &= w^{[2]} x \\ a^{[3]} &= g(z^{[3]}) = g(w^{[3]} a^{[2]}) \\ &\vdots \\ \hat{y} &= w^{[L]} \underbrace{\begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \\ 0 & 0 \end{bmatrix}^{L-1} \circ}_{\text{Input}} \underbrace{w^{[1]} \circ w^{[2]} \circ \dots \circ}_{\text{Input}} x \end{aligned}$$

1.5^{L-1} x
0.5^{L-1} x

Deep networks activation might explodes/vanished because of exponential increase/decrease

Weight initialization for DNN

Single neuron example

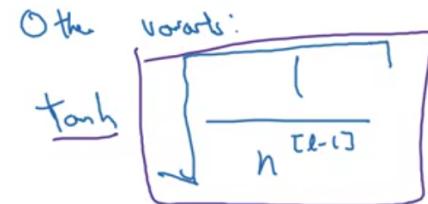


$$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n \quad \cancel{\text{X}}$$

Large $n \rightarrow$ Smaller w_i

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

$$w^{[1]} = \text{np.random.rand}(\dots) * \sqrt{\frac{2}{n^{[1-1]}}} \quad \text{ReLU} \quad g^{[1]}(z) = \text{ReLU}(z)$$



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

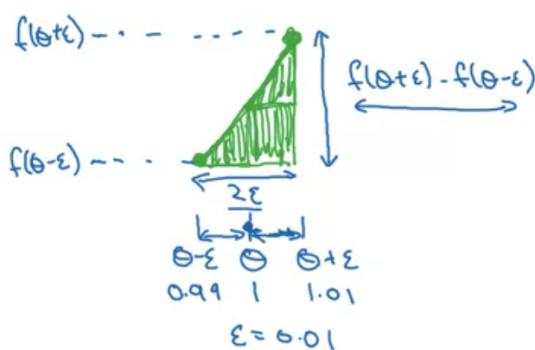
 \uparrow

- fixing initialization to consider fan in and normalize the sum of weights to gaussian with mean 0 and variance 1 reduce the problem
- nomarator 2 proposed for Reuly, for tanh it is better to use nomarator 1 (Xavier)

Numerical approximation of gradients

Checking your derivative computation

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



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

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

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

Approx error: 0.0001
(prev slide: 3.0301, error: 0.03)

$$f'(\theta) = \lim_{\epsilon \rightarrow 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$$

$$\frac{O(\epsilon^2)}{0.01} = 0.0001$$

$$\frac{f(\theta+\epsilon) - f(\theta)}{\epsilon} \quad \text{error: } O(\epsilon)$$

Andr

When implementing back propagation it usefull to make sure gradients calculation implemented correctly by comparing to numerical calculated gradients

Gradient checking

- take all parameters and reshape to one vector theta
- take all derivatives and reshape to one vector dtheta
- for each index of the vector: $d\theta_{approx}[i] = J(\theta_1 \dots \theta_{i-1}, \theta_i + \epsilon, \theta_{i+1} \dots) - J(\theta_1 \dots \theta_{i-1}, \theta_i - \epsilon, \theta_{i+1} \dots)$
- check $(|d\theta_{approx} - d\theta|) / (|d\theta| + |d\theta_{approx}|) < \text{maximum_epsolin}$

Gradient checking implementation notes

- Don't use in training - only for debug (very slow)
- If grad check fail, look at specific components to try to identify bug.
- Don't forget the regularization term
- Doesn't work with dropout, at least not straight forward. can turn off and check other logic and then turn on (without validate dropout)
- There are cases that implementation is correct only for small parameters w/b around zero, and doesn't work for big numbers, so run the grad checks also after few iterations (when few parameters will be larger)

Week 2

Optimization algorithms

Mini-batch vs. gradient descent

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \mathbf{x}^{(3)} & \dots & \mathbf{x}^{(1000)} & \dots & \mathbf{x}^{(2000)} & \dots & \mathbf{x}^{(5000)} & \dots & \mathbf{x}^{(m)} \end{bmatrix}_{(n_x, m)}$$

$$\mathbf{Y} = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(1000)} & \dots & y^{(2000)} & \dots & y^{(5000)} & \dots & y^{(m)} \end{bmatrix}_{(1, m)}$$

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

5,000 mini-batches of 1,000 each

Mini-batch t: $\underline{\mathbf{x}^{(t)}, \mathbf{y}^{(t)}}$

$$\begin{cases} \mathbf{x}^{(i)} \\ \mathbf{z}^{[l]} \\ \mathbf{x}^{(t)}, \mathbf{y}^{(t)} \end{cases}$$

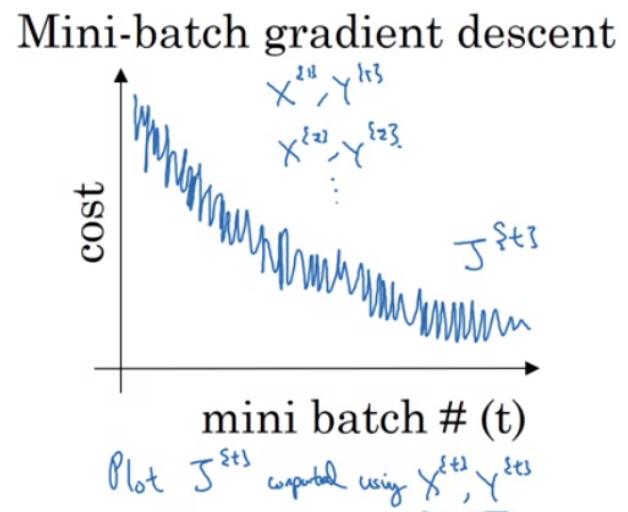
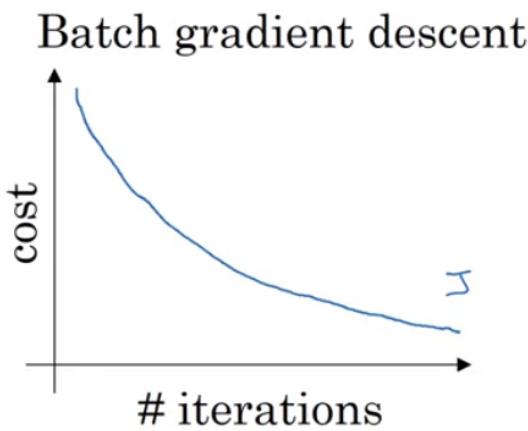
Andrew Ng

- regular implementation of gradient descent that calculate step after going over all M samples is very slow on big data.
- Split the data to mini-batches and update parameters after each mini-batch processing
- Still user vectorized implementation on each mini-batch

- replace m on the formula to the batch size
- "1 epoch" - pass through whole training set

Understanding mini-batch gradient descent

Training with mini batch gradient descent

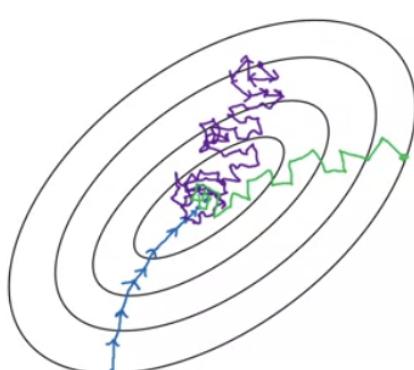


- Cost may not decrease on each iteration because it train on different samples batch each time

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 $(X^{(1)}, Y^{(1)}) = (x^{(1)}, y^{(1)}) \dots (x^{(n)}, y^{(n)})$ mini-batch.

In practice: Somewhat in-between 1 and m



Stochastic
gradient
descent

{

Use sparingly
for vectorization

In-between
(minibatch size
not too big/small)

Faster learning.

- Vectorization.
($n > 1000$)
- Make passes without
processing entire training set.

Batch
gradient descent
(minibatch size = m)

↓

Two long
per iteration

Andrew Ng

- if mini-batch size = m → same as regular gradient descent. very smooth with low noise and large steps. Too long per iteration
- if minibatch size = 1 → stochastic gradient descent (SGD). extremely noisy and sometime goes in the wrong direction. will never converge to final point. loose vectorization

- few thousands mini-batch size provides advantage of both extremes

Choosing your mini-batch size

- Use batch gradient descent for small training set (few thousands)
- Typical mini-batch size: 64, 128, 256, 512... (try to use power of 2)
- Make sure minibatch fits in cpu/gpu memory

Exponentially weighted averages

Temperature in London

$$\theta_1 = 40^\circ\text{F} \quad 4^\circ\text{C} \leftarrow$$

$$\theta_2 = 49^\circ\text{F} \quad 9^\circ\text{C}$$

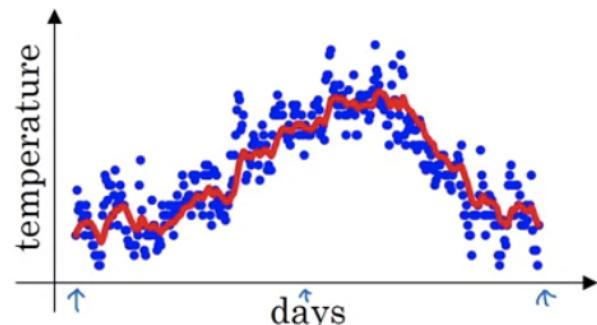
$$\theta_3 = 45^\circ\text{F} \quad \vdots$$

$$\vdots$$

$$\theta_{180} = 60^\circ\text{F} \quad 15^\circ\text{C}$$

$$\theta_{181} = 56^\circ\text{F} \quad \vdots$$

$$\vdots$$



$$V_0 = 0$$

$$V_1 = 0.9 V_0 + 0.1 \theta_1$$

$$V_2 = 0.9 V_1 + 0.1 \theta_2$$

$$V_3 = 0.9 V_2 + 0.1 \theta_3$$

$$\vdots$$

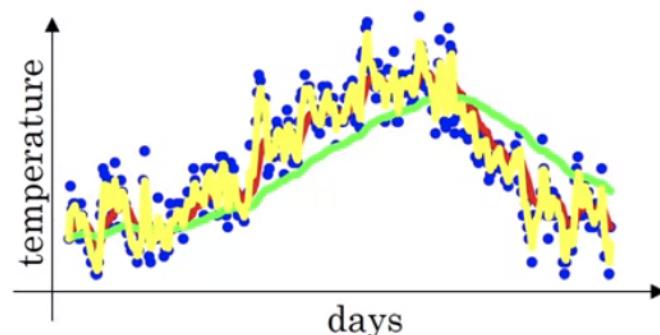
$$V_t = 0.9 V_{t-1} + 0.1 \theta_t$$

Exponentially weighted averages

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

$\beta = 0.9$: ≈ 10 day's temperature
 $\beta = 0.98$: ≈ 50 days
 $\beta = 0.5$: ≈ 2 days

V_t is approximately
average over
 $\rightarrow \approx \frac{1}{1-\beta}$ days'
temperature.



$$\frac{1}{1-0.98} = 50$$

Understanding exponentially weighted averages

Exponentially weighted averages

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

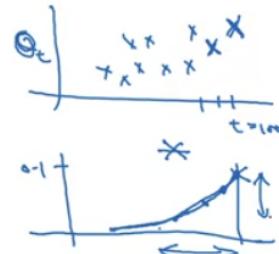
$$v_{100} = 0.9 v_{99} + 0.1 \theta_{100}$$

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

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

...

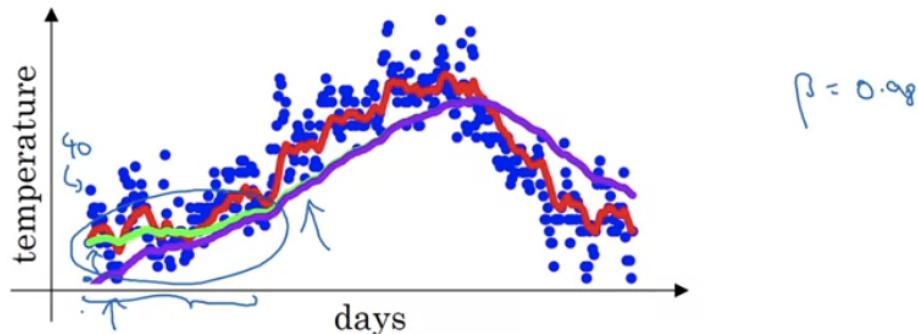
$$\begin{aligned} v_{100} &= 0.1 \theta_{100} + 0.9 \cancel{(0.1 \theta_{99})} + 0.9 \cancel{(0.1 \theta_{98})} + \dots \\ &= 0.1 \theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1 (0.9)^2 \theta_{98} + 0.1 (0.9)^3 \theta_{97} + 0.1 (0.9)^4 \theta_{96} \\ 0.9^{10} &\approx 0.35 \approx \frac{1}{e} \quad \frac{(1-\epsilon)^{1/\epsilon}}{\epsilon} = \frac{1}{e} \end{aligned}$$



- After $1/\epsilon$ days coeff $\approx 1/e$, roughly how many days actually affect the average
- usually just save the last value V_θ and update it every day

Bias correction in exponentially weighted averages

Bias correction



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

$$V_0 = 0$$

$$V_1 = 0.98 V_0 + 0.02 \theta_1$$

$$V_2 = 0.98 V_1 + 0.02 \theta_2$$

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

$$= 0.0196 \theta_1 + 0.02 \theta_2$$

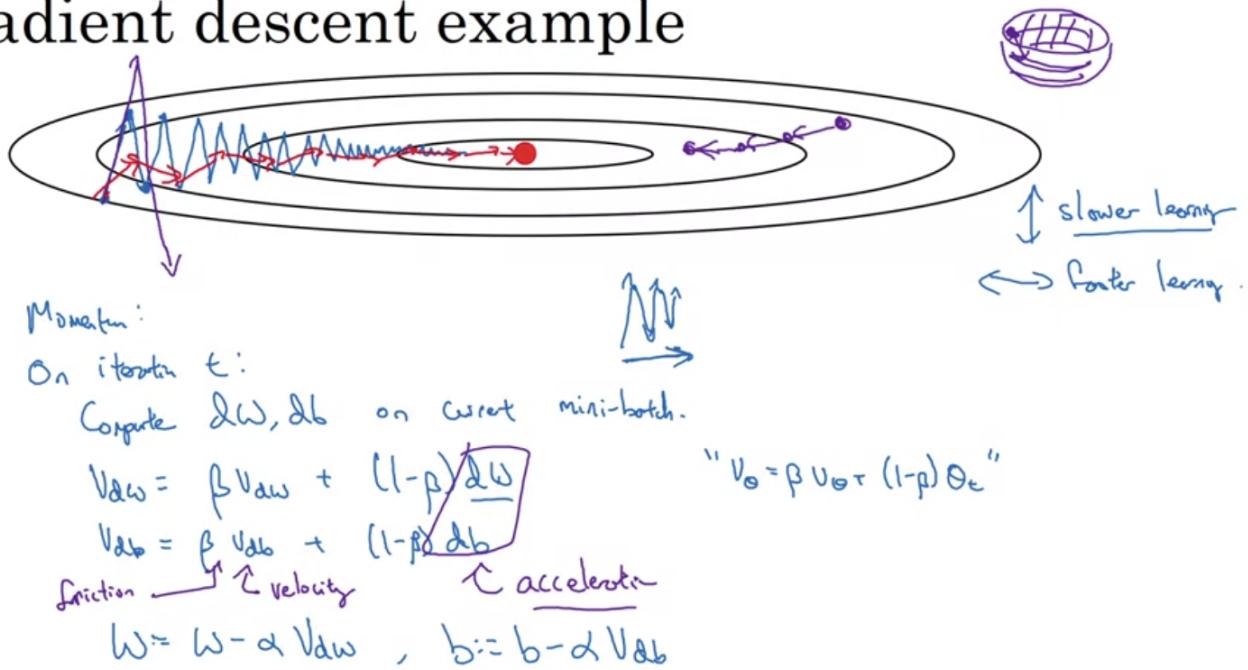
$$\left| \begin{array}{l} \frac{V_t}{1 - \beta^t} \\ t=2: 1 - \beta^t = 1 - (0.98)^2 = 0.0396 \\ \frac{V_2}{0.0396} = \frac{0.0196 \theta_1 + 0.02 \theta_2}{0.0396} \end{array} \right.$$

Andrew Ng

- needs to correct the initial phase values which doesn't have enough history. cause coefficients to sum up to 1.

Gradient descent with momentum

Gradient descent example



- Wants to eliminate oscillations that slow convergence
- The idea is to compute exponentially weighted average of the mini-batches derivatives and use it instead of the mini-batch derivatives to update parameters.
- Vertical direction average will be close to zero. Horizontal direction average will increase.
- Nice analogy to acceleration and velocity of ball rolling down a bowl.

Implementation details

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

On iteration t :

Compute dW, db on the current mini-batch

$$\begin{aligned} \rightarrow v_{dw} &= \beta v_{dw} + (1-\beta) \frac{dw}{db} \\ \rightarrow v_{db} &= \beta v_{db} + (1-\beta) \frac{db}{dw} \end{aligned} \quad \left| \begin{array}{l} \overbrace{v_{dw} = \beta v_{dw} + \frac{dw}{db}}^{\text{average over last } \approx 10 \text{ gradients}} \\ \overbrace{(1-\beta)}^{\cancel{1-\beta t}} \end{array} \right. \quad \begin{aligned} W &= W - \underbrace{\alpha v_{dw}}_{\cancel{1-\beta t}} \\ b &= b - \underbrace{\alpha v_{db}}_{\cancel{1-\beta t}} \end{aligned}$$

Hyperparameters: α, β

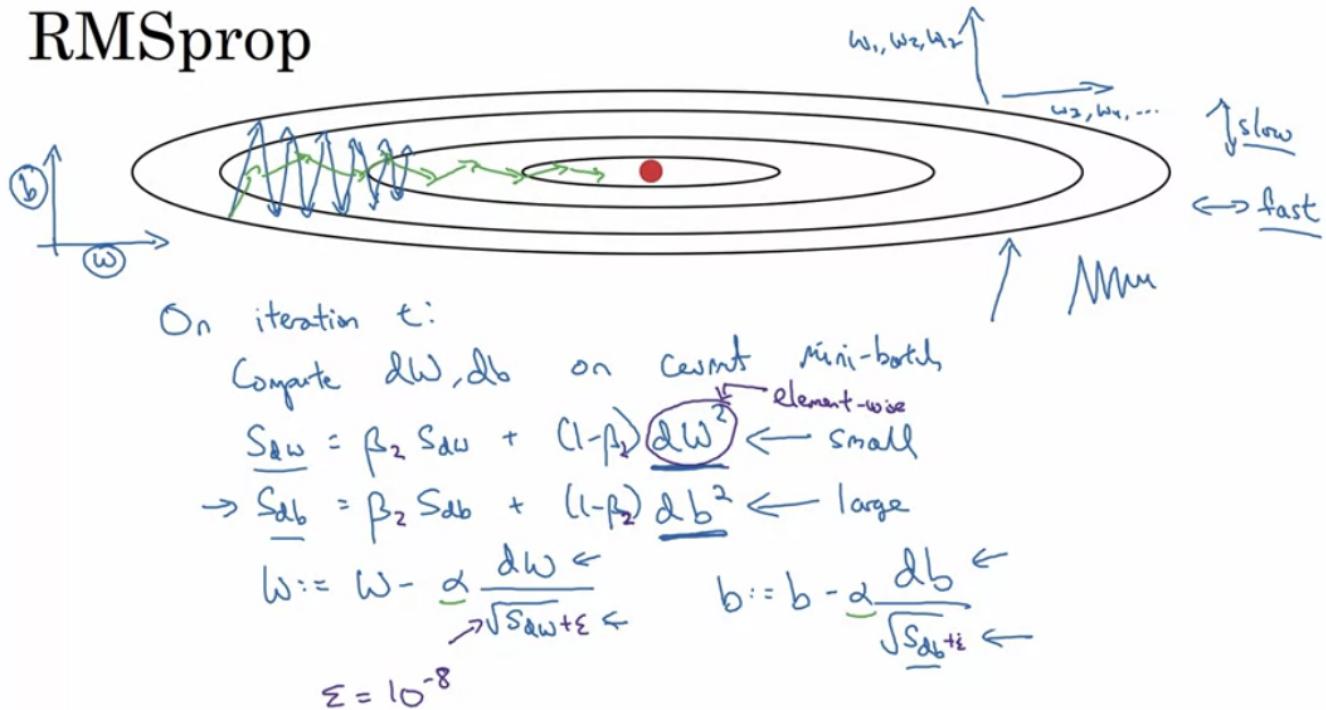
$$\beta = 0.9$$

average over last ≈ 10 gradients

- How do you calculate V_{dw}, V_{db} ?
- What hyperparameters do you have?
- Bias correction: in practice doesn't do it because after 10 iteration (for beta=0.9) there is no longer bias.

- sometimes using formulation without (1-beta) which is less intuitive. beta affect scaling of the new derivatives as well.

RMSprop



- What is the intuition to divide with the exponential weighted averages of elementwise square root derivatives? reduce oscillations on the vertical direction
- as results, can increase alpha
- Why do we add small value epsilon to the denominator? to ensure numerical stability and eliminate divide by ~zero.

Adam optimization algorithm

Adam optimization algorithm

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

On iteration t :

Compute $\delta w, \delta b$ using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) \delta w, \quad V_{db} = \beta_1 V_{db} + (1 - \beta_1) \delta b \quad \leftarrow \text{"moment" } \beta_1$$

$$S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) \delta w^2, \quad S_{db} = \beta_2 S_{db} + (1 - \beta_2) \delta b^2 \quad \leftarrow \text{"RMSprop" } \beta_2$$

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

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

$$w := w - \alpha \frac{V_{dw}^{\text{corrected}}}{\sqrt{S_{dw}^{\text{corrected}}} + \epsilon}$$

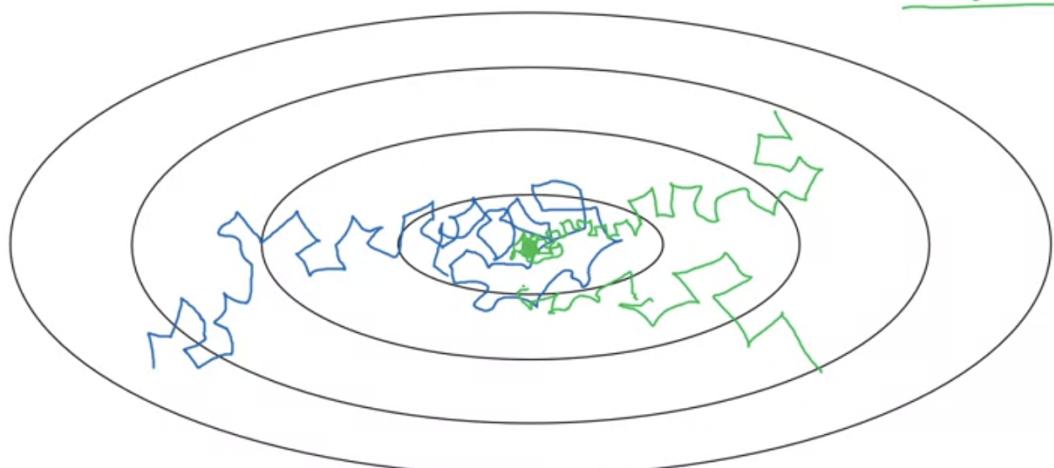
$$b := b - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}}} + \epsilon}$$

- (slides missing square of db on the rmsprop expression)
- What two other methods combined in Adam optimization? momentum and rmsprop
- In typical Adam implementation we do perform bias correction
- What hyperparameters do we have ? alpha, beta_1, beta_2, epsilon
- In practice most of the times try range of alpha and doesn't change the others
- what hyperparameters recommended by the paper: beta_1=0.9, beta_2=0.999, epsilon=10E-8
- Where Adam come from: Adaptive moment estimation

Learning rate decay

Learning rate decay

Slowly reduce α



- What is the intuition? when you far away from the minima you want large steps, but when approaching the minima you want to reduce step size in order to prevent oscillation around it.

Learning rate decay

1 epoch = 1 pass through data.

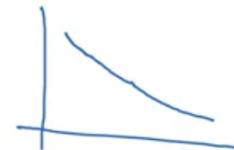
$$\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch-num}} \alpha_0$$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
:	i



$$\alpha_0 = 0.2$$

$$\text{decay.rate} = 1$$



- what hyper parameters do we have? decay_rate and alpha_zero
- slides alpha numbers are wrong (0.67 instead 0.067 etc.)

Other learning rate decay methods

Other learning rate decay methods

forwards

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

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

discrete staircase

Manual decay.

The problem of local optima

- Most points of zero gradient on high dimensional space are saddle points and not local optima.
- A lot of the intuition about lower dimension spaces don't transfer to higher dimension spaces.
- So local optima are not really problem. what is the problem? plateaus.

to summarize:

- unlikely to get stuck in a bad local optima
- plateaus can make learning slow

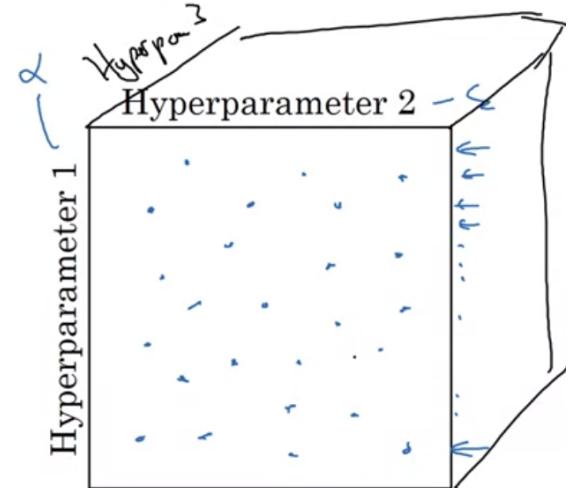
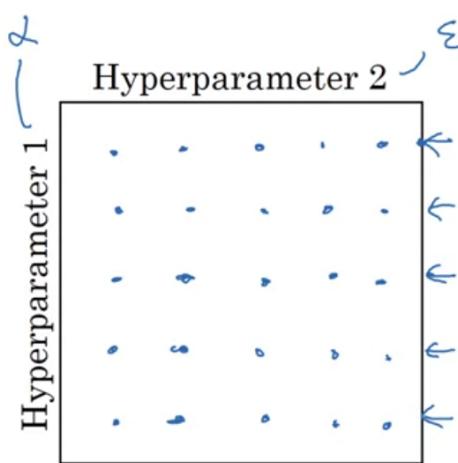
Week 3

Hyperparameter tuning

Tuning process

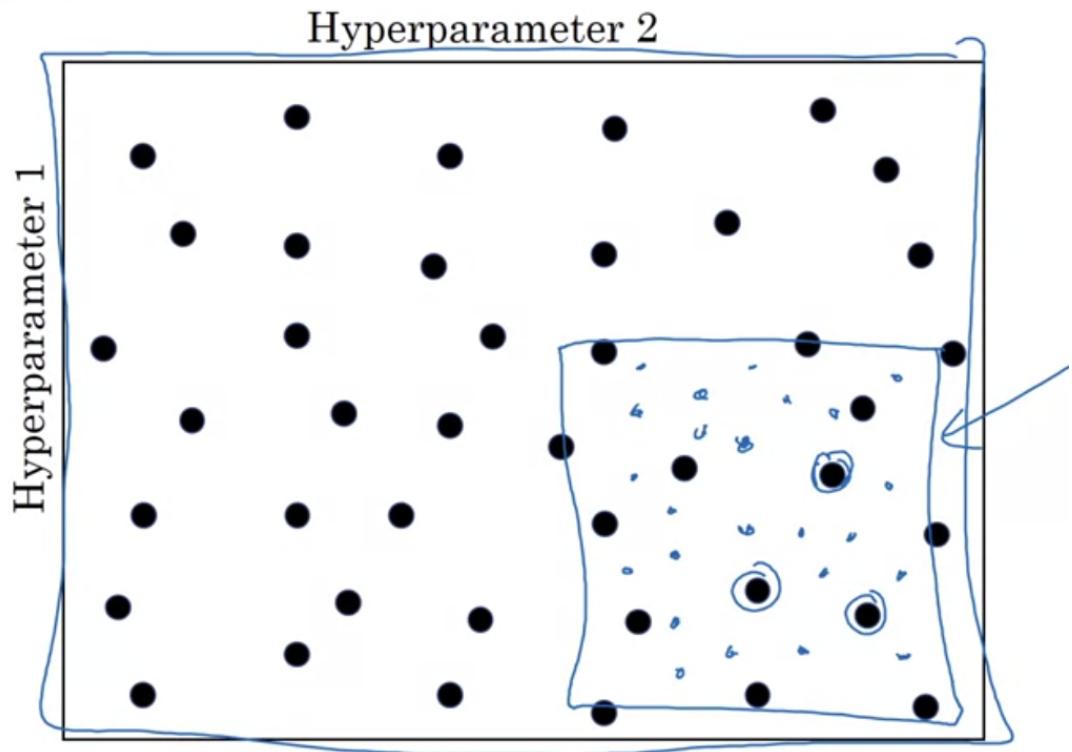
- DNN requires many hyper parameter tuning
- which are the most important? (according to Andrew)
 1. learning rate
 2. momentum beta, hidden units, mini-batch size
 3. number of layers, learning rate decay
- Andrew almost never tunes adam betas and epsilon

Try random values: Don't use a grid



- Why to use random search and not to use grid search? don't waste compute power/time on hyperparameters that almost doesn't affect (for example epsilon).

Coarse to fine

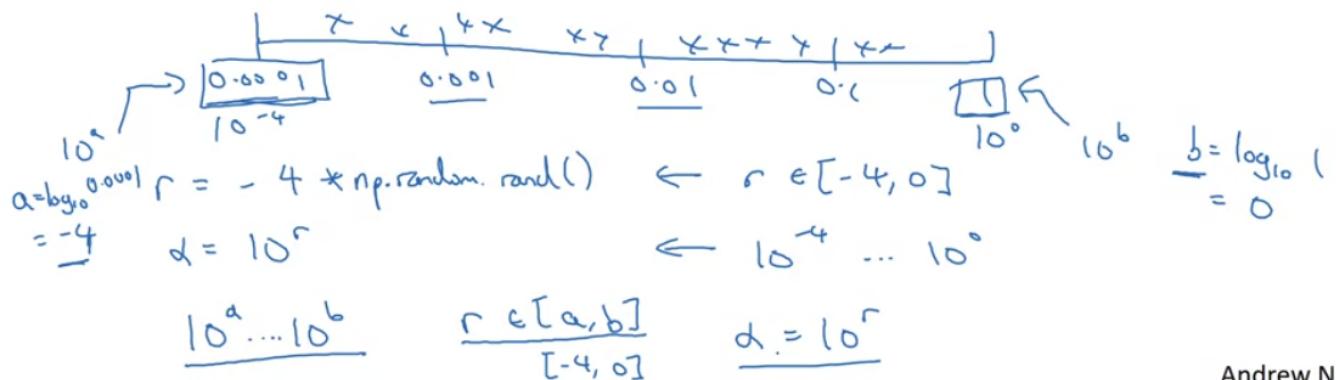
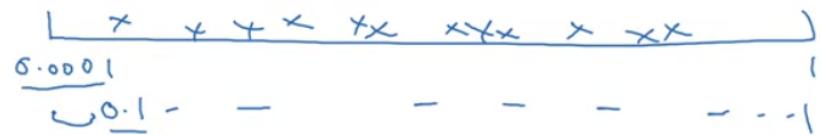


- In coarse to fine you zoom in to limited region after finding that this area provides better results

Using an appropriate scale to pick hyperparameters

Appropriate scale for hyperparameters

$$\alpha = 0.0001, \dots, 1$$

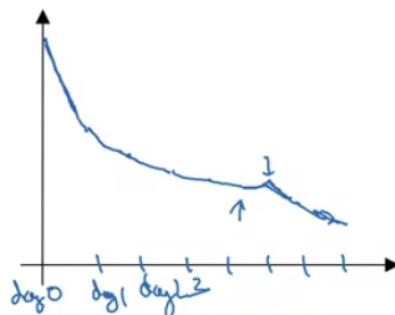


Andrew Ng

- Uniform sampling for hyper parameter tuning are not always reasonable
- learning_rate: better to sample uniformly from the log scale.
- beta: sampling uniformly from log-scale of for (1-beta)

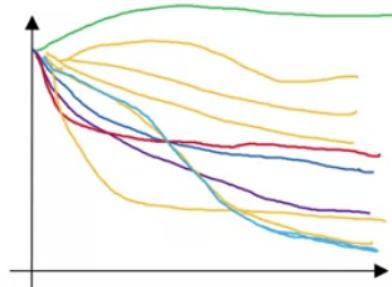
Hyperparameters tuing in practice: Pandas vs. Caviar

Babysitting one model



Panda ↪

Training many models in parallel



Caviar ↪

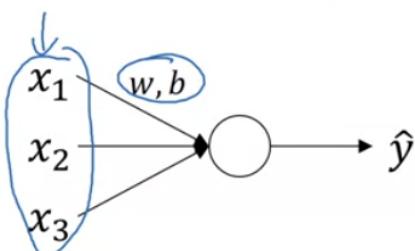
Andrew Ng

- Babysitting one model: running one model, suite for limited resource environment. train -> evaluate -> train again after changing few parameters and again... maybe change the model itself and run again etc.
- Training many models in parallel: needs enough computers.

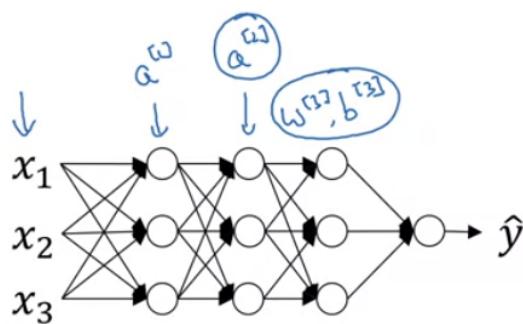
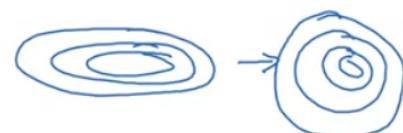
Batch Normalization

Normalizing activations in a network

Normalizing inputs to speed up learning



$$\begin{aligned}\mu &= \frac{1}{m} \sum_i x^{(i)} \\ X &= X - \mu \quad \text{← element-wise} \\ \sigma^2 &= \frac{1}{m} \sum_i (x^{(i)} - \mu)^2 \\ Z &= X / \sigma^2\end{aligned}$$



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

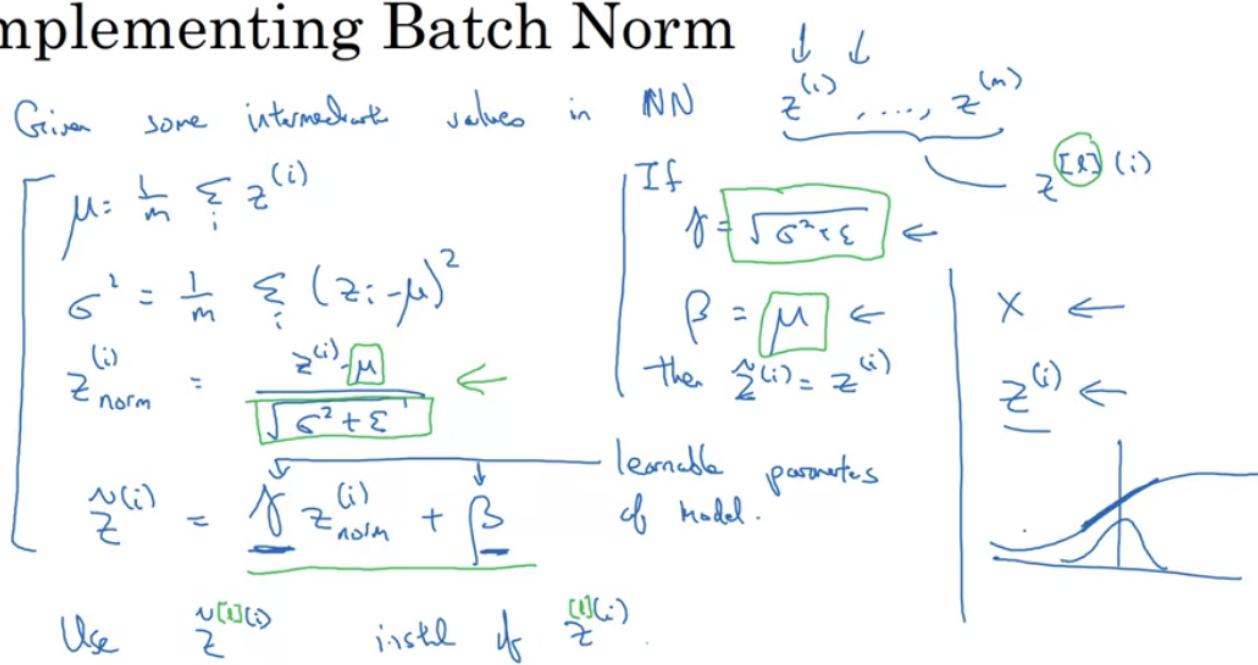
$$\text{Normalize } \frac{z^{[2]}}{\uparrow}$$

Andrew Ng

- In the same manner that we normalize NN inputs in order to make optimization faster, we would like to normalize intermediate results between layers

- There is a debate on the literature if you should normalize before or after activation. In practice pre activation is done more often.

Implementing Batch Norm

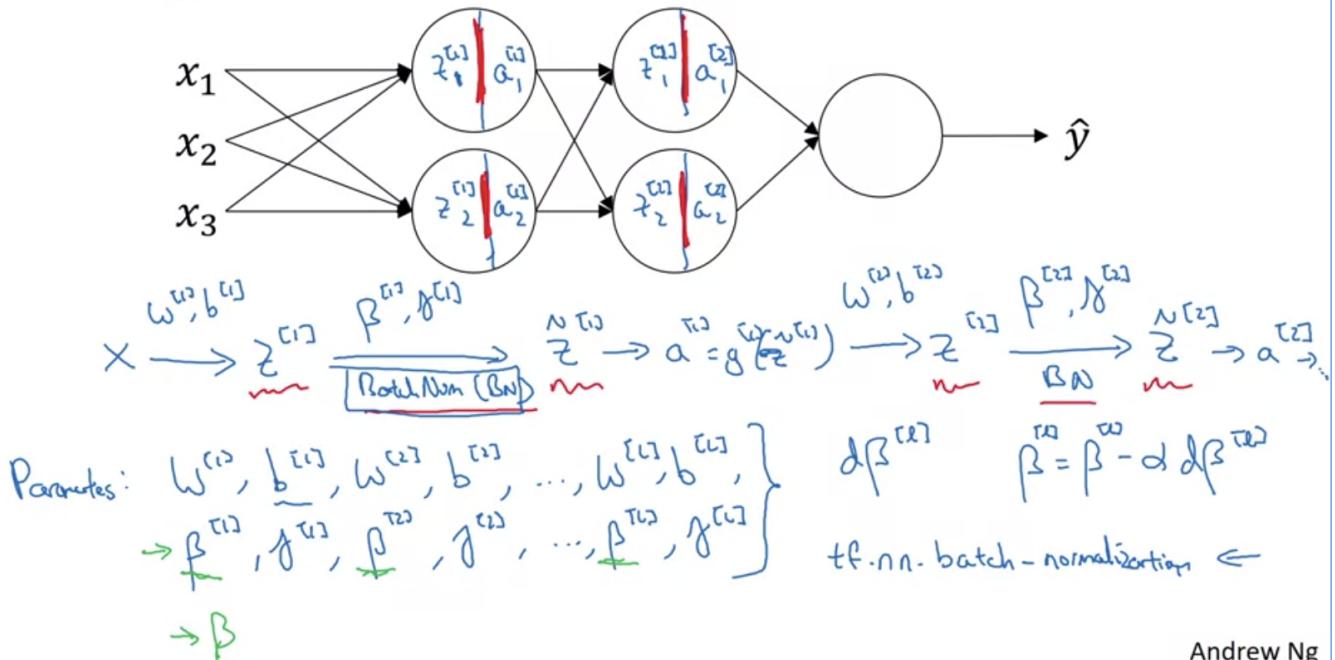


Andrew Ng

- Why do we add epsilon to the norm calculation? stability in case of variance zero
- We don't want always normal(0,1) hence adding to parameters gamma, beta (not hyperparameter)
- What values of gamma and beta will cancel normalization?

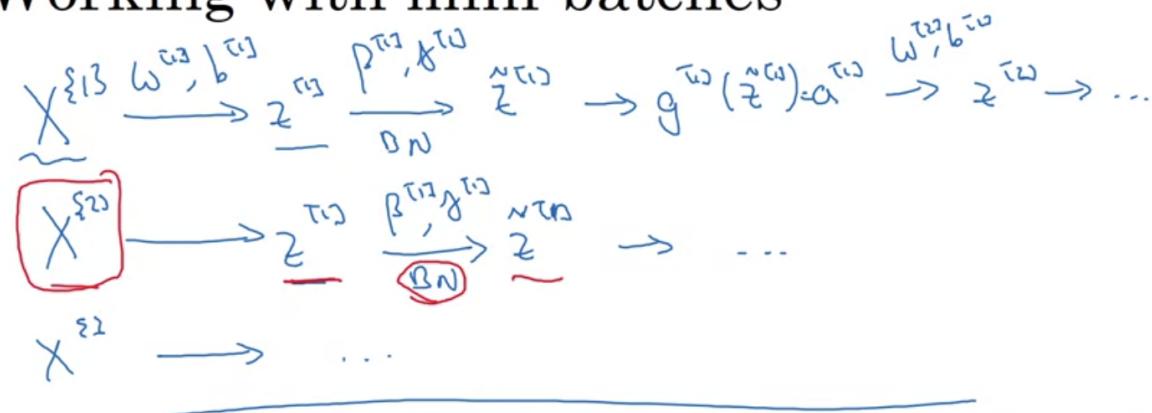
Fitting Batch Norm into a NN

Adding Batch Norm to a network



- beta and gamma vector parameters added to each layer
- can update the batch parameters in the same way we optimize the NN parameters

Working with mini-batches



Parameters: $\{w^{[t]}, \cancel{b^{[t]}}, \beta^{[t]}, \gamma^{[t]}\}$

$$\underline{z}^{[t]} \\ (\underline{n}^{[t]}, 1)$$

$$\begin{aligned} \underline{z}^{[t]} &= w^{[t]} \underline{a}^{[t-1]} + \cancel{b^{[t]}} \\ \underline{z}^{[t]} &= w^{[t]} \underline{a}^{[t-1]} \\ \underline{z}^{[t]}_{\text{norm}} &= \gamma^{[t]} \underline{z}^{[t]}_{\text{norm}} + \beta^{[t]} \end{aligned}$$

Andrew Ng

- In practice, working with mini-batches
- Each mini-batch used only its batch data statistics
- We no longer need the bias parameter b , because we cancel it on the normalization process and then add the normalization batch beta as bias
- the dimension of beta gamma for each layer is (#cells at layer, 1)

Implementing gradient descent

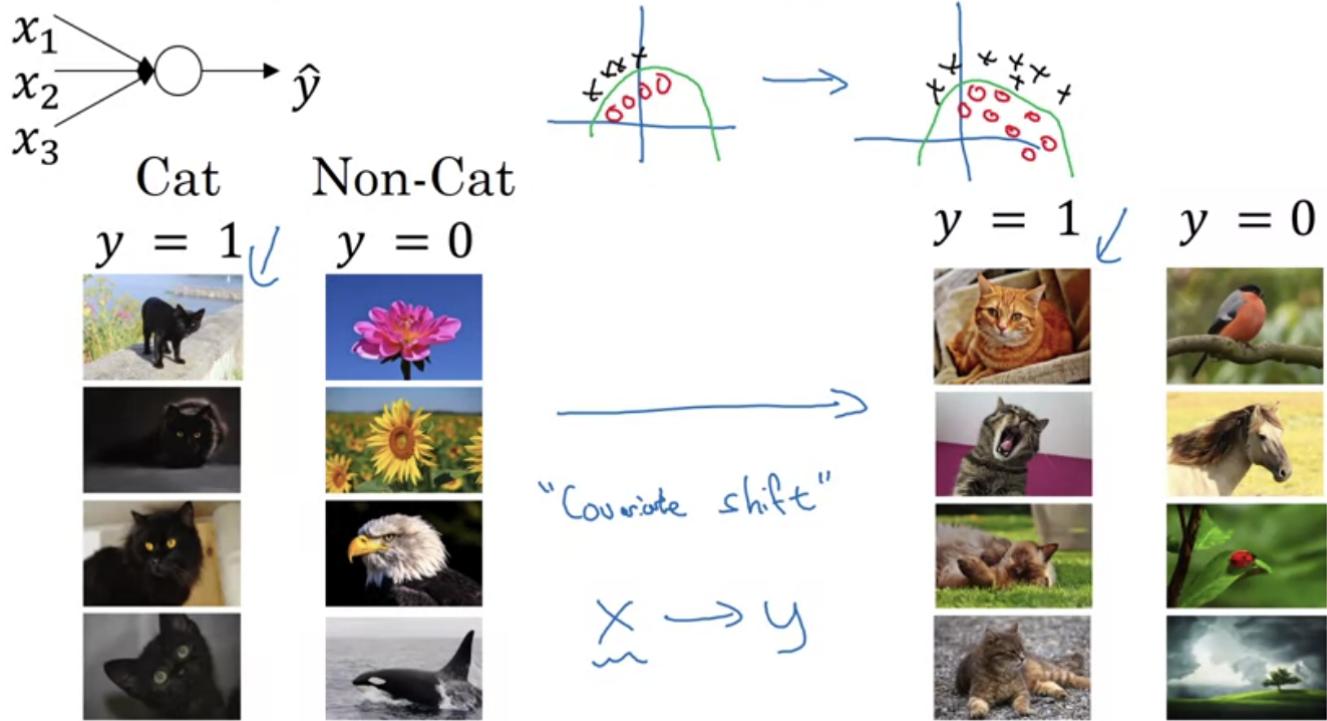
for $t = 1 \dots \text{numMiniBatches}$
Compute forward pass on $X^{t,1}, X^{t,2}, X^{t,3}$.

In each hidden layer, use BN to replace $\underline{z}^{[t]}$ with $\underline{\hat{z}}^{[t]}$.
Use backprop to compute $d\underline{w}^{[t]}, \cancel{d\underline{b}^{[t]}}, d\underline{\beta}^{[t]}, d\underline{\gamma}^{[t]}$
Update parameters $\left. \begin{array}{l} \underline{w}^{[t]} := \underline{w}^{[t]} - \alpha d\underline{w}^{[t]} \\ \underline{\beta}^{[t]} := \underline{\beta}^{[t]} - \alpha d\underline{\beta}^{[t]} \\ \underline{\gamma}^{[t]} := \dots \end{array} \right\}$

Works w/ momentum, RMSprop, Adam.

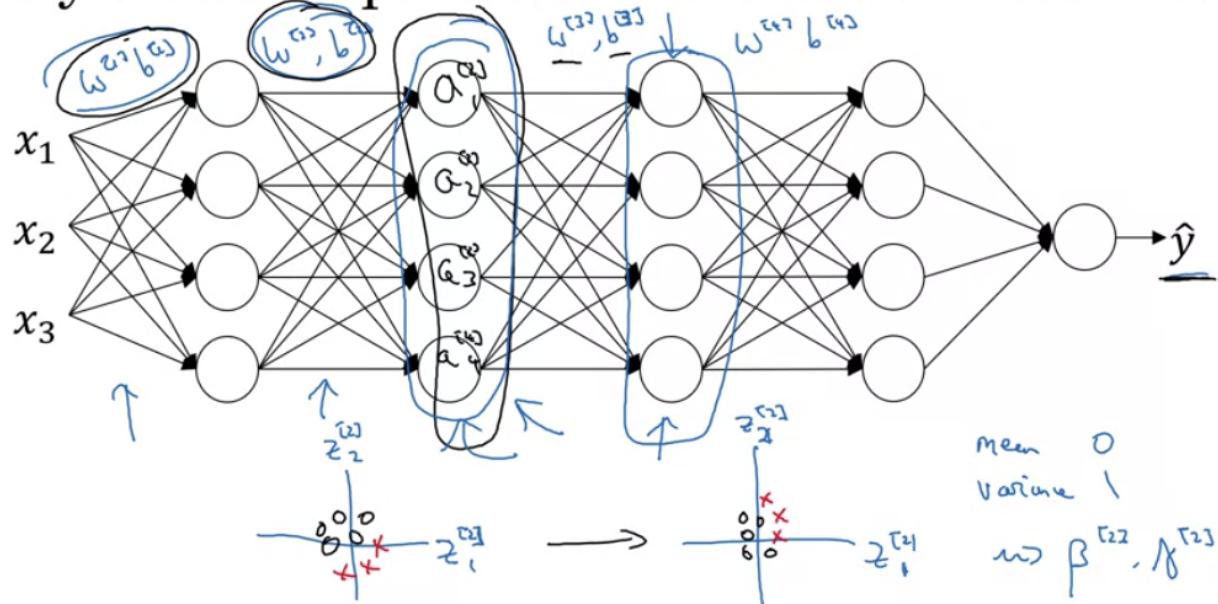
Why does Batch Norm work?

Learning on shifting input distribution



- Makes features on deeper layers more robust to changes on previous layers

Why this is a problem with neural networks?



- batch norm ensure that the mean and variance of inner layers output will stay the same even if the input distribution changed, decoupling layers dependency on previous layers shifts

Batch Norm as regularization

X

- 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.

$$\hat{z}^{[l]} = \frac{z^{[l]} - \mu}{\sigma^2 + \epsilon}$$

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

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

$$\mu, \sigma^2$$

mini-batch : 64 512

- large mini-batches reduce the regularization effect (less noisy beta, gamma between mini-batches)
- Don't try to use it for regularization effect (although it does have this unintended effect)

Batch Norm at test time

Batch Norm at test time

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

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

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

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

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

$x^{[1]}, x^{[2]}, x^{[3]}, \dots$

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

$\theta_1, \theta_2, \theta_3, \dots$

$\tilde{z}_{\text{norm}} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}}$

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

Andrew Ng

- At test time might need to process the examples one at a time
- Usually use exponential weighted average across mini-batches to estimate testtime beta and gamma parameters
- Why not preserve online average too? (question of mine)

Multi-class classification

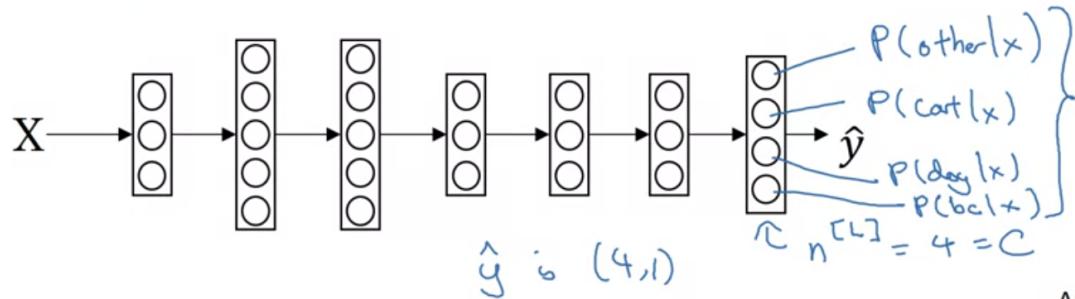
Softmax regression

Recognizing cats, dogs, and baby chicks, other



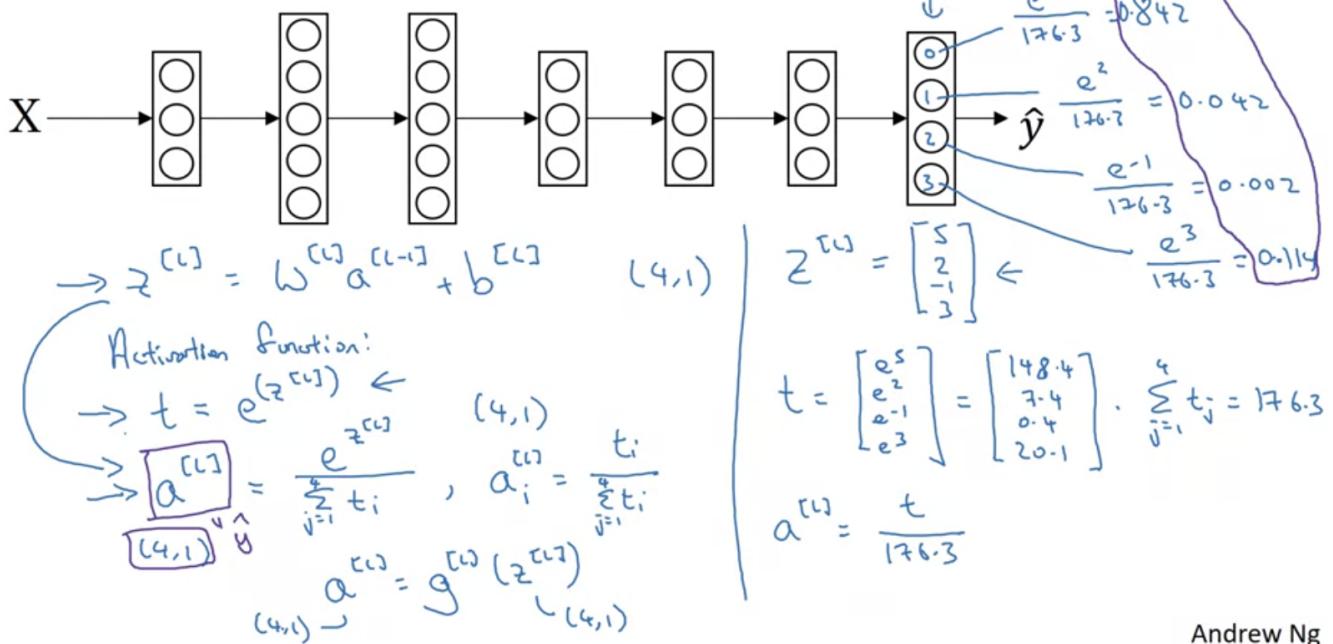
3 1 2 0 3 2 0 1

$$C = \# \text{classes} = 4 \quad (0, \dots, 3)$$

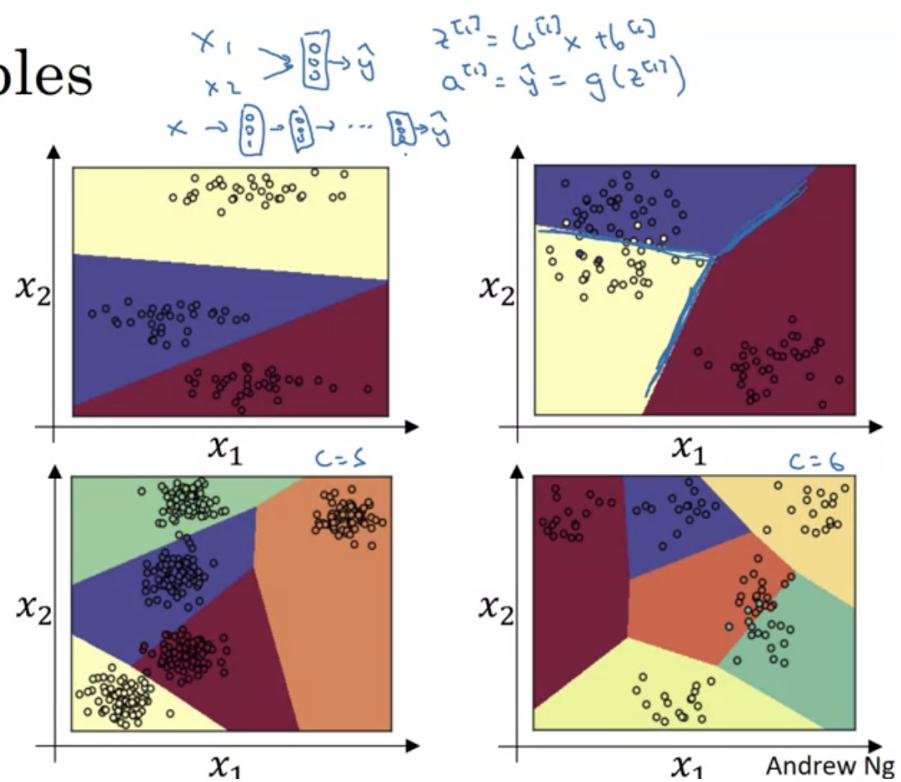
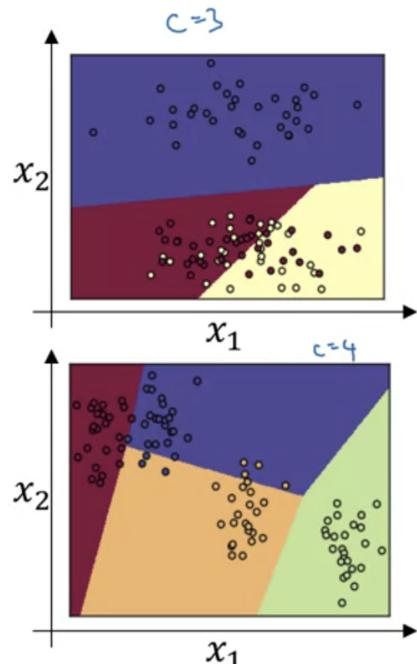


Andrew Ng

Softmax layer



Softmax examples



Training a softmax classifier

Understanding softmax

$$(4,1)$$

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

"Soft max"

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

"hard max"

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

$C = 4 \quad g^{[L]}(\cdot)$

Softmax regression generalizes logistic regression to C classes.

If $C=2$, softmax reduces to logistic regression. $\alpha^{[L]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$

Andrew Ng

Loss function

$$(4,1)$$

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{- cat } y_2 = 1$$

$y_1 = y_3 = y_4 = 0$

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

small

$$(4,1)$$

$$\alpha^{[L]} \approx \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \quad C = 4$$

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

$- y_2 \log \hat{y}_2 = -\log \hat{y}_2$. Make \hat{y}_2 big.

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

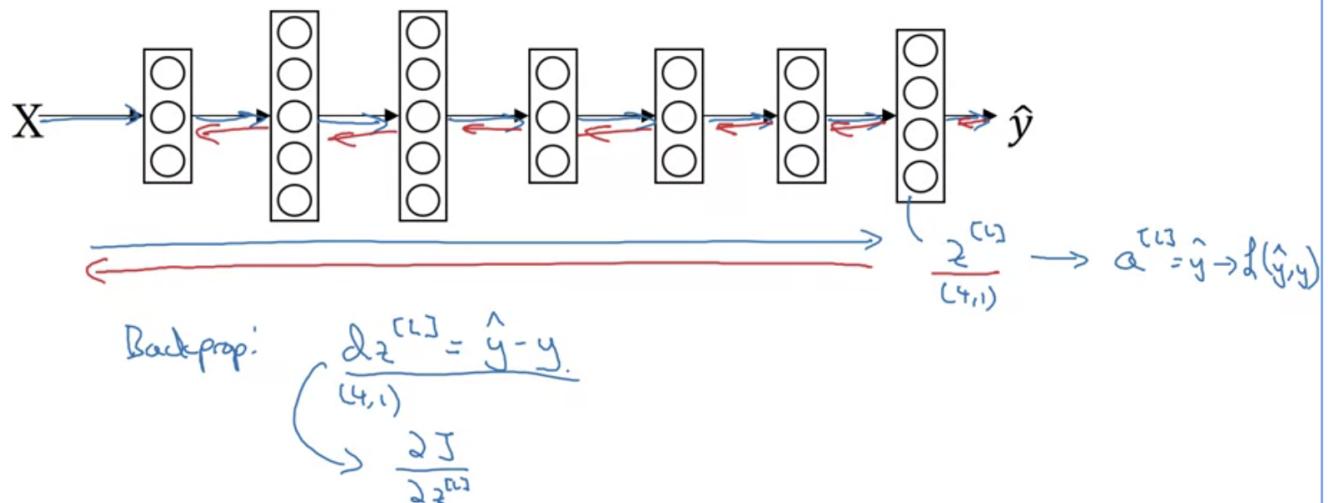
$$= \begin{bmatrix} 0 & 0 & 1 & \dots \\ 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & \dots \end{bmatrix} \quad (4, m)$$

$$\hat{Y} = [\hat{y}^{(1)} \ \dots \ \hat{y}^{(m)}]$$

$$= \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \quad \dots \quad (4, m)$$

Andrew Ng

Gradient descent with softmax



- softmax derivatives is: $y_{\text{hat}} - y$

Introduction to programming 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)

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))
```