Course 2 Improving Deep Neural Networks:

Hyperparameter tuning, Regularization and Optimization

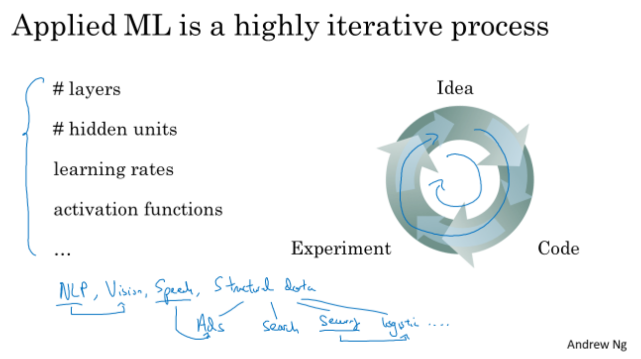
Study Notes by Jane Huang

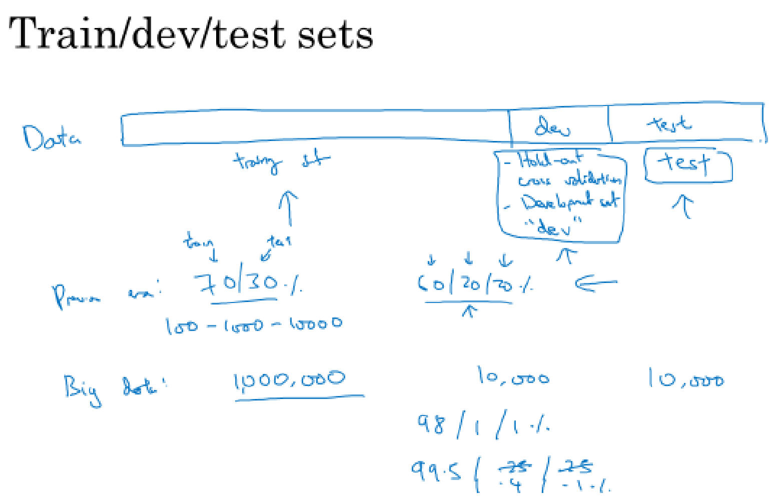
08172018

Table of Contents

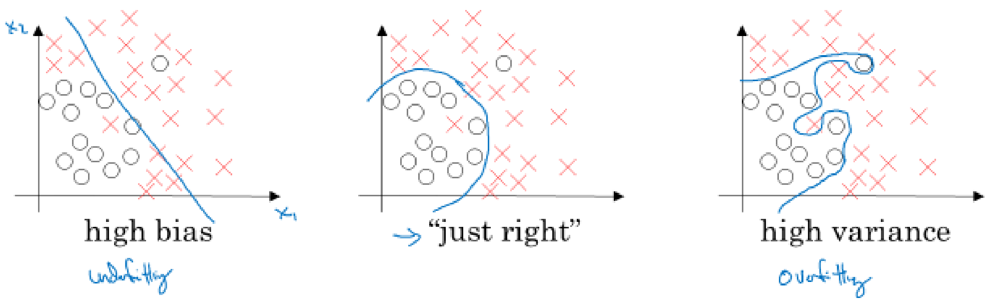
Week 1 Practical aspects of Deep Learning

1.1 Train/dev/test sets


Applied ML is a highly iterative process


Train/dev/test sets

1.2 Bias and Variance



1.3 Regularization

If you use L1 regulariztion, w will be sparse. If you use L2 regularization, "weight decay"

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$ $\lambda =$ regularization parameter

lambda lambd

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

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

L_2 regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w \leftarrow$

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

Neural network

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

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

"Frobenius norm" $\|\cdot\|_2^2$ $\|\cdot\|_F^2$

$$dW^{[l]} = \boxed{(\text{from backprop}) + \frac{\lambda}{m} W^{[l]}} \qquad \underbrace{\frac{\partial J}{\partial W^{[l]}}} = dW^{[l]}$$

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

"Weight decay"

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

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

$$= \underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{<1} \underbrace{W^{[l]} - \alpha(\text{from backprop})}$$

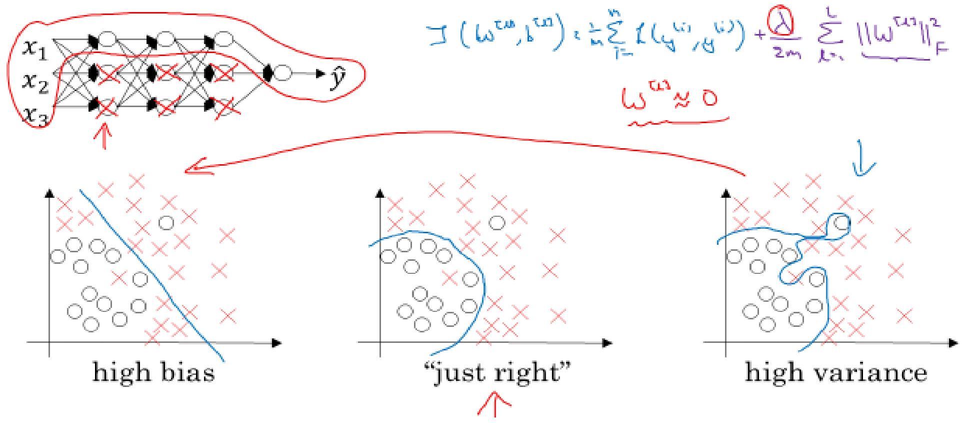
Andrew Ng

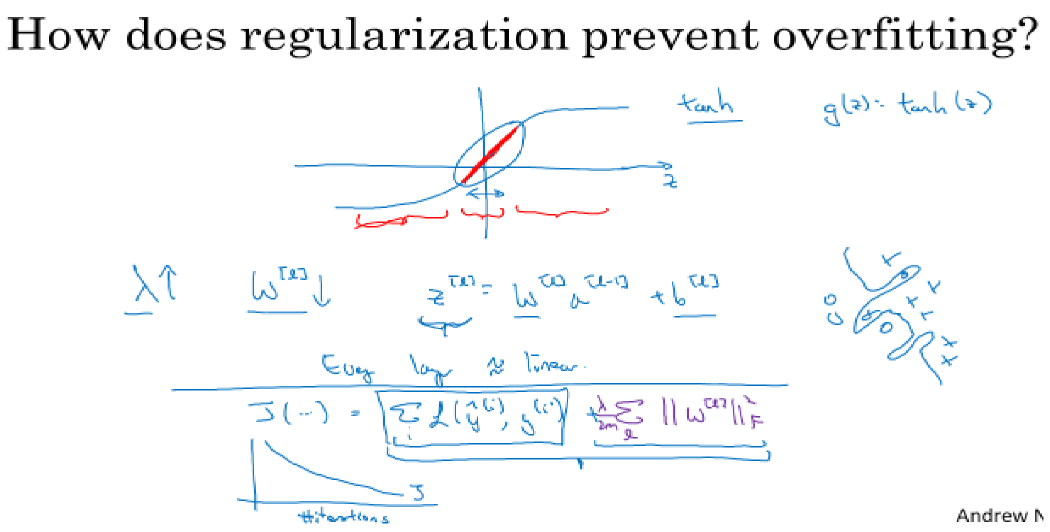
"Frobenius norm"

How does regularization prevent overfitting?

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

$W^{[l]} \approx 0$

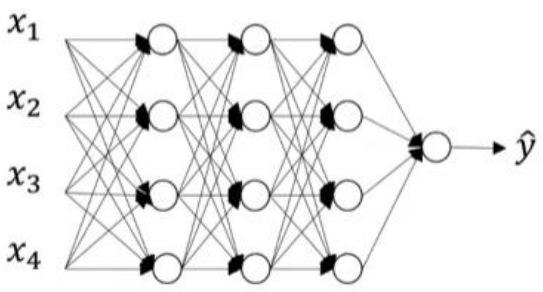


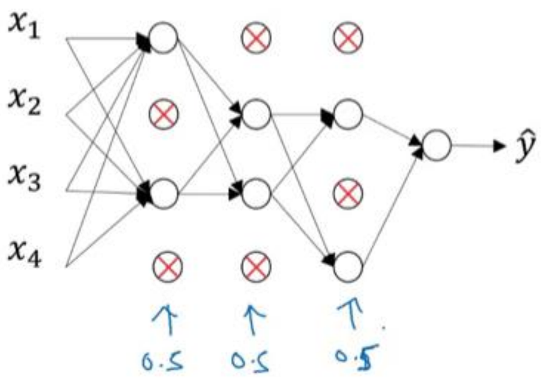

How does regularization prevent overfitting?
tanh
g(z) = tanh(z)
z
λ↑
W[l]↓
z[l] = W[l] a[l-1] + b[l]
Every layer ≈ linear.
J(...) = Σ L(ŷ(i), y(i)) + λ/2m Σ ||w[l]||²_F
J
#iterations
Andrew N

Every layer ~ Linear

If the regularization becomes very large, the parameters W very small, so Z will be relatively small, kind of ignoring the effects of b for now, so Z will be relatively small or, really, I should say it takes on a small range of values. And so the activation function if is tanh, say, will be relatively linear. And so your whole neural network will be computing something not too far from a big linear function which is therefore pretty simple function rather than a very complex highly non-linear function. And so is also much less able to overfit.

1.4 Dropout regularization

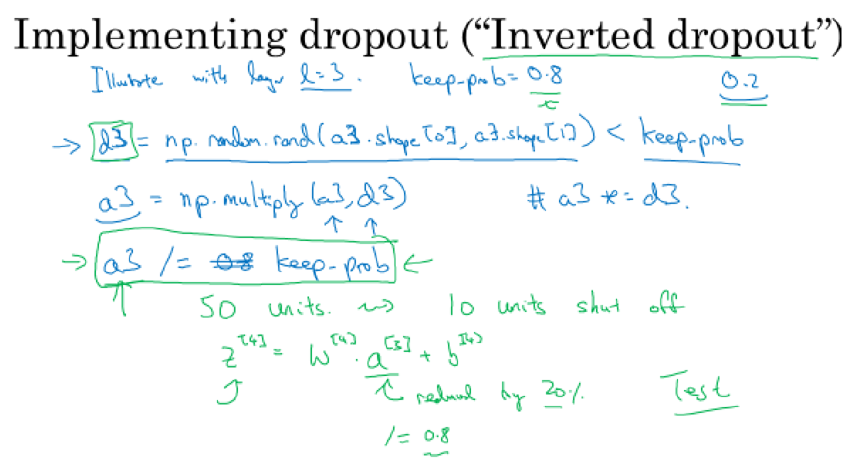




After dropout, you end up with a much smaller, much diminished network.

D3 is a Boolean array

In this example, dropout=0.2, keep-prob=1-dropout=0.8



Implementing dropout ("Inverted dropout")

Illustrate with layer $l=3$. keep-prob = 0.8 0.2

→ $\boxed{d3}$ = np.random.rand(a3.shape[0], a3.shape[1]) < keep-prob

a3 = np.multiply(a3, d3) # a3 *= d3.

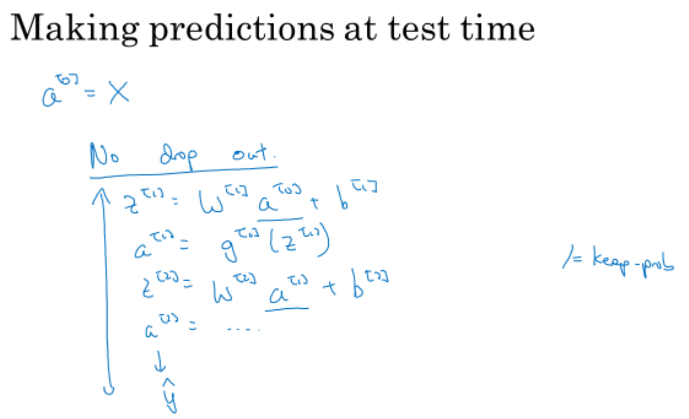
→ $\boxed{\text{a3 /= } \cancel{0.8} \text{ keep-prob}}$ ←

50 units → 10 units shut off

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

reduced by 20%. Test

/= 0.8



Making predictions at test time

$a^{[0]} = X$

No drop out.

$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$

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

$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = \dots$

$\hat{y}$

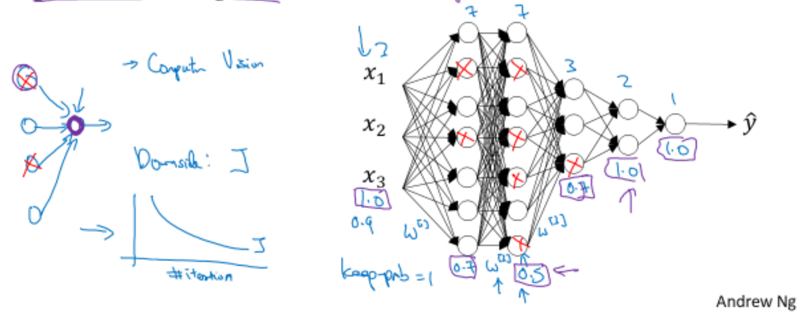
/= keep-prob

Don't implement dropout at test time, otherwise it is adding noise

No need to add extra scaling parameter

Why does drop-out work?

Intuition: Can't rely on any one feature, so have to spread out weights. → Shrink weights. L_2

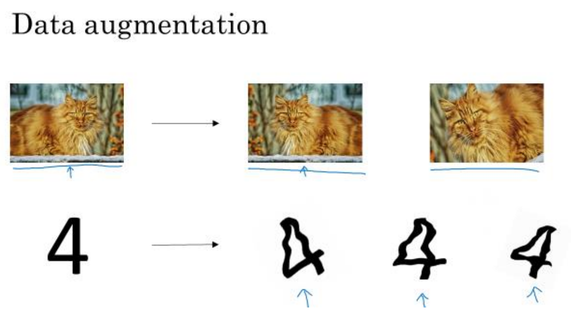


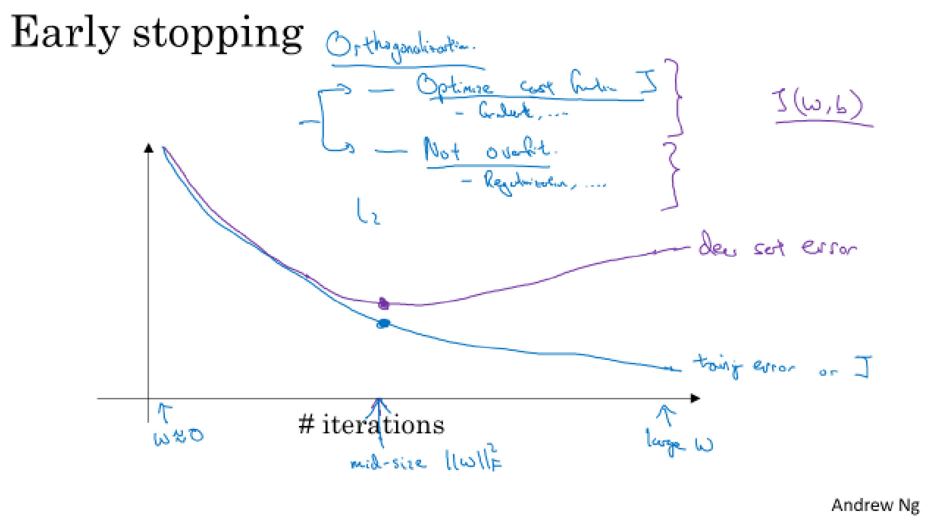
Drop out has similar effect as L2 regularization

But for layers where you're more worried about over-fitting, really the layers with a lot of parameters, you can set the key prop to be smaller to apply a more powerful form of drop out.

It's kind of like cranking up the regularization parameter lambda of L2 regularization where you try to regularize some layers more than others. Normally no dropout at input layer.

1.5 Other regularization methods



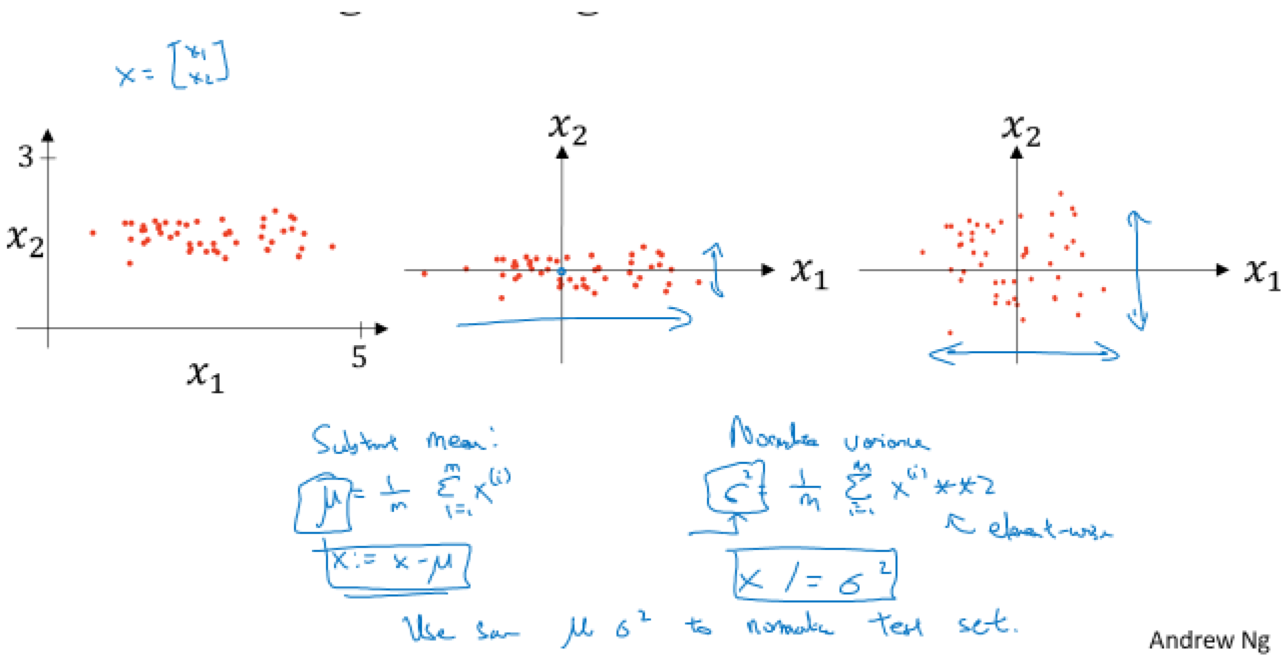


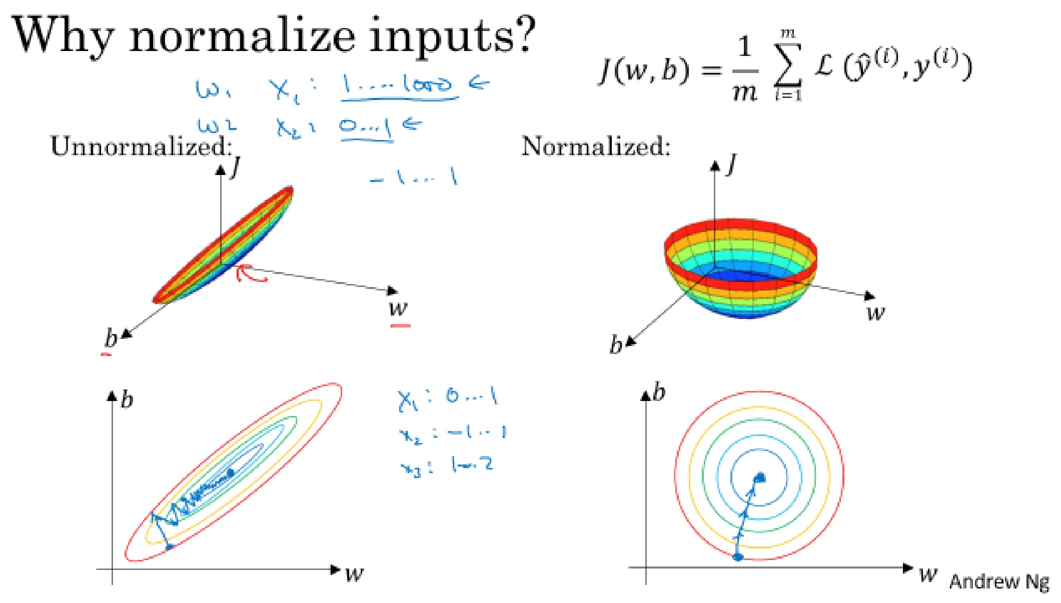
Orthogonalization

- Optimizing the cost function
  Gradient
- Not overfit
  Regularization

And the advantage of early stopping is that running the gradient descent process just once, you get to try out values of small w, mid-size w, and large w, without needing to try a lot of values of the L2 regularization hyperparameter lambda. But I (Andrew) personally prefer to just use L2 regularization and try different values of lambda. That's assuming you can afford the computation to do so. But early stopping does let you get a similar effect without needing to explicitly try lots of different values of lambda. So you've now seen how to use data augmentation as well as if you wish early stopping in order to reduce variance or prevent over fitting your neural network.

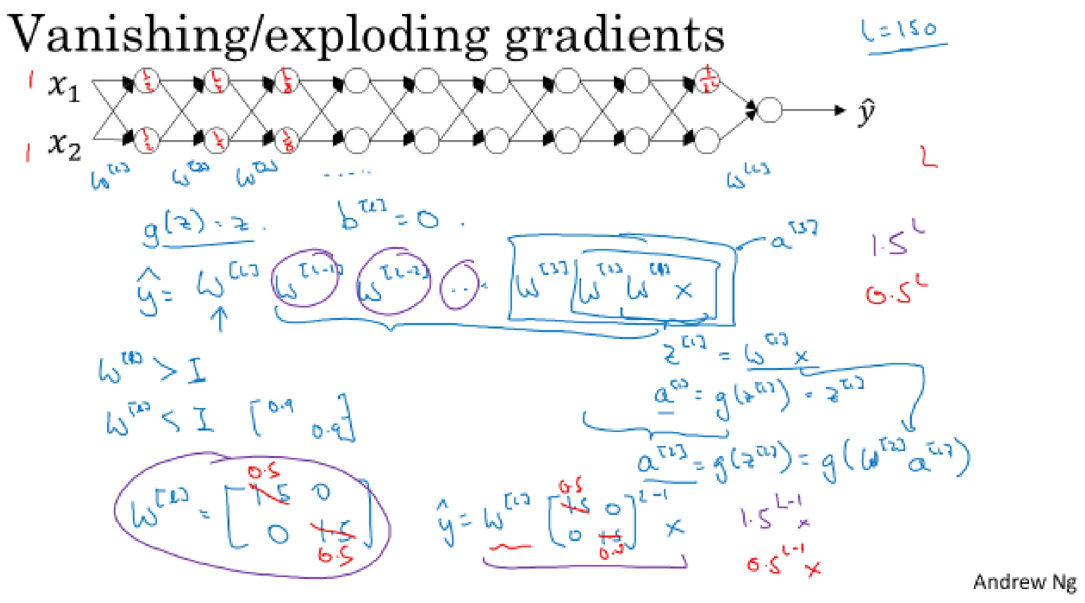
1.6 Normalizing inputs


$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
3
x_2
x_1
5
x_2
x_1
x_2
x_1
Subtract mean:
$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
$x := x - \mu$
Normalize variance
$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} ** 2$
element-wise
$x /= \sigma^2$
Use same μ, σ^2 to normalize test set.
Andrew Ng



If you use unnormalized input features, it's more likely that your cost function will look like this, it's a very squished out bowl, very elongated cost function, where the minimum you're trying to find is maybe over there. But if your features are on very different scales, say the feature X1 ranges from 1 to 1,000, and the feature X2 ranges from 0 to 1, then it turns out that the ratio or the range of values for the parameters w1 and w2 will end up taking on very different values. And so maybe these axes should be w1 and w2, but I'll plot w and b, then your cost function can be a very elongated bowl like that. So if you part the contours of this function, you can have a very elongated function like that. Whereas if you normalize the features, then your cost function will on average look more symmetric. And if you're running gradient descent on the cost function like the one on the left, then you might have to use a very small learning rate because if you're here that gradient descent might need a lot of steps to oscillate back and forth before it finally finds its way to the minimum. Whereas if you have a more spherical contours, then wherever you start gradient descent can pretty much go straight to the minimum. You can take much larger steps with gradient descent rather than needing to oscillate around like like the picture on the left. Of course in practice w is a high-dimensional vector, and so trying to plot this in 2D doesn't convey all the intuitions correctly. But the rough intuition that your cost function will be more round and easier to optimize when your features are all on similar scales.

1.7 Vanishing / Exploding gradient



If linear activation function, for example, g(z)=z

If w_l>1 exploding gradients

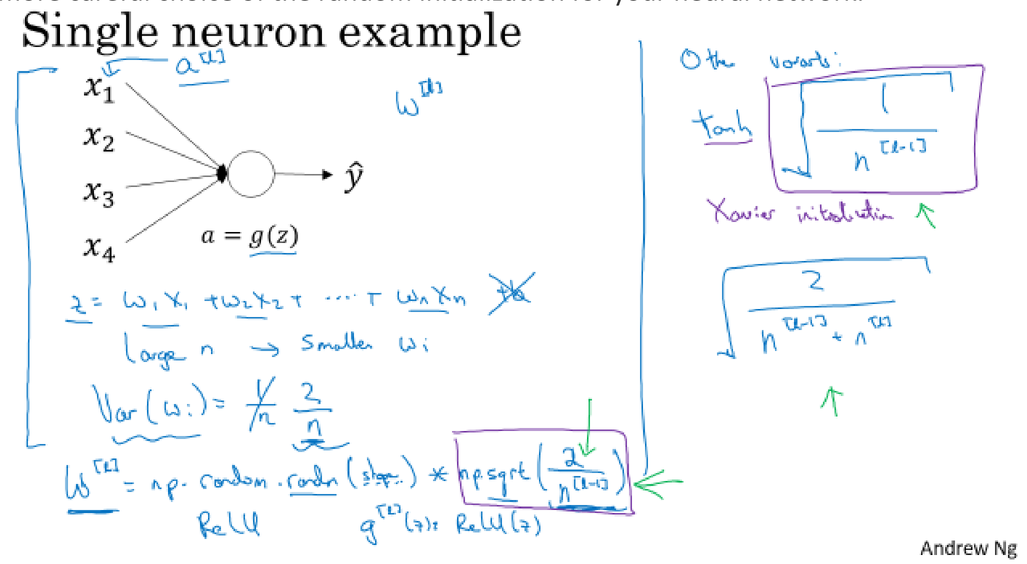
If w_l<1 vanishing gradients

So the intuition I hope you can take away from this is that at the weights W, if they're all just a little bit bigger than one or just a little bit bigger than the identity matrix, then with a very deep network the activations can explode. And if W is just a little bit less than identity. So this maybe here's 0.9, 0.9, then you have a very deep network, the activations will decrease exponentially. And even though I went through this argument in terms of activations increasing or decreasing exponentially as a function of L, a similar argument can be used to show that the derivatives or the gradients the computer is going to send will also increase exponentially or decrease exponentially as a function of the number of layers. 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.

To summarize, you've seen how deep networks suffer from the problems of vanishing or exploding gradients. In fact, for a long time this problem was a huge barrier to training deep neural networks. It turns out there's a partial solution that doesn't completely solve this problem but it helps a lot which is careful choice of how you initialize the weights.

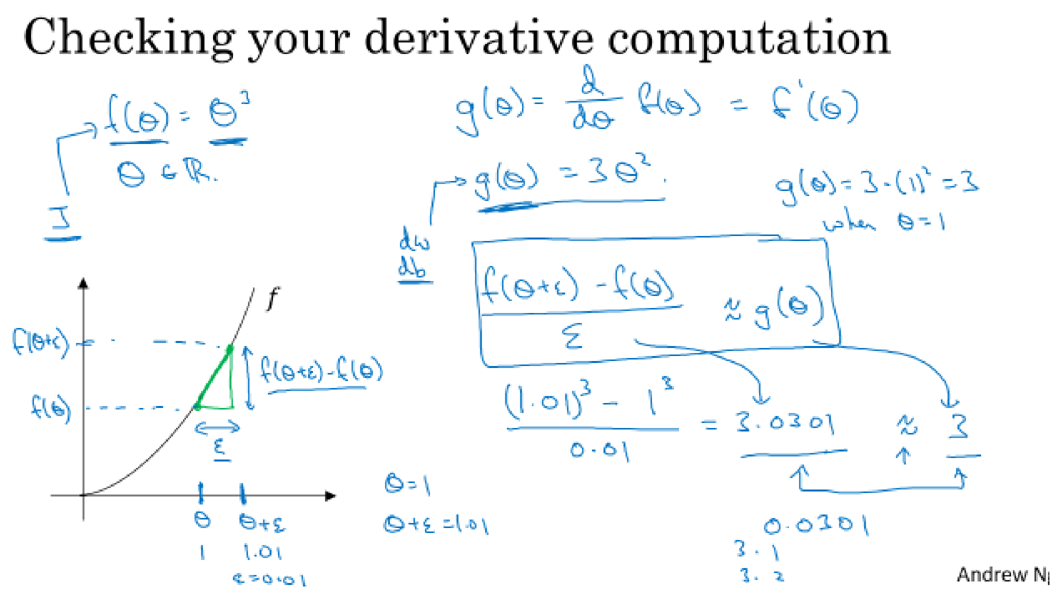
1.8 Weight Initialization for Deep Networks

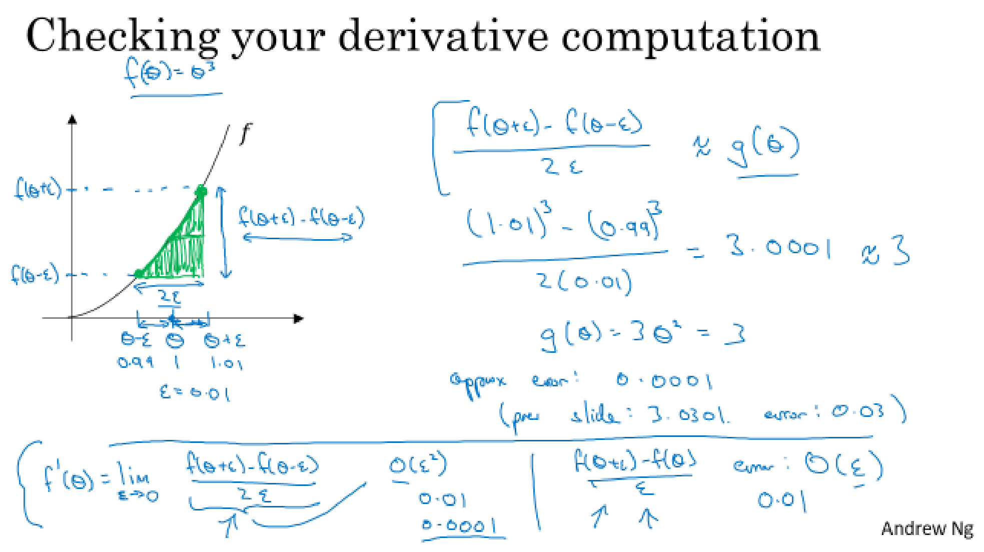
A partial solution to vanishing/exploding gradient, doesn't solve it entirely but helps a lot, is better or more careful choice of the random initialization for your neural network.


Single neuron example
x_1
x_2
x_3
x_4
$a^{[l]}$
$W^{[l]}$
$\hat{y}$
$a = g(z)$
$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$
Large n → Smaller w_i
$Var(w_i) = \frac{1}{n}$ $\frac{2}{n}$
$W^{[l]}$ = np.random.randn(shape) * np.sqrt($\frac{2}{n^{[l-1]}}$)
ReLU
$g^{[l]}(z) = ReLU(z)$
Other variants:
tanh
$\sqrt{\frac{1}{n^{[l-1]}}}$
Xavier initialization
$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
Andrew Ng

Xavier Initialization

1.9 Numerical approximation of gradients


Checking your derivative computation
$f(\theta) = \theta^3$
$\theta \in \mathbb{R}$
J
$g(\theta) = \frac{d}{d\theta} f(\theta) = f'(\theta)$
$g(\theta) = 3\theta^2$
$g(\theta) = 3 \cdot (1)^2 = 3$ when $\theta = 1$
dw
db
$\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} \approx g(\theta)$
$\frac{(1.01)^3 - 1^3}{0.01} = 3.0301 \approx 3$
f
$f(\theta+\varepsilon)$
$f(\theta)$
$f(\theta+\varepsilon) - f(\theta)$
ε
θ
$\theta+\varepsilon$
1
1.01
$\varepsilon = 0.01$
$\theta = 1$
$\theta + \varepsilon = 1.01$
0.0301
3.1
3.2
Andrew Ng



Checking your derivative computation

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

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

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

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

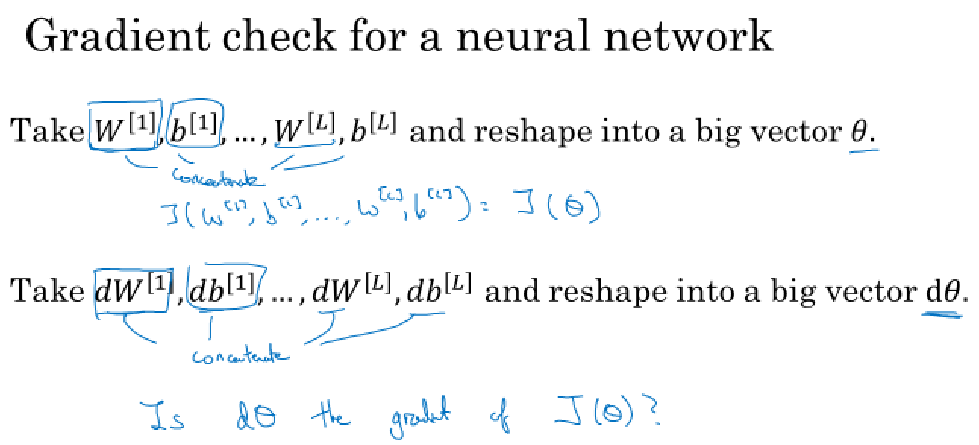
approx error: 0.0001

(prev slide: 3.0301. error: 0.03)

$$f'(\theta)=\lim_{\varepsilon\to 0}\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} \quad O(\varepsilon^2) \quad 0.01 \quad 0.0001 \qquad \Big| \qquad \frac{f(\theta+\varepsilon)-f(\theta)}{\varepsilon} \quad \text{error: } O(\varepsilon) \quad 0.01$$

Andrew Ng

1.10 Gradient Checking



Gradient check for a neural network

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

concatenate

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

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

concatenate

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

- You gotta take all of these Ws and reshape them into vectors, and then concatenate all of these things, so that you have a giant vector theta. The cost function is a function of theta, therefore a function of the Ws and Bs.
- You gotta take all of these dWs and reshape them into vectors, and then concatenate all of these things, so that you have a giant vector dtheta.

Gradient checking (Grad check)

$J(\theta) = J(\theta_1, \theta_2, \theta_3, \dots)$

for each i:

$\rightarrow \underline{d\theta_{approx}[i]} = \dfrac{J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i} + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i} - \varepsilon, \dots)}{2\varepsilon}$

$\approx \underline{d\theta[i]} = \dfrac{\partial J}{\partial \theta_i}$ | $d\theta_{approx} \overset{?}{\approx} d\theta$

Check $\dfrac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx \boxed{10^{-7} - \text{great!}} \leftarrow$

10^{-5}

$\underline{\varepsilon = 10^{-7}}$ $\rightarrow 10^{-3}$ – worry. $\leftarrow$

Gradient checking implementation notes

- Don't use in training – only to debug $d\theta_{approx}[i] \longleftrightarrow d\theta[i]$

- If algorithm fails grad check, look at components to try to identify bug $db^{[l]}$ $dw^{[l]}$

- Remember regularization. $J(\theta) = \frac{1}{m} \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \underline{\frac{\lambda}{2m} \sum_l \|w^{[l]}\|_F^2}$
  $d\theta$ = grad of J wrt. θ

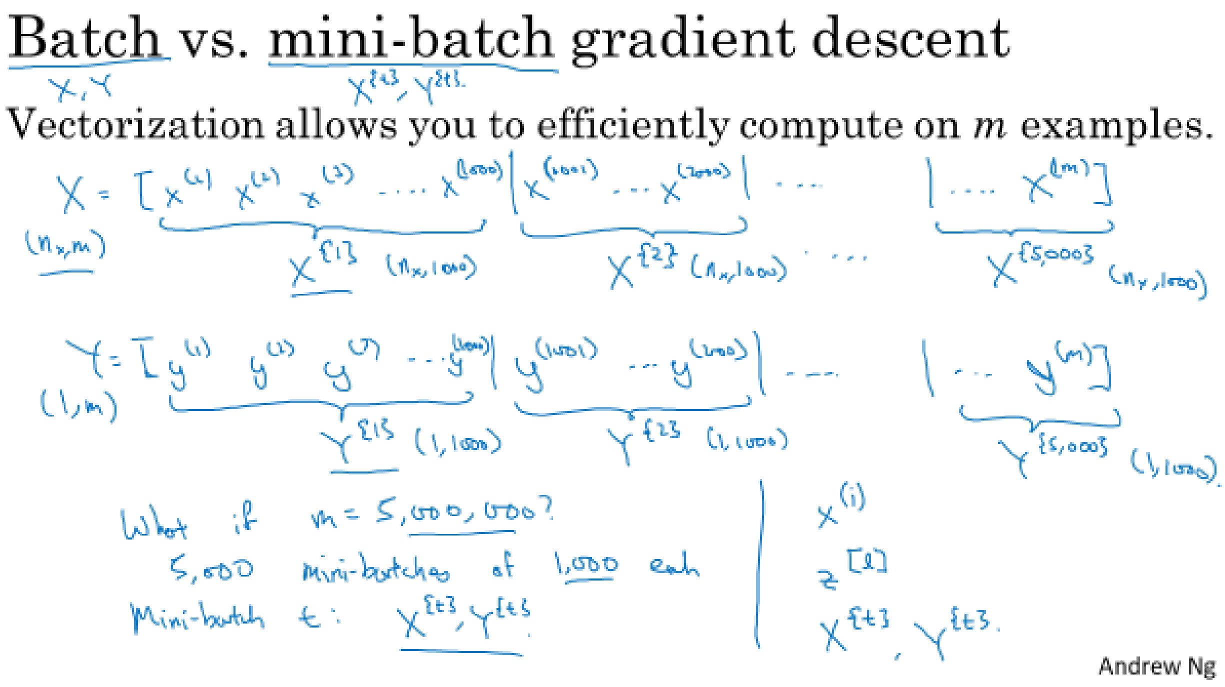
- Doesn't work with dropout. J $\underline{\text{keep-prob} = 1.0}$

- Run at random initialization; perhaps again after some training.
  $w, b \approx 0$

You can do the grad check by setting keep_prob=1.0. If it is correct, then turn on drop out.

Week 2 **Optimization algorithms**

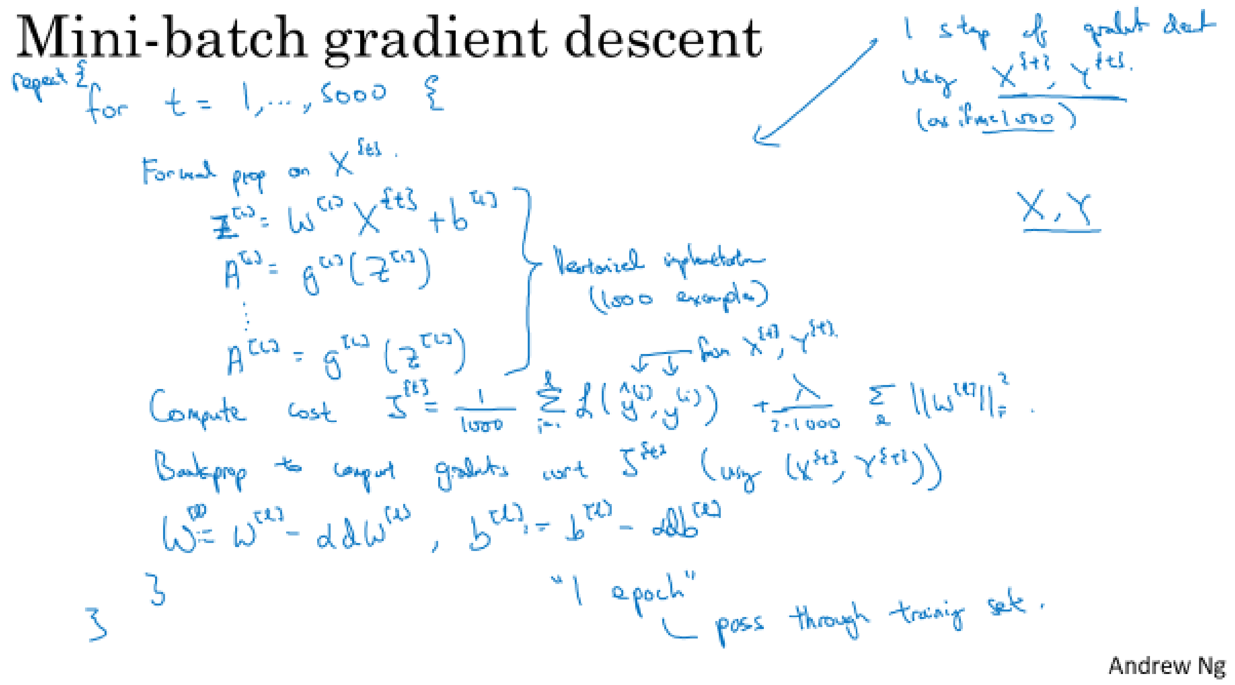
2.1 Mini-batch gradient descent


Batch vs. mini-batch gradient descent
X, Y
X{t}, Y{t}
Vectorization allows you to efficiently compute on m examples.
X = [x(1) x(2) x(3) .... x(1000) | x(1001) ... x(2000) | ... | ... x(m)]
(nx, m)
X{1} (nx, 1000)
X{2} (nx, 1000)
X{5000} (nx, 1000)
Y = [y(1) y(2) y(3) ... y(1000) | y(1001) ... y(2000) | ... | ... y(m)]
(1, m)
Y{1} (1, 1000)
Y{2} (1, 1000)
Y{5,000} (1, 1000)
What if m = 5,000,000?
5,000 mini-batches of 1,000 each
Mini-batch t: X{t}, Y{t}
x(i)
z[l]
X{t}, Y{t}
Andrew Ng

With the implementation of gradient descent on your whole training set, what you have to do is, you have to process your entire training set before you take one little step of gradient descent. And then you have to process your entire training sets of five million training samples again before you take another little step of gradient descent. So, it turns out that you can get a faster algorithm if you let gradient descent start to make some progress even before you finish processing your entire, your giant training sets of 5 million examples.

Let's say that you split up your training set into smaller, little baby training sets and these baby training sets are called mini-batches.

Use curly bracket to index mini-batch



Mini-batch gradient descent

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

X, Y

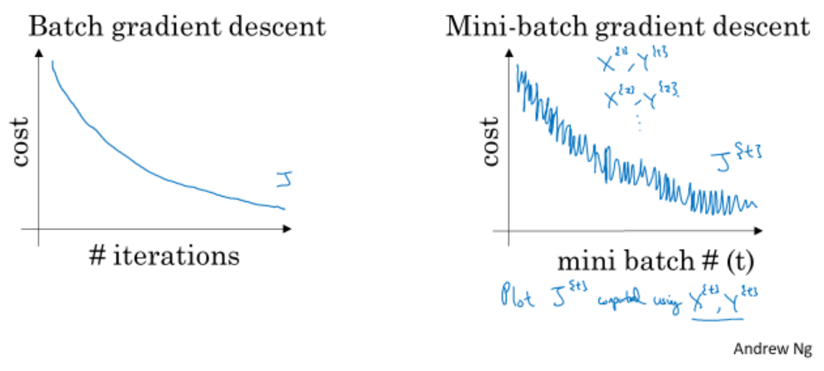
```
repeat {
    for $t = 1, \dots, 5000$ {
        Forward prop on $X^{\{t\}}$.
            $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
            $A^{[1]} = g^{[1]}(Z^{[1]})$
            ⋮
            $A^{[L]} = g^{[L]}(Z^{[L]})$
        } Vectorized implementation (1000 examples)
        Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$.   (from $X^{\{t\}}, Y^{\{t\}}$)
        Backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)
        $W^{[l]} := W^{[l]} - \alpha dW^{[l]}$, $b^{[l]} := b^{[l]} - \alpha db^{[l]}$
    }
}
```

"1 epoch"
└ pass through training set.

Andrew Ng

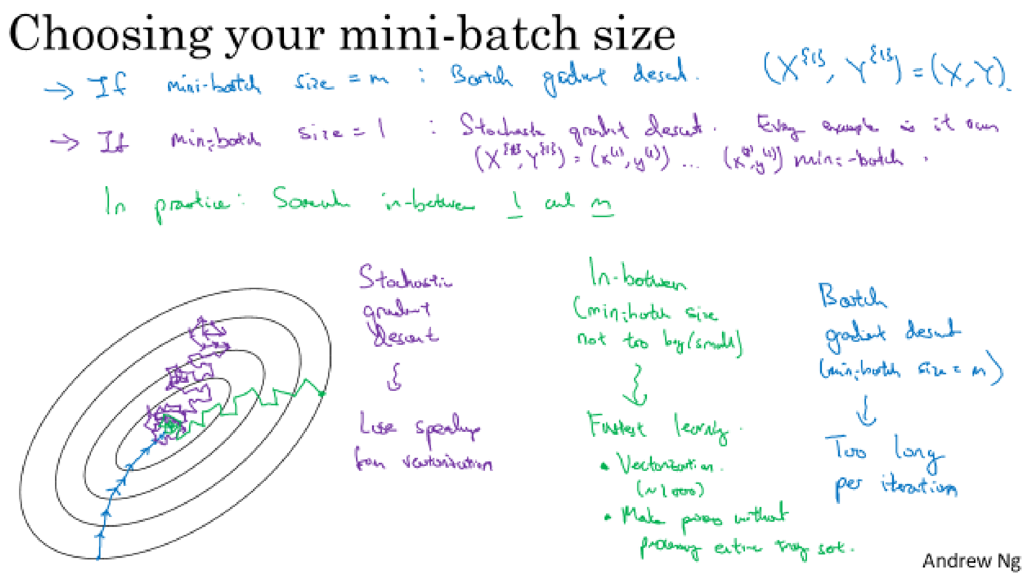
The code I have written down here is also called doing one epoch of training and epoch is a word that means a single pass through the training set. Whereas with batch gradient descent, a single pass through the training allows you to take only one gradient descent step. With mini-batch gradient descent, a single pass through the training set, that is one epoch, allows you to take 5,000 gradient descent steps. Now of course you want to take multiple passes through the training set which you usually want to, you might want another for loop for another while loop out there. So you keep taking passes through the training set until hopefully you converge with approximately converge. When you have a lot training set, mini-batch gradient descent runs much faster than batch gradient descent and that's pretty much what everyone in Deep Learning will use when you're training on a large data set.

Training with mini batch gradient descent



- With batch gradient descent on every iteration you go through the entire training set and you'd expect the cost to go down on every single iteration. So if we've had the cost function j as a function of different iterations it should decrease on every single iteration. And if it ever goes up even on iteration then something is wrong. Maybe you're running ways to big.

- On mini batch gradient descent though, if you plot progress on your cost function, then it may not decrease on every iteration. In particular, on every iteration you're processing some X{t}, Y{t} and so if you plot the cost function J{t}, which is computer using just X{t}, Y{t}. Then it's as if on every iteration you're training on a different training set or really training on a different mini batch. So you plot the cross function J, you're more likely to see something that looks like this. It should trend downwards, but it's also going to be a little bit noisier.



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 it own $(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)})$ … $(x^{(2)}, y^{(2)})$ mini-batch.

In practice: Somewhere in-between $\underline{1}$ and $\underline{m}$

Stochastic gradient descent → Lose speedup from vectorization

In-between (mini-batch size not too big/small) → Fastest learning.
- Vectorization. (~1000)
- Make progress without processing entire training set.

Batch gradient descent (mini-batch size = m) → Too long per iteration

Andrew Ng

In practice, somewhere between 1 and m (sample size)

If these are the contours of the cost function you're trying to minimize so your minimum is there. Then batch gradient descent might start somewhere and be able to take relatively low noise, relatively large steps. And you could just keep matching to the minimum. In contrast with stochastic gradient descent If you start somewhere let's pick a different starting point. Then on every iteration you're taking gradient descent with just a single strain example so most of the time you hit two at the global minimum. But sometimes you hit in the wrong direction if that one example happens to point you in a bad direction. So stochastic gradient descent can be extremely noisy. And on average, it'll take you in a good direction, but sometimes it'll head in the wrong direction as well. As stochastic gradient descent won't ever converge, it'll always just kind of oscillate and wander around the region of the minimum. But it won't ever just head to the minimum and stay there. In practice, the mini-batch size you use will be somewhere in between.

Choosing your mini-batch size

- If small training set (m<2000): Use batch gradient descent
- If bigger training set

  Typical mini-batch size

  -> 64, 128, 256, 512

Make sure mini-batch fit in CPU/GPU memory.

In practice of course the mini batch size is another hyper parameter that you might do a quick search over to try to figure out which one is most sufficient of reducing the cost function j. So what i would do is just try several different values. Try a few different powers of two and then see if you can pick one that makes your gradient descent optimization algorithm as efficient as possible. But it turns out there're even more efficient algorithms than gradient descent or mini-batch gradient descent. Let's start talking about them in the next few videos.

2.2Exponentially weighted averages

Temperature in London

$\theta_1 = 40°F$ 4°C ←

$\theta_2 = 49°F$ 9°C

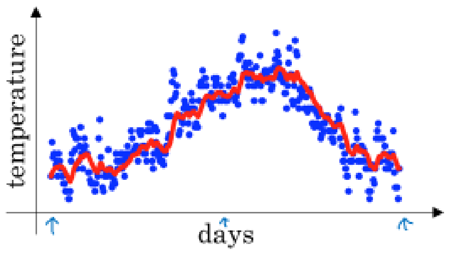
$\theta_3 = 45°F$ ⋮

⋮

$\theta_{180} = 60°F$ 15°C

$\theta_{181} = 56°F$ ⋮

⋮



$V_0 = 0$

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

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

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

⋮

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

Andrew Ng

Exponentially weighted averages

moving

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

$\beta = 0.9$: ≈ 10 days' temperature

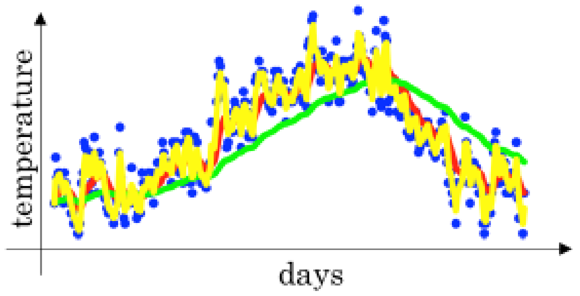
$\beta = 0.98$: ≈ 50 days

$\beta = 0.5$: ≈ 2 days

V_t as approximately average over

→ $\approx \frac{1}{1-\beta}$ days' temperature.

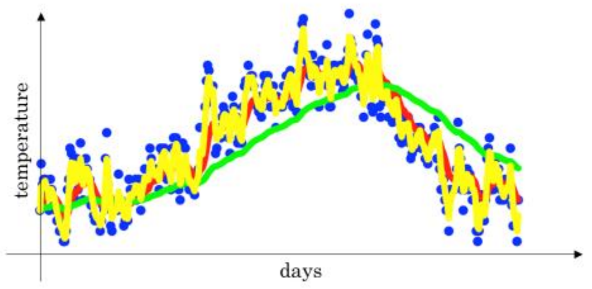
$\frac{1}{1-0.98} = 50$



Andrew Ng

Exponentially weighted averages

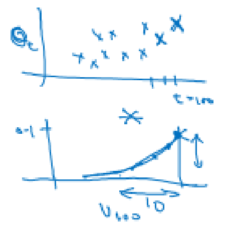
$\boxed{v_t = \beta v_{t-1} + (1-\beta)\theta_t}$ $\beta = 0.9$ 0.98 0.5



Andrew Ng

Exponentially weighted averages

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



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

$\varepsilon = 1-\beta$

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

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

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

…

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

$\underline{v_{100}} = 0.1\,\theta_{100} + 0.9\,\cancel{v_{99}}\,(0.1\,\underline{\theta_{99}} + 0.9\,\cancel{v_{98}})$

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

$\underline{0.9^{10}} \approx \underline{0.35} \approx \frac{1}{e}$ $\quad \underbrace{(1-\varepsilon)}_{0.9}{}^{1/\varepsilon} = \frac{1}{e}$ $\quad 0.98\,?$

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

Andrew Ng

Implementing exponentially weighted averages

$v_0 = 0$
$v_1 = \beta v_0 + (1-\beta)\,\theta_1$
$v_2 = \beta v_1 + (1-\beta)\,\theta_2$
$v_3 = \beta v_2 + (1-\beta)\,\theta_3$
…

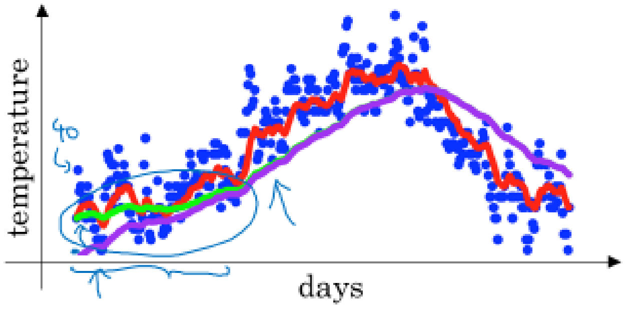
$v_\theta := 0$
$v_\theta := \beta v + (1-\beta)\,\theta_1$
$v_\theta := \beta v + (1-\beta)\,\theta_2$
⋮

```
→ $v_\theta = 0$
Repeat {
    Get next $\theta_t$
    $v_\theta := \beta v_\theta + (1-\beta)\theta_t$ ←
}
```

Andrew Ng

2.3 Bias correction in exponentially weighted averages

Bias correction



$\beta = 0.98$

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

$V_0 = 0$

$V_1 = \cancel{0.98 V_0 +} 0.02\,\theta_1$

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

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

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

$\frac{V_t}{1-\beta^t}$

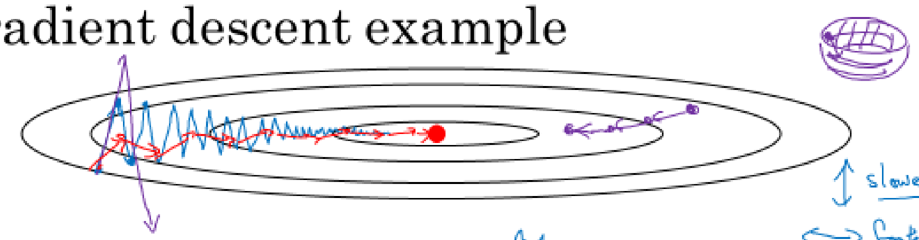
$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

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

Andrew Ng

2.4 Gradient descent with momentum

Gradient descent example



$\updownarrow$ slower learning

$\leftrightarrow$ faster learning

Momentum:

On iteration t:

Compute dW, db on current mini-batch.

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

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

friction ⟶ $\uparrow$ velocity $\quad$ $\uparrow$ acceleration

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

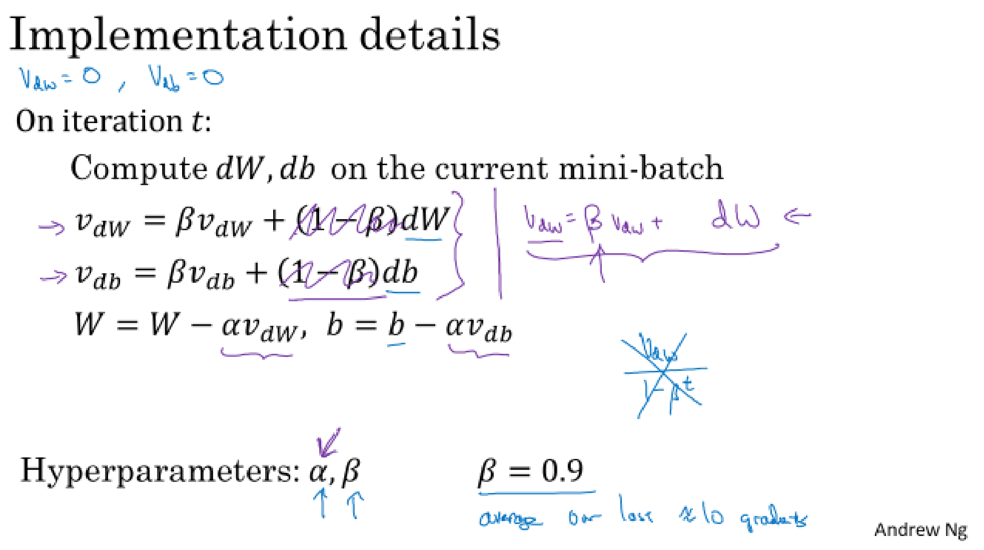
$W := W - \alpha V_{dW}, \quad b := b - \alpha V_{db}$

deeplearning.ai

Andrew Ng

There's an algorithm called momentum, or gradient descent with momentum that almost always works faster than the standard gradient descent algorithm. In one sentence, the basic idea is to compute an exponentially weighted average of your gradients, and then use that gradient to update your weights instead. As an example let's say that you're trying to optimize a cost function which has contours like this. So the red dot denotes the position of the minimum. Maybe you start gradient descent here and if you take one iteration of gradient descent either or descent maybe end up -heading there. But now you're on the other side of this ellipse, and if you take another step of gradient descent maybe you end up doing that. And then another step, another step, and so on. And you see that gradient descents will sort of take a lot of steps, right? Just slowly oscillate toward the minimum. And this up and down oscillations slows down gradient descent and prevents you from using a much larger learning rate. In particular, if you were to use a much larger learning rate you might end up over shooting and end up diverging like so. And so the need to prevent the oscillations from getting too big forces you to use a learning rate that's not itself too large. Another way of viewing this problem is that on the vertical axis you want your learning to be a bit slower, because you don't want those oscillations. But on the horizontal axis, you want faster learning.

If you average out these gradients, you find that the oscillations in the vertical direction will tend to average out to something closer to zero. So, in the vertical direction, where you want to slow things down, this will average out positive and negative numbers, so the average will be close to zero. Whereas, on the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the average in the horizontal direction will still be pretty big. So that's why with this algorithm, with a few iterations you find that the gradient descent with momentum ends up eventually just taking steps that are much smaller oscillations in the vertical direction, but are more directed to just moving quickly in the horizontal direction. And so this allows your algorithm to take a more straightforward path, or to damp out the oscillations in this path to the minimum.



Implementation details

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

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

$V_{dw} = \beta V_{dw} + dW$

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

Hyperparameters: α, β

$\beta = 0.9$

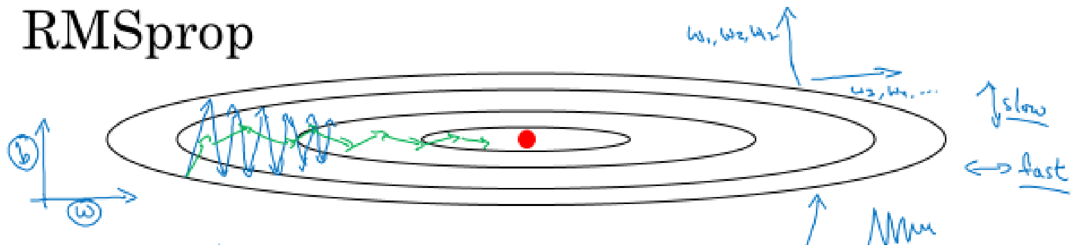
average over last ≈ 10 gradients

Andrew Ng

2.5 RMSprop

RMSprop: root mean squre prop, which also can speed up gradient descent (like momentum)

RMSprop



On iteration t:

Compute dW, db on current mini-batch

$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\, dW^2 \leftarrow \text{small}$$

element-wise

$$\rightarrow S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2 \leftarrow \text{large}$$

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

$$\varepsilon = 10^{-8}$$

That your updates in the, Vertical direction and then horizontal direction you can keep going. And one effect of this is also that you can therefore use a larger learning rate alpha, and get faster learning without diverging in the vertical direction.

So that's RMSprop, and similar to momentum, has the effects of damping out the oscillations in gradient descent, in mini-batch gradient descent. And allowing you to maybe use a larger learning rate alpha. And certainly speeding up the learning speed of your algorithm. One fun fact about RMSprop, it was actually first proposed not in an academic research paper, but in a Coursera course that Jeff Hinton had taught on Coursera many years ago. I guess Coursera wasn't intended to be a platform for dissemination of novel academic research, but it worked out pretty well in that case. And was really from the Coursera course that RMSprop started to become widely known and it really took off.

2.6 Adam optimization algorithm

Adam optimization algorithm

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

On iterate t:

Compute dW, db using current mini-batch

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

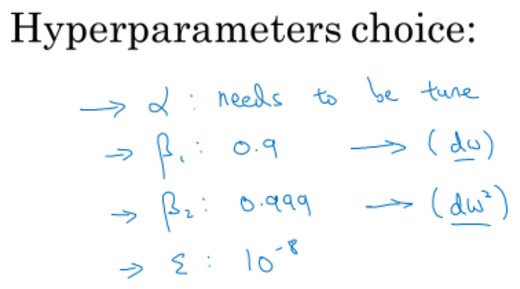
$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dW^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db \leftarrow \text{"RMSprop"}\ \beta_2$$

yhat = np.array([.9, 0.2, 0.1, .4, .9])

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

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

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



Hyperparameters choice:

$\rightarrow \alpha$: needs to be tune

$\rightarrow \beta_1 : 0.9 \longrightarrow (\underline{dw})$

$\rightarrow \beta_2 : 0.999 \longrightarrow (\underline{dw^2})$

$\rightarrow \varepsilon : 10^{-8}$

Adam: Adaptive Moment Estimation

When implementing Adam, what people usually do is just use the default value. So, ß1 and ß2 as well as epsilon. I don't think anyone ever really tunes Epsilon. And then, try a range of values of Alpha to see what works best. You could also tune ß1 and ß2 but it's not done that often among the practitioners I know. ß1 is computing the mean of the derivatives. This is called the first moment. And ß2 is used to compute exponentially weighted average of the ^{2}s and that's called the second moment. So that gives rise to the name adaptive moment estimation. But everyone just calls it the Adam authorization algorithm.

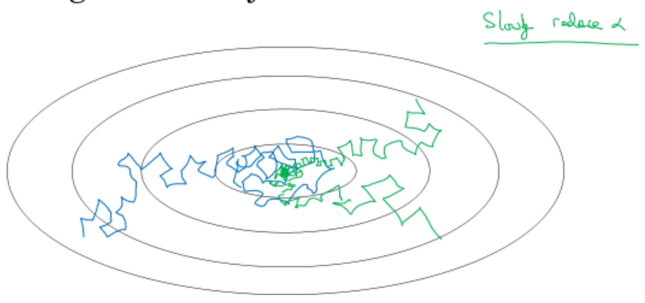
2.7 Learning rate decay

One of the things that might help speed up your learning algorithm, is to slowly reduce your learning rate over time. We call this learning rate decay.

. Let's start with an example of why you might want to implement learning rate decay. Suppose you're implementing mini-batch gradient descent, with a reasonably small mini-batch. Maybe a mini-batch has just 64, 128 examples. Then as you iterate, your steps will be a little bit noisy. And it will tend towards this minimum over here, but it won't exactly converge. But your algorithm might just end up wandering around, and never really converge, because you're using some fixed value for alpha. And there's just some noise in your different mini-batches. But if you were to slowly reduce your learning rate alpha, then during the initial phases, while your learning rate alpha is still large, you can still have relatively fast learning. But then as alpha gets smaller, your steps you take will be slower and smaller. And so you end up oscillating in a tighter region around this minimum, rather than wandering far away, even as training goes on and on.

So the intuition behind slowly reducing alpha, is that maybe during the initial steps of learning, you could afford to take much bigger steps. But then as learning approaches converges, then having a slower learning rate allows you to take smaller steps. So here's how you can implement learning rate decay. Recall that one epoch is one pass,

Learning rate decay



Learning rate decay

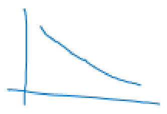
1 epoch = 1 pass through data.

$X^{\{1\}}$
$X^{\{2\}}$
epoch 1
epoch 2

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

$\alpha_0 = 0.2$
decay-rate $= 1$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮



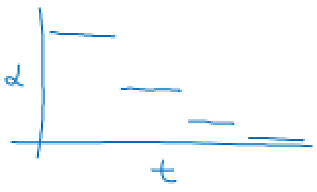
Andrew Ng

Other learning rate decay methods

formula

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

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

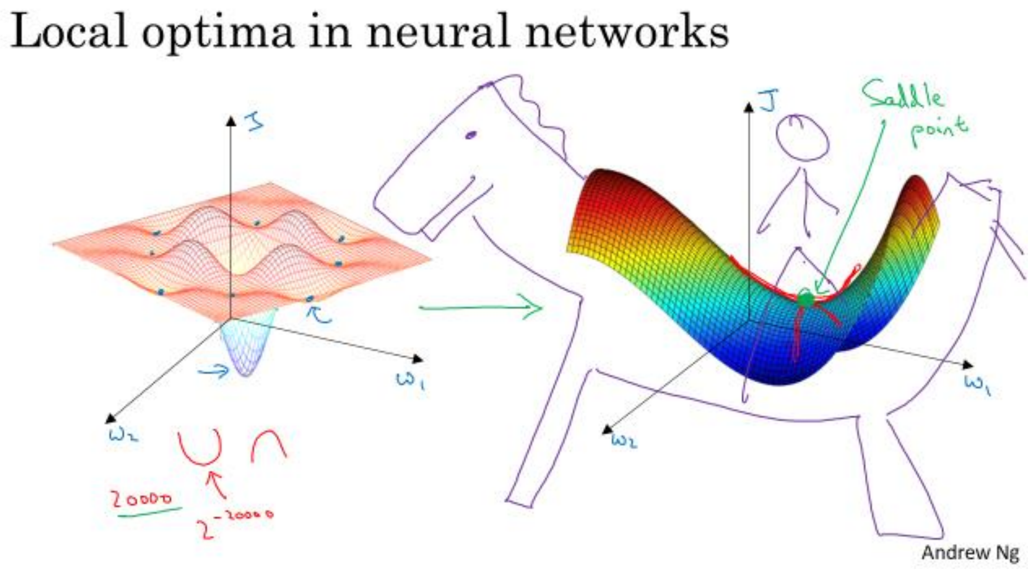


discrete staircase

Manual decay.

Andrew Ng

2.8 The problem of local optima



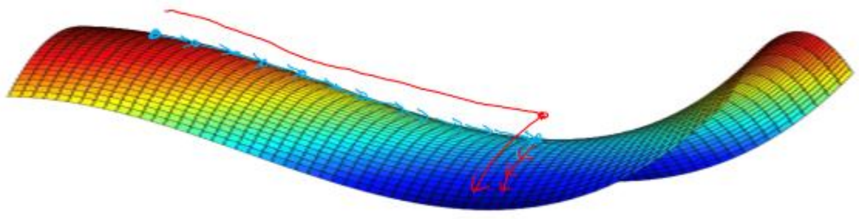
Local optima normally is not a problem.

In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in bad local optima. But as this theory of deep learning has advanced, our understanding of local optima is also changing. It turns out that if you are plotting a figure like this in two dimensions, then it's easy to create plots like this with a lot of different local optima. And these very low dimensional plots used to guide their intuition. But this intuition isn't actually correct. It turns out if you create a neural network, most points of zero gradients are not local optima like points like this. Instead most points of zero gradient in a cost function are saddle points. So, that's a point where the zero gradient, again, just is maybe W1, W2, and the height is the value of the cost function J. But informally, a function of very high dimensional space, if the gradient is zero, then in each direction it can either be a convex light function or a concave light function. And if you are in, say, a 20,000 dimensional space, then for it to be a local optima, all 20,000 directions need to look like this. And so the chance of that happening is maybe very small, maybe two to the minus 20,000. Instead you're much more likely to get some directions where the curve bends up like so, as well as some directions where the curve function is bending down rather than have them all bend upwards. So that's why in very high-dimensional spaces you're actually much more likely to run into a saddle point like that shown on the right, then the local optimum. As for why the surface is called a saddle point, if you can picture, maybe this is a sort of saddle you put on a horse, right?

And so, one of the lessons we learned in history of deep learning is that a lot of our intuitions about low-dimensional spaces, like what you can plot on the left, they really don't transfer to the very high-dimensional spaces that any other algorithms are operating over. Because if you have 20,000 parameters, then J as your function over 20,000 dimensional vector, then you're much more likely to see saddle points than local optimum. If local optima aren't a problem, then what is a problem? It turns out that plateaus can really slow down learning and a plateau is a region where the derivative is close to zero for a long time. So if you're here, then gradient descents will move down the surface, and because the gradient is zero or near zero, the surface is quite flat. You can actually take a very long time, you know, to slowly find your way to maybe this point on the plateau. And then because of a random perturbation of left or right, maybe then finally I'm going to search pen colors for clarity. Your algorithm can then find its way off the plateau. Let it take this very long slope off before it's found its way here and they could get off this plateau.

So the takeaways from this video are, first, you're actually pretty unlikely to get stuck in bad local optima so long as you're training a reasonably large neural network, save a lot of parameters, and the cost function J is defined over a relatively high dimensional space. But second, that plateaus are a problem and you can actually make learning pretty slow. And this is where algorithms like momentum or RmsProp or Adam can really help your learning algorithm as well. And these are scenarios where more sophisticated observation algorithms, such as Adam, can actually speed up the rate at which you could move down the plateau and then get off the plateau. So because your network is solving optimizations problems over such high dimensional spaces, to be honest, I don't think anyone has great intuitions about what these spaces really look like, and our understanding of them is still evolving. But I hope this gives you some better intuition about the challenges that the optimization algorithms may face.

Problem of plateaus

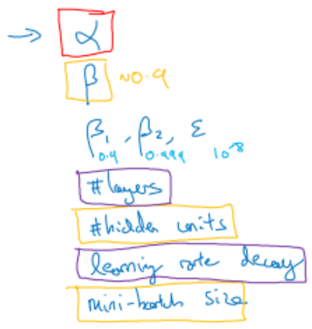


- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Week 3 Hyperparameter tuning, Batch Normalization and Programming Frameworks

3.1 Tuning process

Hyperparameters



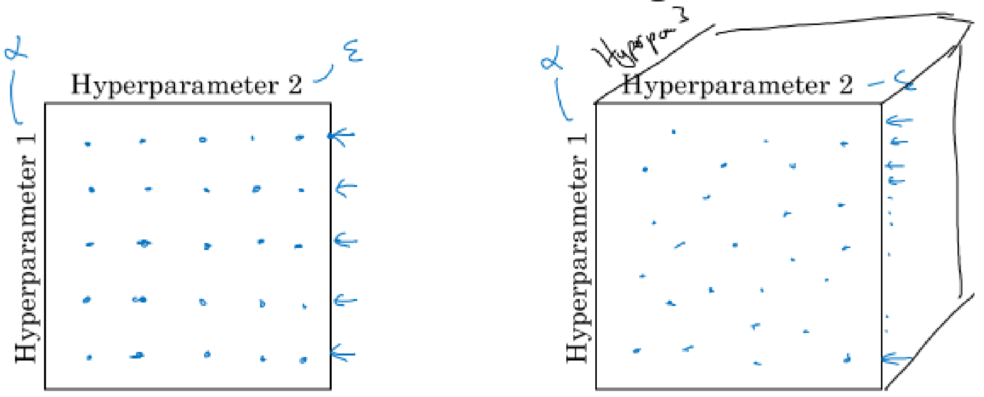
The learning rate, alpha, is the most important hyper-parameter to tune.

Momentum beta is importance, normally the default value 0.9 is fine

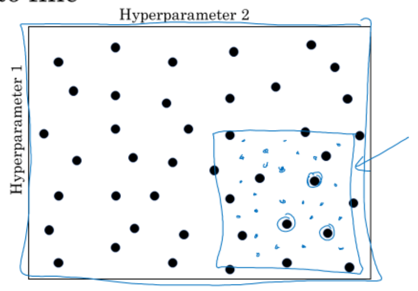
Hyper-parameters beta1, beta2 for Adam are important.

- Red: First important
- Yellow: Second important
- Purple: Third important

Try random values: Don't use a grid



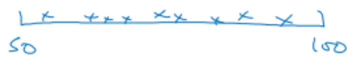
Coarse to fine



3.2Using an appropriate scale to pick hyper-parameters

Picking hyperparameters at random

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

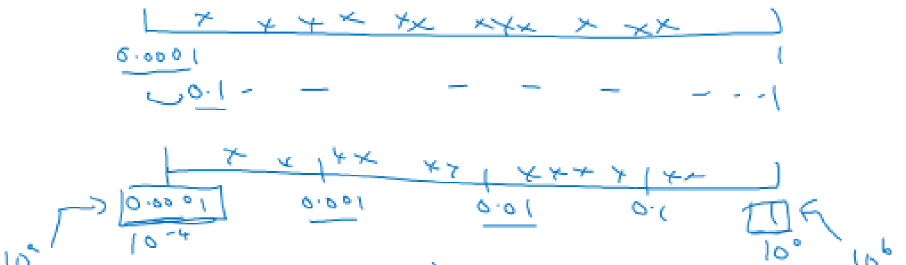


$\rightarrow$ #layers L: $2 - 4$

$2, 3, 4$

Appropriate scale for hyperparameters

$\alpha = 0.0001, \dots, 1$



$a = \log_{10} 0.0001 = -4$ $\quad r = -4 * \text{np.random.rand()} \leftarrow r \in [-4, 0]$ $\quad b = \log_{10} 1 = 0$

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

$\underline{10^a \dots 10^b}$ $\quad \underline{r \in [a, b]}$ $[-4, 0]$ $\quad \underline{\alpha = 10^r}$

Andrew Ng

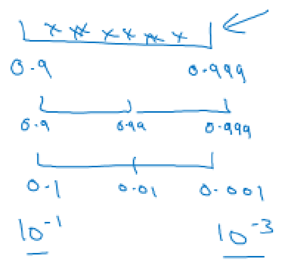
Hyperparameters for exponentially weighted averages

$\beta = 0.9 \quad \dots \quad 0.999$

$\downarrow \qquad \downarrow$

$10 \qquad 1000$

$1 - \beta = 0.1 \quad \dots \quad 0.001$



$\beta: 0.9000 \rightarrow 0.9005 \} \sim 10$

$\beta: 0.999 \rightarrow 0.9995$ $\quad \sim 1000 \quad \sim 2000$ $\quad \frac{1}{1-\beta}$

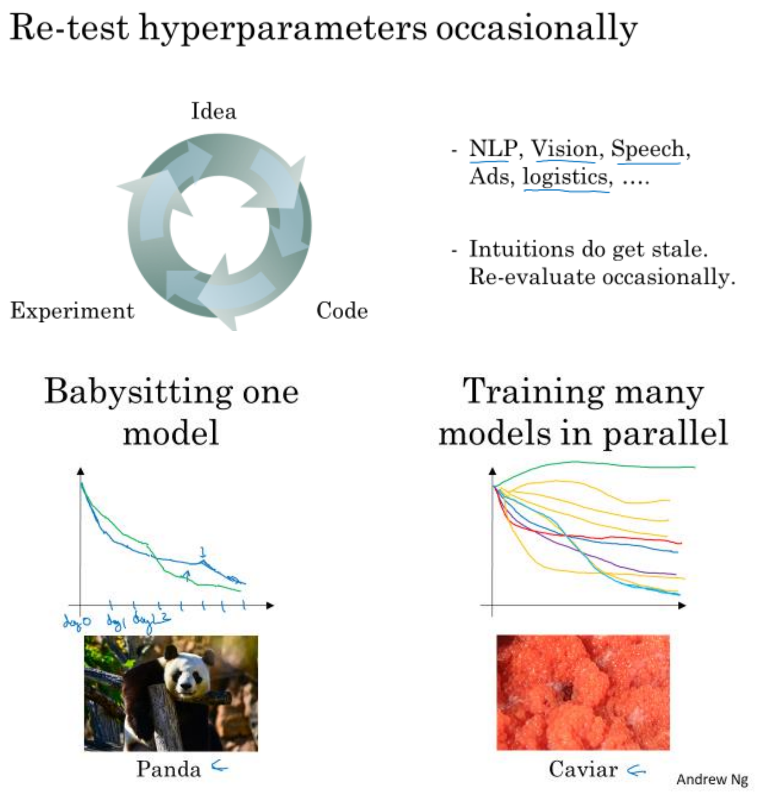
$r \in [-3, -1]$

$1 - \beta = 10^r$

$\beta = 1 - 10^r$

Andrew Ng

3.3Hyperparameters tuning in practice Pandas vs. Caviar

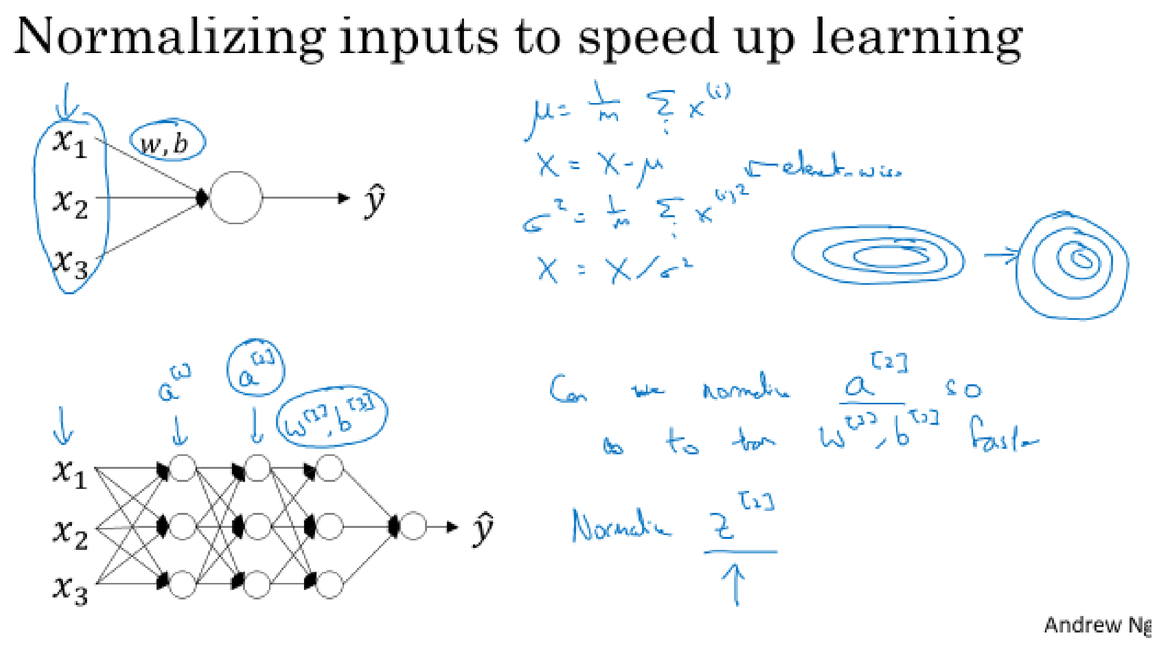


Re-test hyperparameters occasionally

- NLP, Vision, Speech, Ads, logistics, ….
- Intuitions do get stale. Re-evaluate occasionally.

3.4Normalizing activations in a network

Batch normalization



Normalizing inputs to speed up learning

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

$X = X - \mu$ ← element-wise

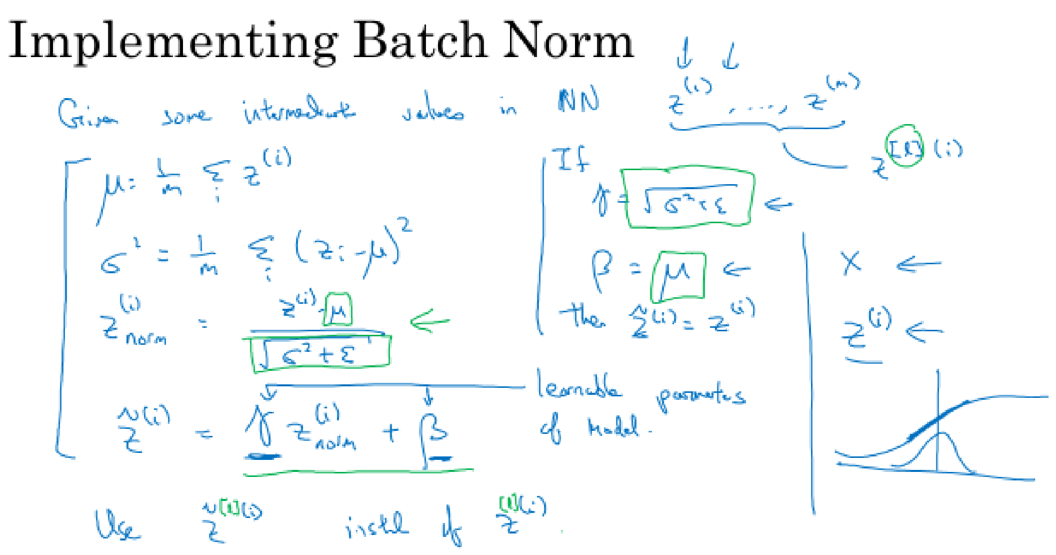
$\sigma^2 = \frac{1}{m} \sum_i x^{(i)2}$

$X = X / \sigma^2$

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

Normalize $z^{[2]}$

We actually normalize the value before activation function (Z), not after (a).



Implementing Batch Norm

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

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

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

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

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

learnable parameters of model.

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

$\beta = \boxed{\mu} \leftarrow$

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

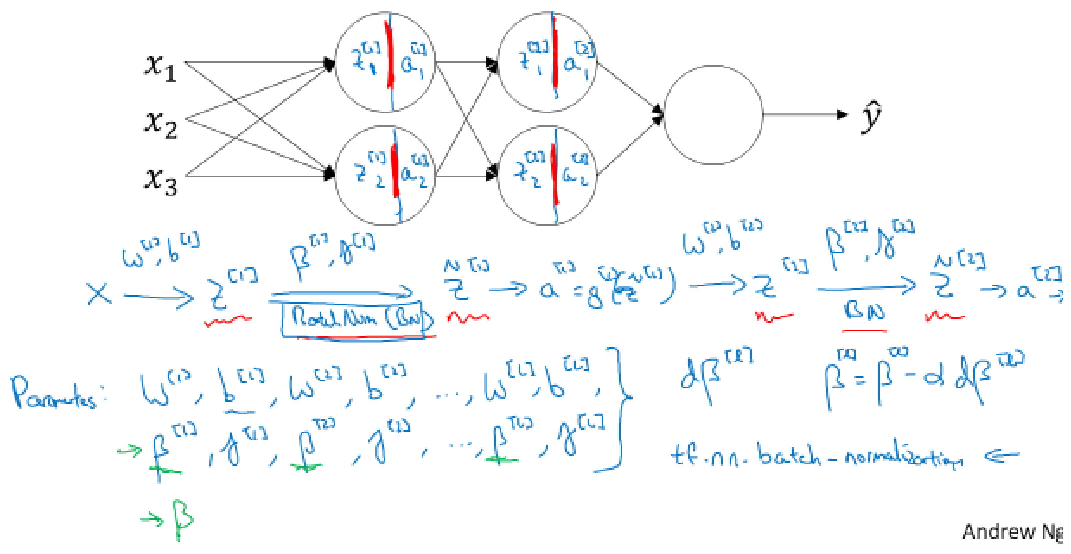
$x \leftarrow$

$z^{(i)} \leftarrow$

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

One difference between the training input and these hidden unit values is you might not want your hidden unit values be forced to have mean 0 and variance 1. For example, if you have a sigmoid activation function, you don't want your values to always be clustered here. You might want them to have a larger variance or have a mean that's different than 0, in order to better take advantage of the nonlinearity of the sigmoid function rather than have all your values be in just this linear regime. So that's why with the parameters gamma and beta, you can now make sure that your zi values have the range of values that you want. But what it does really is it then shows that your hidden units have standardized mean and variance, where the mean and variance are controlled by two explicit parameters gamma and beta which the learning algorithm can set to whatever it wants. So what it really does is it normalizes in mean and variance of these hidden unit values, really the zis, to have some fixed mean and variance. And that mean and variance could be 0 and 1, or it could be some other value, and it's controlled by these parameters gamma and beta.

3.5Fitting Batch Norm into a neural network



$$x \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]},\gamma^{[1]}} \tilde{z}^{[1]} \to a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]},\gamma^{[2]}} \tilde{z}^{[2]} \to a^{[2]} \to$$

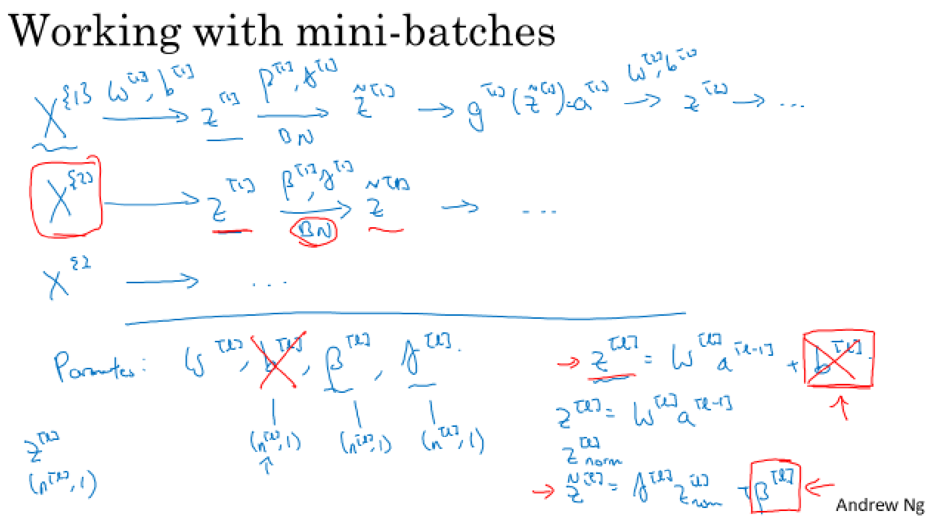
Parameters: $\left.\begin{array}{l} w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]} \\ \to \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]} \end{array}\right\}$

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

tf.nn.batch_normalization $\leftarrow$

$\to \beta$

Andrew Ng



Working with mini-batches

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

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

$X^{\{3\}} \rightarrow \dots$

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

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

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

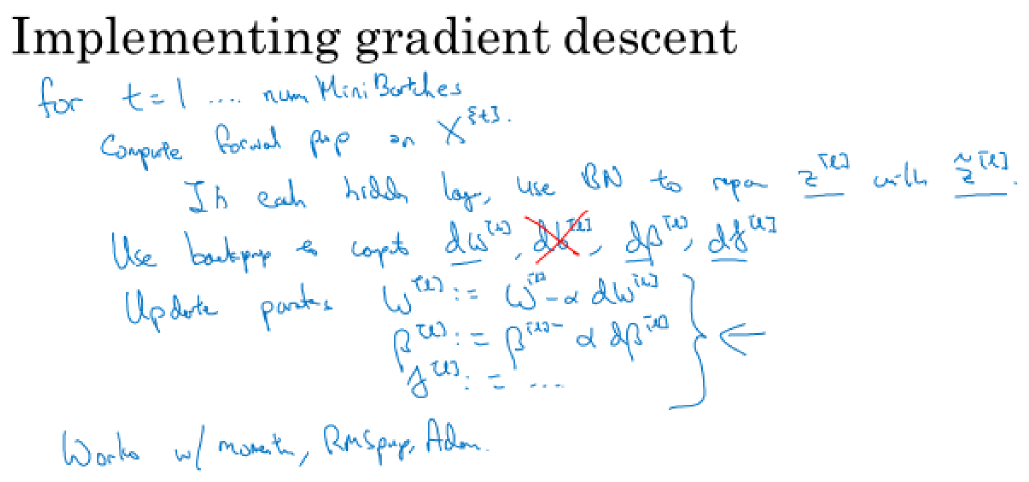
$\rightarrow z^{[l]} = W^{[l]} a^{[l-1]} + \boxed{\cancel{b^{[l]}}}$

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

$z^{[l]}_{norm}$

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

Andrew Ng



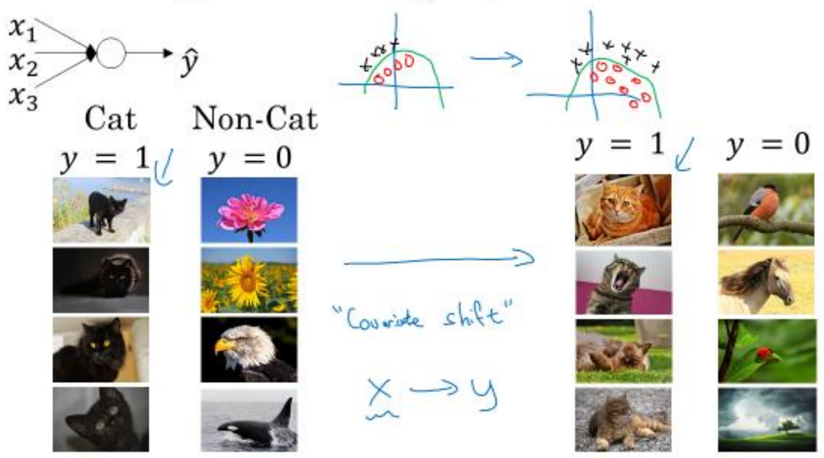
Implementing gradient descent

```
for t = 1 .... num MiniBatches
    Compute forward prop on $X^{\{t\}}$.
        In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$.
    Use backprop to compute $dW^{[l]}, \cancel{db^{[l]}}, d\beta^{[l]}, d\gamma^{[l]}$
    Update parameters $W^{[l]} := W^{[l]} - \alpha dW^{[l]}$
                      $\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$
                      $\gamma^{[l]} := \dots$
```

Works w/ momentum, RMSprop, Adam.

3.6Why does Batch Norm work

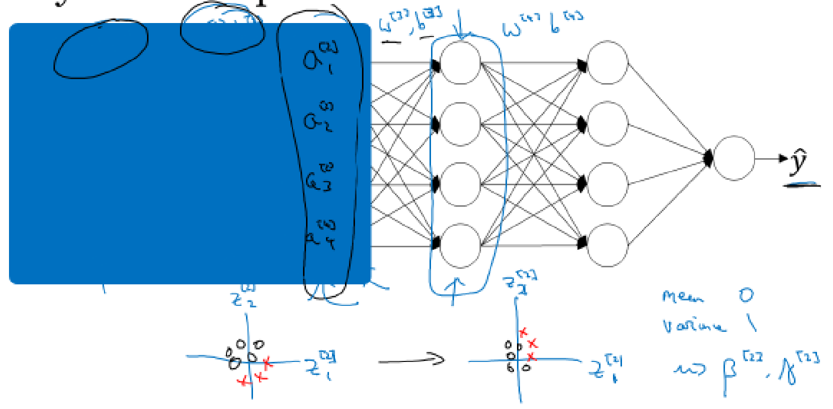
Learning on shifting input distribution



Why does batch norm work? Here's one reason, you've seen how normalizing the input features, the X's, to mean zero and variance one, how that can speed up learning. So rather than having some features that range from zero to one, and some from one to a 1,000, by normalizing all the features, input features X, to take on a similar range of values that can speed up learning. So, one intuition behind why batch norm works is, this is doing a similar thing, but further values in your hidden units and not just for your input there.

This idea of your data distribution changing goes by the somewhat fancy name, covariate shift. And the idea is that, if you've learned some X to Y mapping, if the distribution of X changes, then you might need to retrain your learning algorithm. And this is true even if the function, the ground true function, mapping from X to Y, remains unchanged, which it is in this example, because the ground true function is, is this picture a cat or not. And the need to retain your function becomes even more acute or it becomes even worse if the ground true function shifts as well.

Why this is a problem with neural networks?



Batch Norm as regularization

X

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch. $X^{\{t\}}$

$\tilde{z}^{[l]}$ 64, 128 $z^{[l]}$

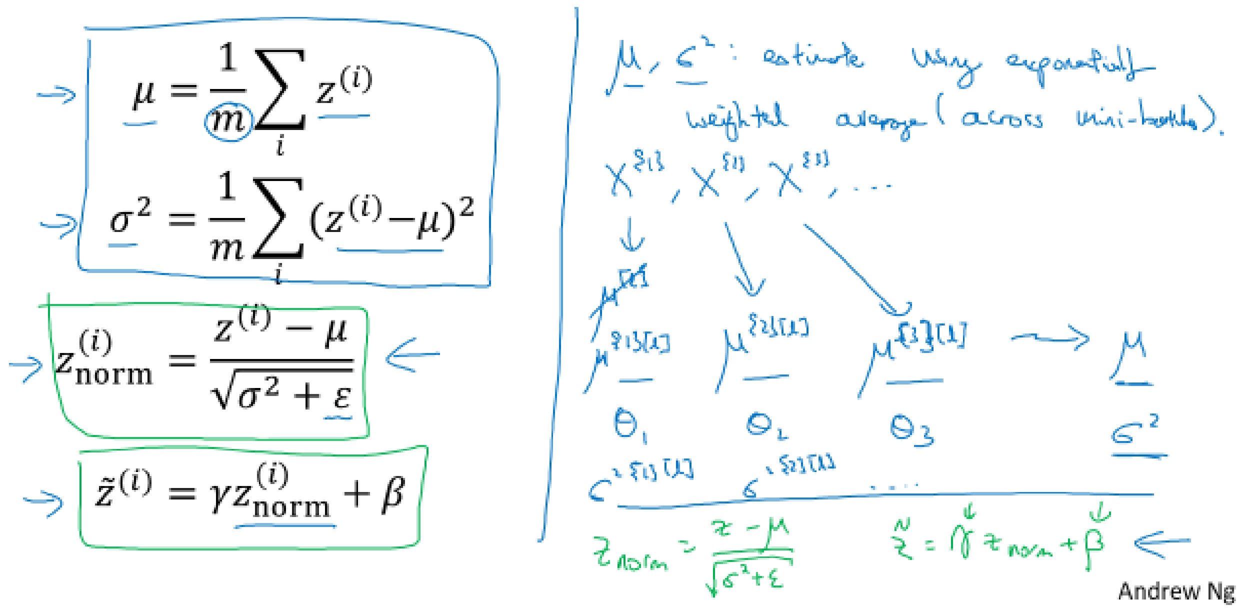
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.

μ, σ^2

- This has a slight regularization effect.

mini-batch : 64 $\longrightarrow$ 512

3.7Batch Norm at test time



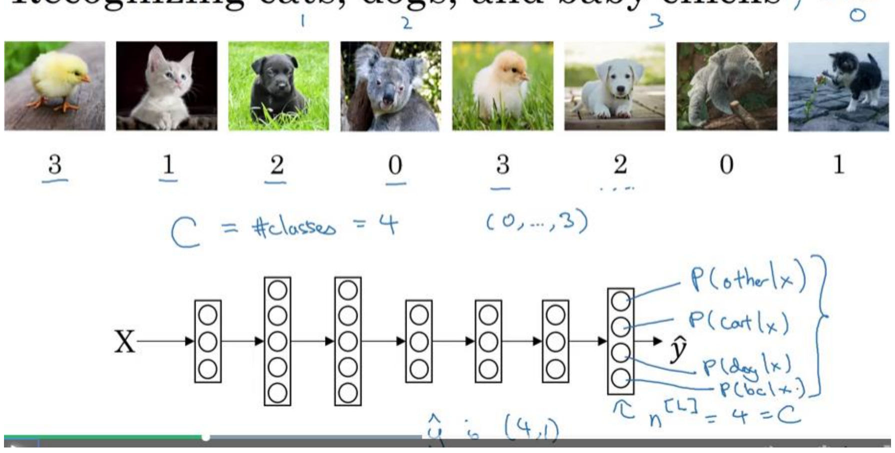
$\mu = \frac{1}{m}\sum_i z^{(i)}$
$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)}-\mu)^2$
$z_{\text{norm}}^{(i)} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\varepsilon}}$
$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$
μ, σ²: estimate using exponentially weighted average (across mini-batches).
X{1}, X{2}, X{3}, ...
μ{1}[l] μ{2}[l] μ{3}[l] → μ
θ1 θ2 θ3
σ²{1}[l] σ²{2}[l] ... σ²
z_norm = (z − μ)/√(σ²+ε)
z̃ = γ z_norm + β
Andrew Ng

So the takeaway from this is that during training time mu and sigma squared are computed on an entire mini batch of say 64 engine, 28 or some number of examples. But that test time, you might need to process a single example at a time. So, the way to do that is to estimate mu and sigma squared from your training set and there are many ways to do that. You could in theory run your whole training set through your final network to get mu and sigma squared. But in practice, what people usually do is implement and exponentially weighted average where you just keep track of the mu and sigma squared values you're seeing during training and use and exponentially the weighted average, also sometimes called the running average, to just get a rough estimate of mu and sigma squared and then you use those values of mu and sigma squared that test time to do the scale and you need the head and unit values Z. In practice, this process is pretty robust to the exact way you used to estimate mu and sigma squared. So, I wouldn't worry too much about exactly how you do this and if you're using a deep learning framework, they'll usually have some default way to estimate the mu and sigma squared that should work reasonably well as well. But in practice, any reasonable way to estimate the mean and variance of your head and unit values Z should work fine at test.

3.8Softmax Regression

There's a generalization of logistic regression called Softmax regression. The less you make predictions where you're trying to recognize one of C or one of multiple classes, rather than just recognize two classes

Recognizing cats, dogs, and baby chicks, other

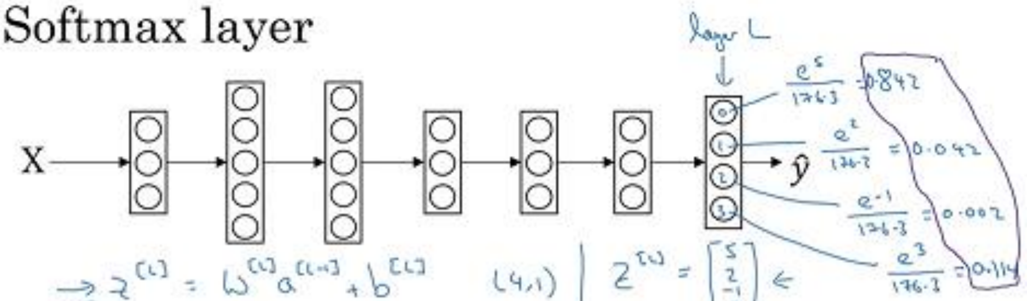


C = #classes = 4 (0, …, 3)

$n^{[L]} = 4 = C$

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

Softmax layer



layer L
X
$\hat{y}$
0
1
2
3
$\frac{e^5}{176.3} = 0.842$
$\frac{e^2}{176.3} = 0.042$
$\frac{e^{-1}}{176.3} = 0.002$
$\frac{e^3}{176.3} = 0.114$

$\rightarrow z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]}$ (4,1)

Activation function:

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

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

$\boxed{(4,1)}$ $\hat{y}$

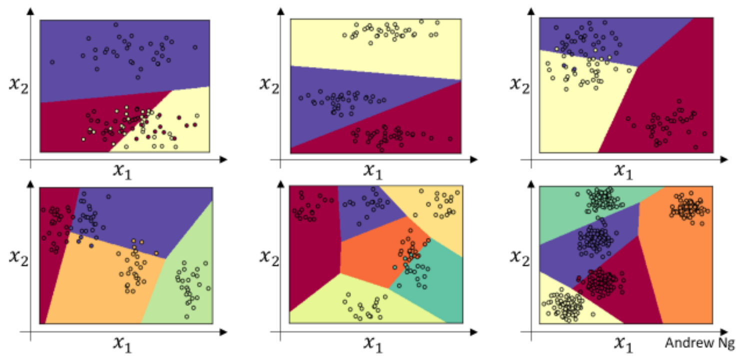
$a^{[L]} = g^{[L]}(z^{[L]})$

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

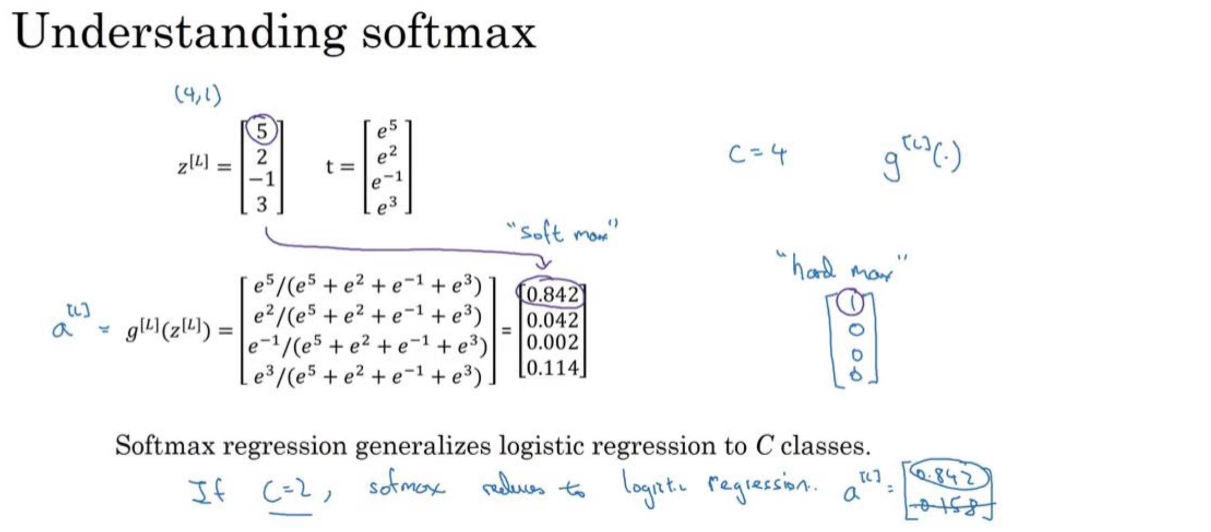
$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}, \quad \sum_{j=1}^{4} t_j = 176.3$

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

Softmax examples



3.9 Training a softmax classifier



Understanding softmax

(4,1)

$z^{[L]} = \begin{bmatrix} \boxed{5} \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$ $\qquad C=4 \qquad g^{[L]}(\cdot)$

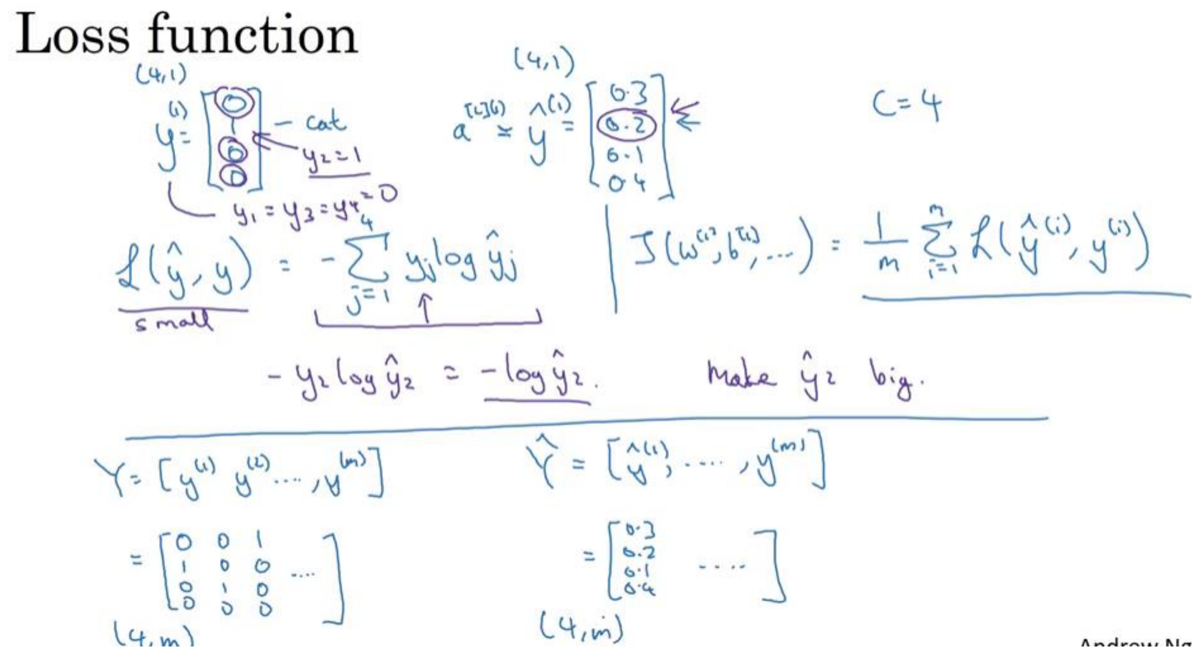
"soft max"

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

"hard max" $\begin{bmatrix} \boxed{1} \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Softmax regression generalizes logistic regression to C classes.

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



Loss function

(4,1) $\qquad$ (4,1)

$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ — cat, $y_2 = 1$, $y_1 = y_3 = y_4 = 0$ $\qquad a^{[L](i)} = \hat{y}^{(i)} = \begin{bmatrix} 0.3 \\ \boxed{0.2} \\ 0.1 \\ 0.4 \end{bmatrix}$ $\qquad C=4$

$\underbrace{\mathcal{L}(\hat{y}, y)}_{\text{small}} = \underbrace{-\sum_{j=1}^{4} y_j \log \hat{y}_j}$ $\qquad \Big| \qquad J(w^{[1]}, b^{[1]}, \dots) = \underline{\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$

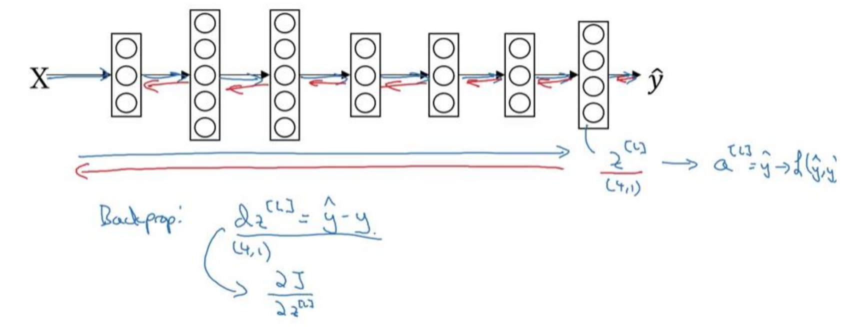
$-y_2 \log \hat{y}_2 = \underline{-\log \hat{y}_2}$. Make $\hat{y}_2$ big.

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

$= \underset{(4,m)}{\begin{bmatrix} 0 & 0 & 1 & \\ 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}}$ $\qquad = \underset{(4,m)}{\begin{bmatrix} 0.3 & \\ 0.2 & \dots \\ 0.1 & \\ 0.4 & \end{bmatrix}}$

Andrew Ng

Gradient descent with softmax


X
$\hat{y}$
$\underset{(4,1)}{z^{[L]}} \longrightarrow a^{[L]} = \hat{y} \to \mathcal{L}(\hat{y}, y)$
Backprop: $\underset{(4,1)}{dz^{[L]}} = \hat{y} - y$
$\frac{\partial J}{\partial z^{[L]}}$

3.10Deep 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)

3.11TensorFlow

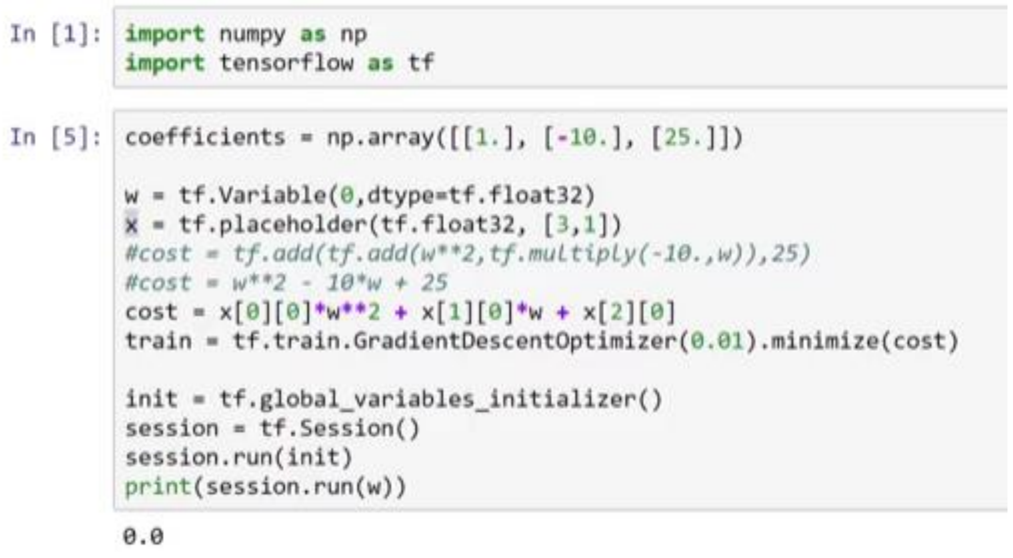
Motivating problem

$$\underset{(\text{cost})}{J(w)} = \boxed{w^2 - 10w + 25}$$

$$(w-5)^2$$

$$w = 5$$

$$J(\overset{\uparrow}{w}, \overset{\uparrow}{b})$$



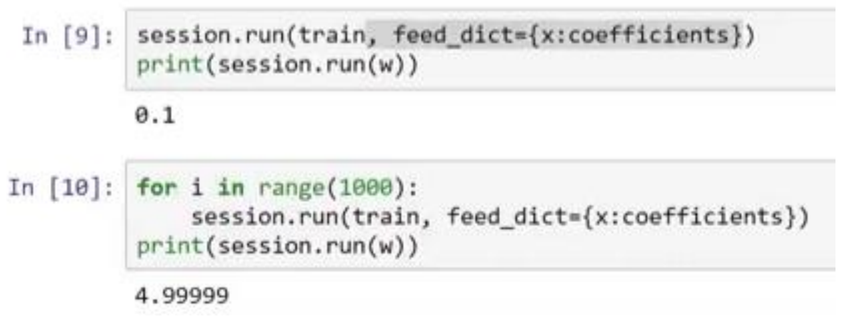
```
In [1]: import numpy as np
        import tensorflow as tf
```

```
In [5]: coefficients = np.array([[1.], [-10.], [25.]])

        w = tf.Variable(0,dtype=tf.float32)
        x = tf.placeholder(tf.float32, [3,1])
        #cost = tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        #cost = w**2 - 10*w + 25
        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(init)
        print(session.run(w))
```

```
0.0
```



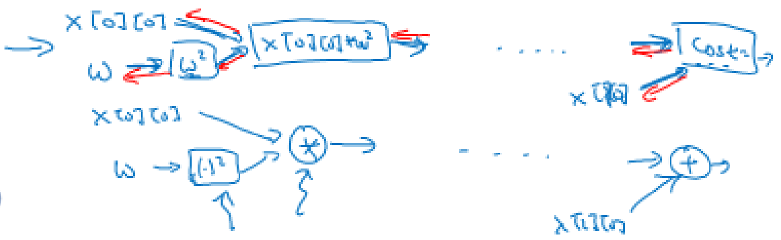
```
In [9]:  session.run(train, feed_dict={x:coefficients})
         print(session.run(w))

         0.1

In [10]: for i in range(1000):
             session.run(train, feed_dict={x:coefficients})
         print(session.run(w))

         4.99999
```

Code example



```
import numpy as np
import tensorflow as tf

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

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

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

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

Andrew Ng