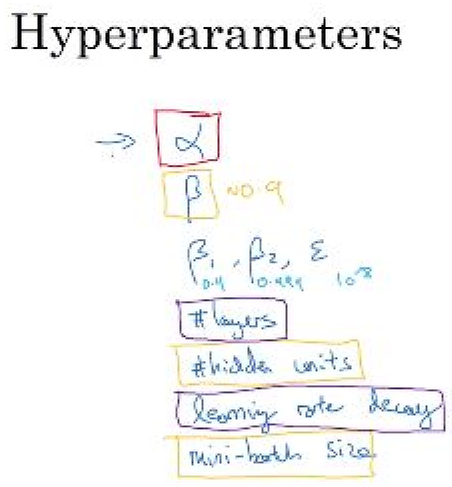
Andrew Ng's Improving Deep Neural Networks Class

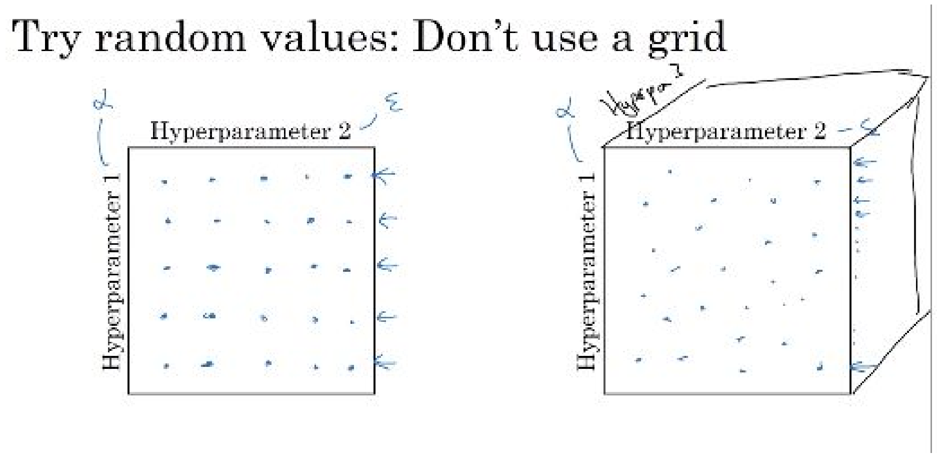
Week 3: Hyperparameter Tuning, Batch Normalization and Programming Frameworks

Tuning Process

- One of the challenges with hyperparameter tuning is the number of parameters to tune. How do we prioritize which ones to tune? The most important one to tune is the learning rate α, then β, $\#\ hidden\ units\ \&\ mini-batch\ size$ .Finally you look at $\#\ layers\ \&\ learning\ rate\ decay$. Professor Ng has never had to tune $\beta_{1,}\ \beta_2 \&\ \varepsilon$ and uses the default values of 0.9, 0.999 and 10^{-8}.

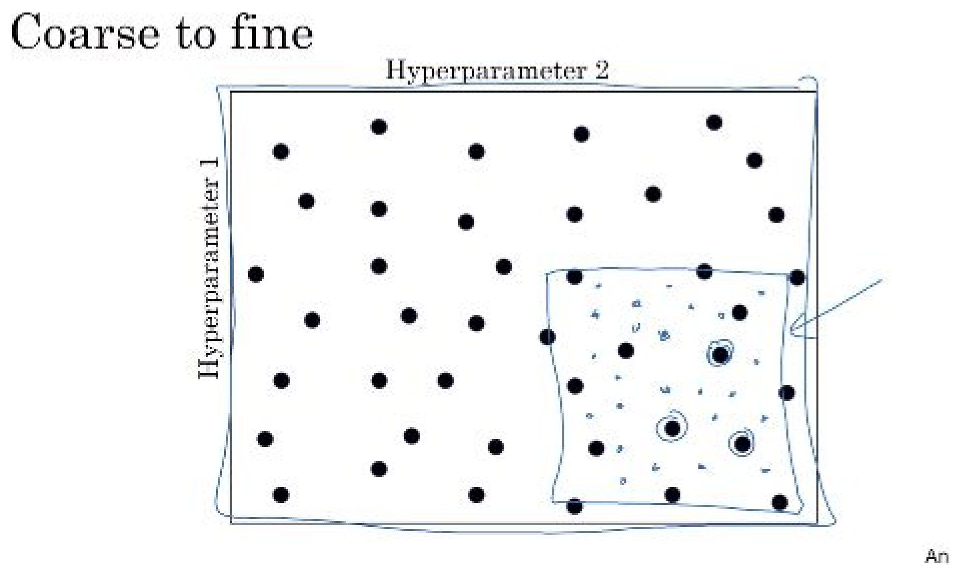


- How do we choose what set of values to use for each parameter to tune? An earlier generation of ML used a grid where you systematically explore these values.



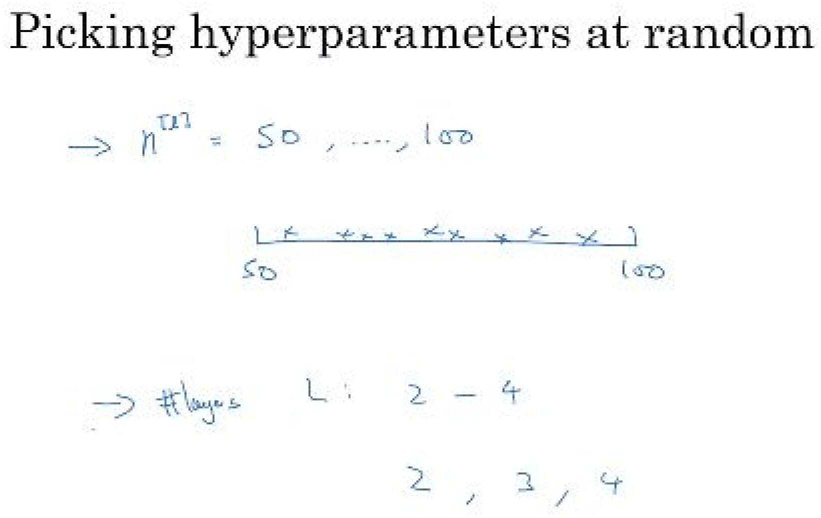
Professor Ng doesn't recommend we use a grid because the grid does not prioritize the most important hyperparameter to tune. Instead, he suggests we don't use a grid and try random values; this practice ensures you have more choices for the most important hyperparameters.

- Another way to go is to use the Coarse to Fine method. In this method you find a few points that work well and then you decide to concentrate in that smaller square and try a few more points there.

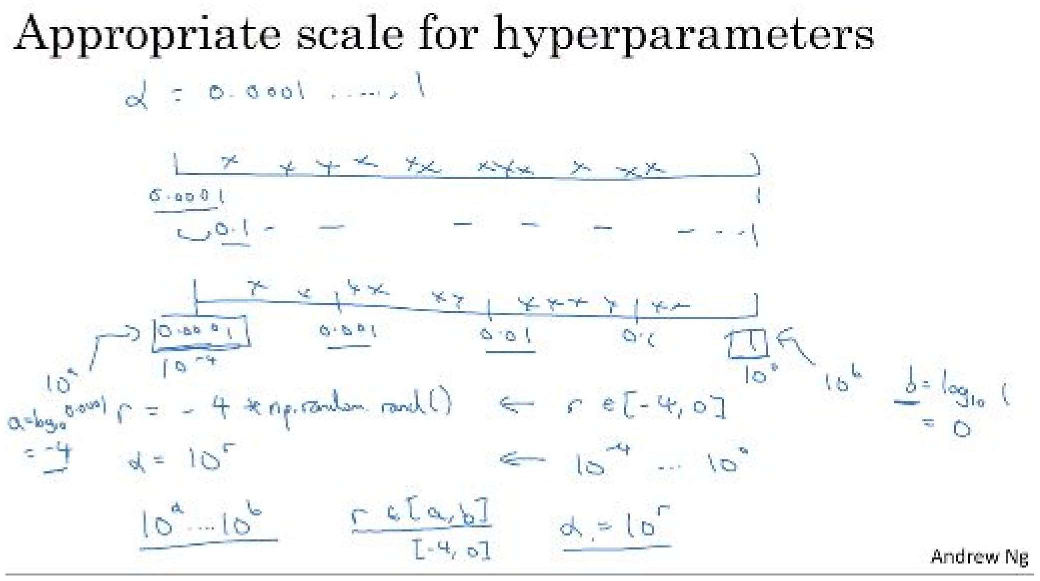


Using an appropriate scale to pick hyperparameters

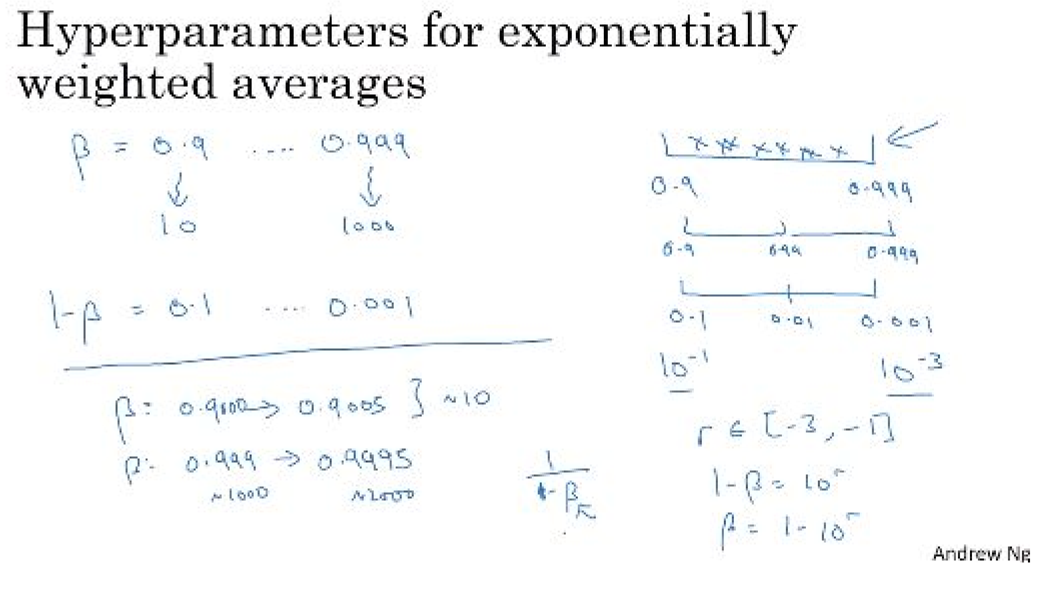
- In the examples below, random sampling over a range (50-100, 2-4, etc.) is a reasonable thing to do but it is not true for all hyperparameters.



- Let's take for example α and let's assume suitable values are between 0.0001 and 1. If we sample uniformly random on the number line for this range we would end up spending most of our resources (90%) checking values between 0.1 and 1 which seems wrong. What we do in these cases is we actually sample using the logarithmic scale and we sample uniformly random on this scale. In python we do this by having $r = 4 * np.random.rand()$ *and* $\alpha = 10^r$. r would be between [-4, 0] so we end up with a range for alpha of $10^{-4}...10^0$. The general rule is $10^a...10^b,\ r\ \varepsilon\ [a,\ b]$ *and* $\alpha = 10^r$. And this is how we random sample on this scale.

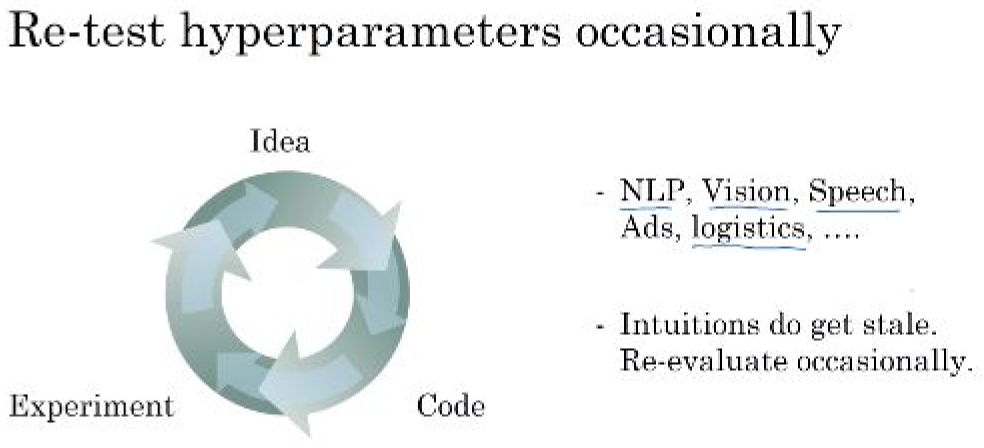


- Another tricky case is for picking the scale for exponentially weighted averages. Let's assume we have β *between* 0.9 *and* 0.999. If we sample over the number line for this range we have the same case as before, so what we do instead is to use a range of $1 - \beta = 0.1...0.001$. Using the same method as before our range is from 10^{-1} *to* 10^{-3} *so* $r\ \varepsilon\ [-3,\ -1],\ 1 - \beta = 10^r$ *so* $\beta = 1 - 10^r$.



Hyperparameter tuning in practice: Pandas vs Caviar

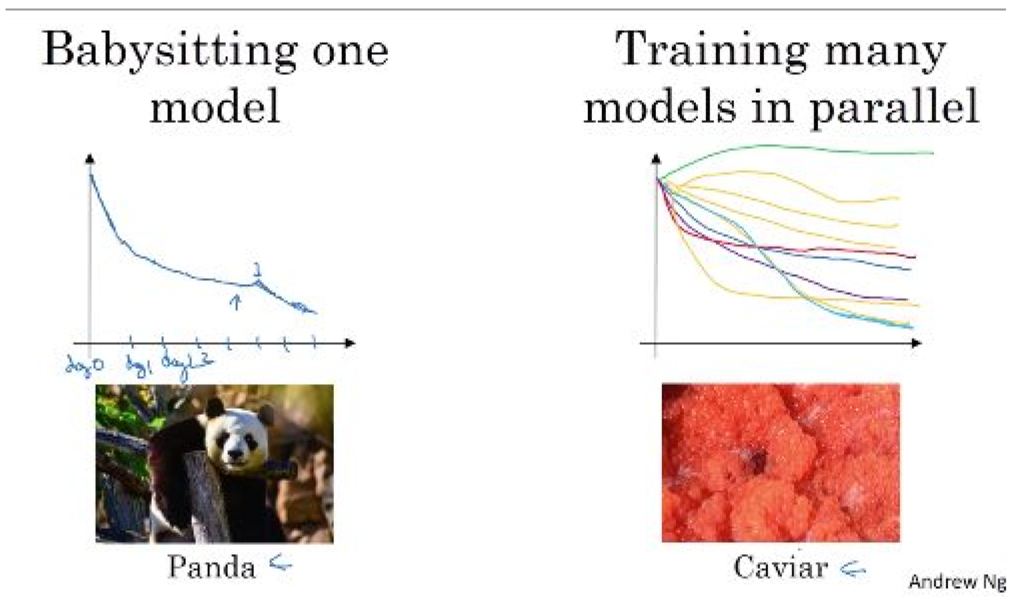
- We are going to wrap up this section by going over a couple of more practical tips. The first one is that intuitions get stale over time so we have to retest hyper parameters every few months.



Re-test hyperparameters occasionally

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

- How do we go searching for hyperparameters? We have two ways of doing it:
  - The Panda approach (on the left) where we are babysitting one model and say, we are daily checking how the model is doing and we tweak it daily; we might increase the learning rate one day; we might do something different the next day.
  - The Caviar approach (on the right) where we have the capacity to train multiple models in parallel.



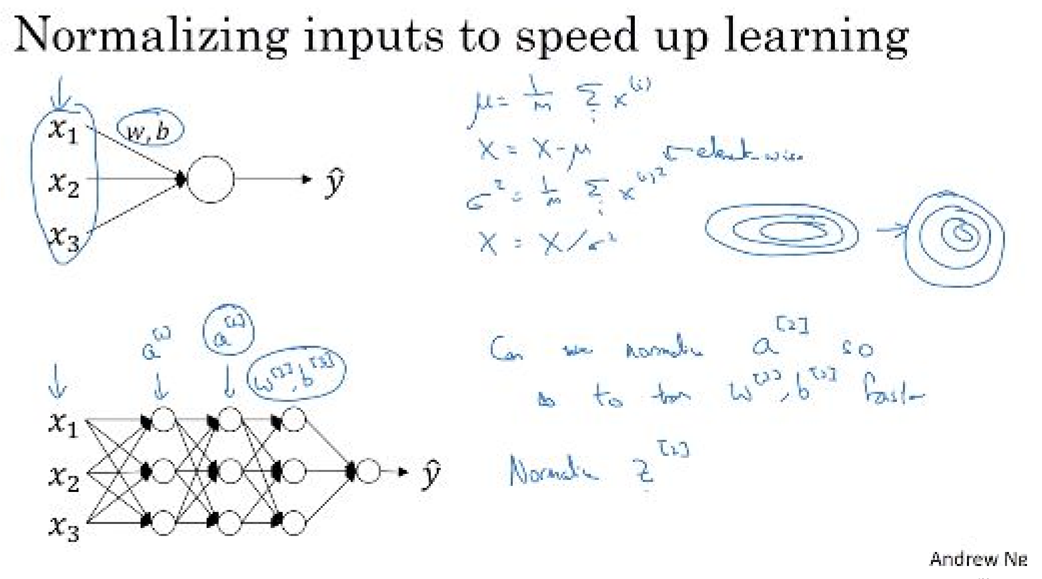
Babysitting one model

Training many models in parallel

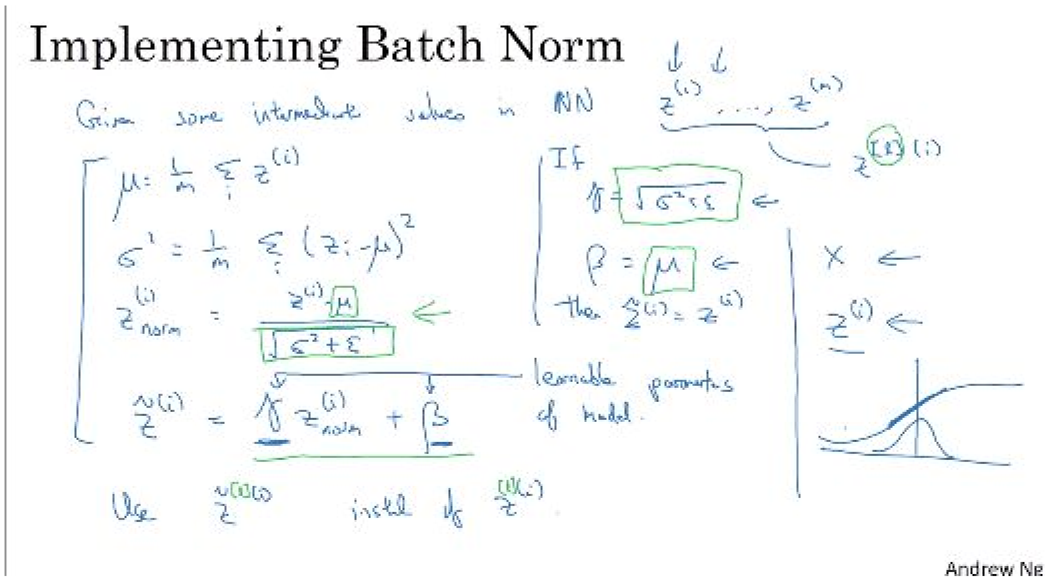
Usually the deciding factor is the amount of computational power we have. If we don't have a lot we go with the panda approach; if we have a lot of power we can try the caviar approach.

Batch Normalization: Normalizing activations in a network

- We saw in the past how normalization helps to speed up learning. If we are using logistic regression (the one neuron neural network) we compute the mean, subtract it from X and then compute σ^2 and divide X/σ^2 . This allows us to move from a learning problem where we have elongated contours to a more round learning problem where gradient descent will be able to optimize our learning faster. How would we do the same with a neural network? In the example below wouldn't it be nice if we can normalize $a^{[2]}$ so that $w^{[3]} and\ b^{[3]}$ can be trained faster? We will see how to do it by normalizing $z^{[2]}$.


Normalizing inputs to speed up learning
x_1 x_2 x_3 w, b $\hat{y}$
$\mu = \frac{1}{m}\sum_i x^{(i)}$
$X = X - \mu$
element-wise
$\sigma^2 = \frac{1}{m}\sum_i x^{(i)2}$
$X = X/\sigma^2$
$a^{[1]}$ $a^{[2]}$ $w^{[3]}, b^{[3]}$
Can we normalize $a^{[2]}$ so as to train $w^{[3]}, b^{[3]}$ faster
Normalize $z^{[2]}$
Andrew Ng

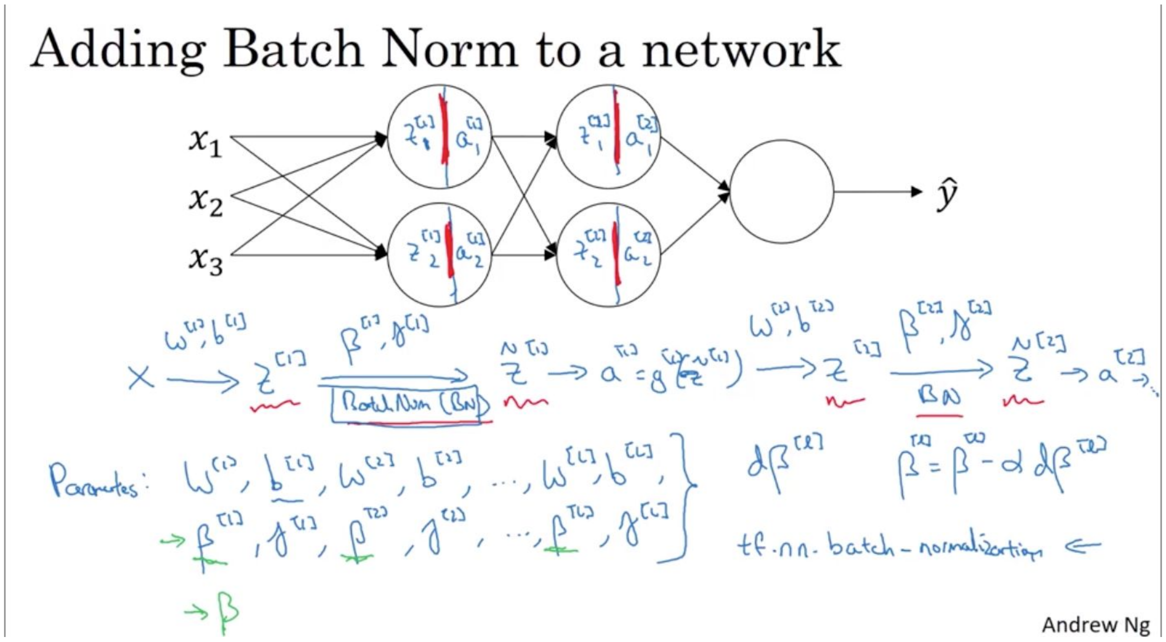
- Let's see how we implement batch norm: we start by calculating $\mu = 1/m\ * \sum_{i=1}^{m} z^{(i)}$. We then compute the variance: $\sigma = 1/m * \sum_{i=1}^{m} (z^{(i)} - \mu)^2$ and finally $z^{(i)}_{norm} = z(i) * \mu/\sqrt{\sigma^2 + \varepsilon}$.


Implementing Batch Norm
Given some intermediate values in NN $z^{(1)}, \dots, z^{(m)}$
$z^{[l](i)}$
$\mu = \frac{1}{m}\sum_i z^{(i)}$
$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$
$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$
$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$
learnable parameters of model
If $\gamma = \sqrt{\sigma^2 + \varepsilon}$
$\beta = \mu$
then $\tilde{z}^{(i)} = z^{(i)}$
x
$z^{(i)}$
Use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$
Andrew Ng

Now all z values have 0 mean and standard variance but maybe we don't want all hidden neurons in the layer to have the same mean and variance. So we compute a new variable $\tilde{z}^{(i)} = \gamma * z^{(i)}_{norm} + \beta$ and we use $\tilde{z}^{(i)}$ instead of $z^{(i)}$. Both $\gamma\ and\ \beta$ are learnable parameters. The effect of both of these parameters is that it allows us to set the mean of $\tilde{z}^{(i)}$ to whatever we want it to be. Summarizing, our intuition is that we saw how normalizing the inputs X can help learning in a neural network. We also saw how batch norm applies normalization not only to the input layer but also to the units in one of the hidden layers of the neural network.

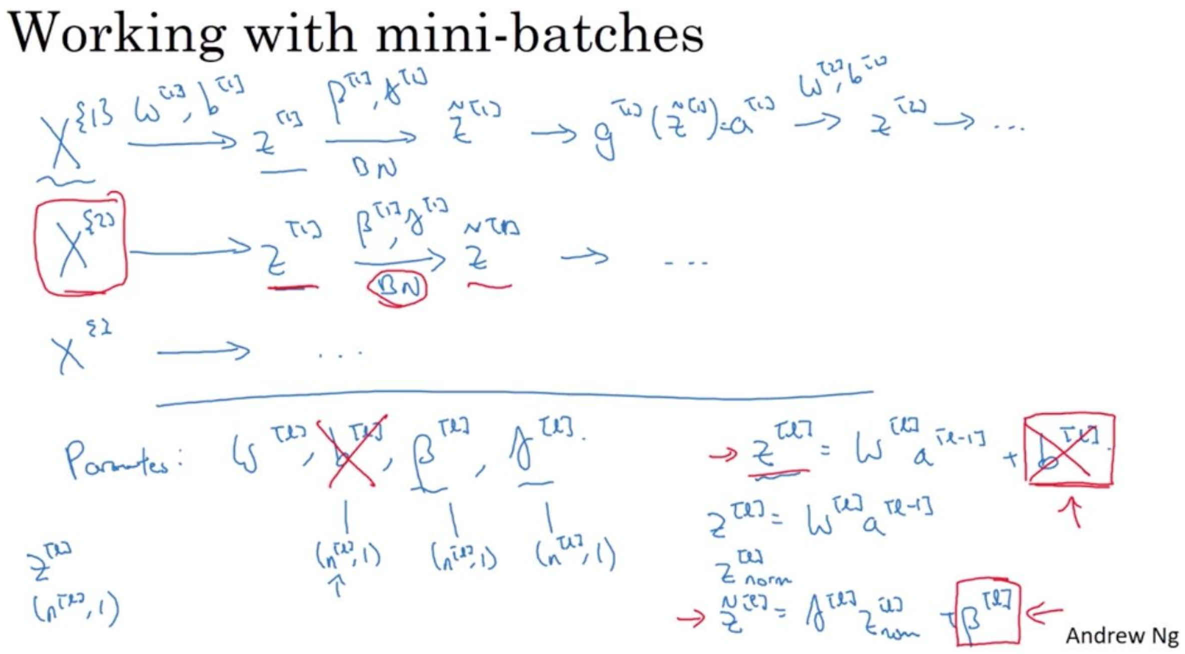
Batch Normalization: Fitting batch norm into a neural network

- How and where do we add batch norm to a neural network? It is added between calculating z and applying the activation function a. Using the neural network below, we can add batch norm as follows:
    - For the first layer we do the following:
        - $X\ (guided\ by\ parameters\ w^{[1]} \&\ b^{[1]}) \rightarrow \tilde{Z}^{[1]}(we\ apply\ batch\ norm\ using\ \beta^{[1]} \&\ \gamma^{[1]})\ \rightarrow a^{[1]}$
        - $a^{[1]} =\ g(\tilde{z}^{[1]})$
    - For the second layer we do the following:
        - $a^{[1]} \rightarrow z^{[2]}(guided\ by\ parameters\ w^{[2]} \&\ b^{[2]}) \rightarrow Z^{[2]}(we\ apply\ batch\ norm\ with\ \beta^{[2]} \&\ \gamma^{[2]}) \rightarrow a^{[2]}$


Adding Batch Norm to a network
x_1
x_2
x_3
$\hat{y}$
$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow$
Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$
$\rightarrow \beta$
$d\beta^{[l]}$
$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$
tf.nn.batch-normalization
Andrew Ng

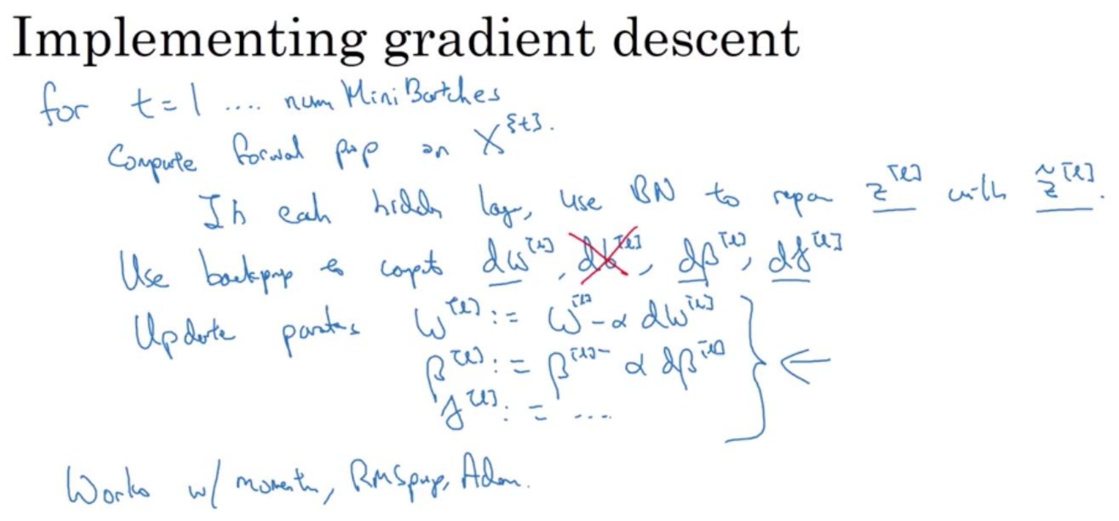
The parameters then for our neural network are now $w^{[1]},\ b^{[1]},\ w^{[2]},\ b^{[2]} \dots w^{[L]},\ b^{[L]}$ and $\beta^{[1]},\ \gamma^{[1]},\ \beta^{[2]},\ \gamma^{[2]}, \dots,\ \beta^{[L]},\ \gamma^{[L]}$. Notice we know have two betas: β for batch norm and our regular hyperparameter β. In practice, deep learning frameworks have a function to implement batch norm; the tensor flow function is $tf.nn.batch - normalization$.

- The previous example assumed we were using the whole training set so we were using batch gradient descent. In practice, batch norm is used with mini batches. We do the same calculations as above but for each mini batch. For the first mini batch we would do the following: $X^{\{1\}} \rightarrow z^{[1]}(guided\ by\ w^{[1]} \&\ b^{[1]}) \rightarrow \tilde{z}^{[1]}(batch\ norm\ with\ \beta^{[1]} \&\ \gamma^{[1]}) \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$ and so on for the other mini batches.



Something else to notice is that we don't need the parameter $b^{[L]}$ so our equations change: $z^{[l]} = w^{[l]}a^{[l-1]}$, we compute $z_{norm}^{[l]}$ and $\tilde{z}^{[l]} = \delta^{[l]} * z_{norm}^{[l]} + \beta^{[l]}$.
- Putting everything together for one step of gradient descent we then have:
  - $for\ t = 1 ... num\ of\ mini\ batches$:
    - $compute\ forward\ prop\ using\ mini\ batch\ X^{\{t\}}$.
    - $In\ each\ hidden\ layer,\ use\ BN\ to\ get\ \tilde{z}^{[l]}$
  - $Use\ backprop\ to\ compute\ dw^{[l]},\ \ dw^{[l]},\ db^{[l]},\ d\beta^{[l]},\ d\gamma^{[l]}$
  - $Update\ parameters:\ w^{[l]} =\ w^{[l]} - \alpha * dw^{[l]},\ \beta^{[l]} =\ \beta^{[l]} - \alpha * d\beta^{[l]}\ \&\ \gamma^{[l]} =\ \gamma^{[l]} - \alpha * d\gamma^{[l]}$

  Notice we can use bn with momentum, RSMProp and Adam, not only with gradient descent.

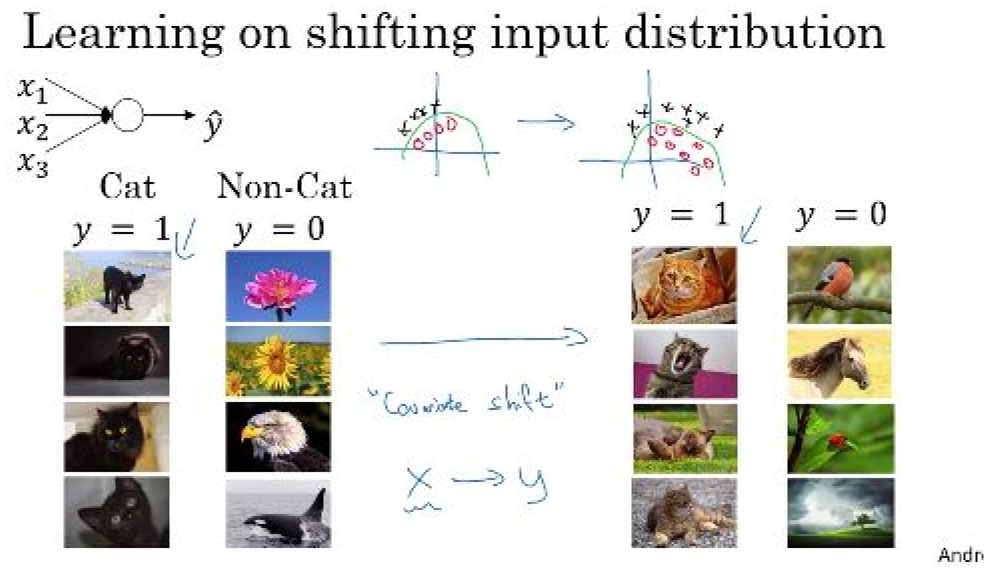


Implementing gradient descent

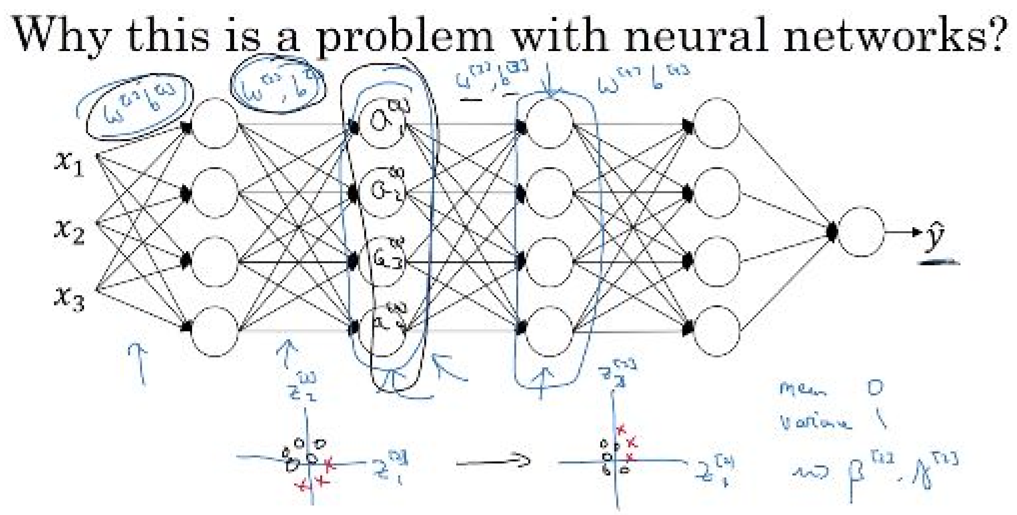
```
for $t = 1 \ldots$ numMiniBatches
    Compute forward prop on $X^{\{t\}}$.
        In each hidden layer, use BN to replace $\underline{z^{[l]}}$ with $\underline{\tilde{z}^{[l]}}$.
    Use backprop to compute $\underline{dW^{[l]}}$, ~~$db^{[l]}$~~, $\underline{d\beta^{[l]}}$, $\underline{d\gamma^{[l]}}$
    Update params $W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$
                  $\beta^{[l]} := \beta^{[l]} - \alpha\, d\beta^{[l]}$
                  $\gamma^{[l]} := \ldots$
Works w/ momentum, RMSprop, Adam.
```

Why does Batch Normalization work?

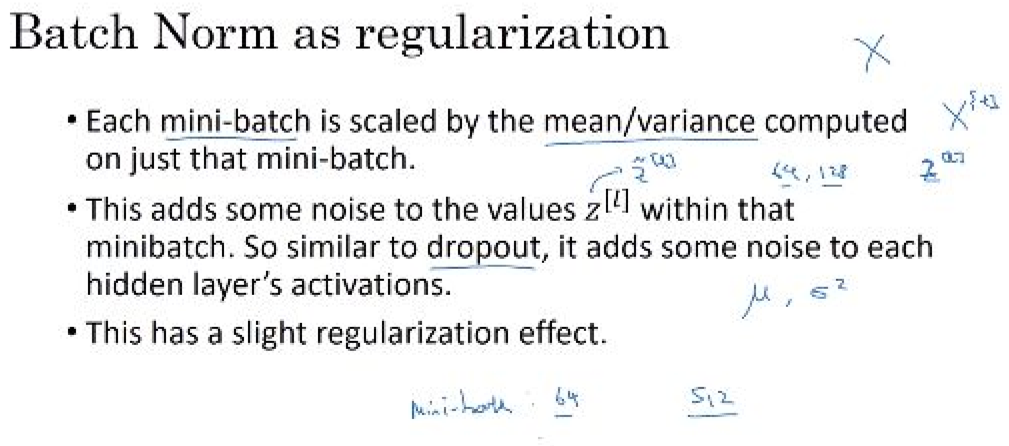
- Let's start by understanding what is called covariate shift: we have a shallow neural network that has been trained with the training set on the left so it knows how to identify a black cat. However, if we 'shift' the input X (the training set) from black cats to 'colored' cats as shown on the right our network will not do well so we need to retrain it.



- Why is covariate shifting a problem for neural networks? Let's illustrate it with the neural network below. We'll concentrate on the 3rd hidden layer. For this layer, the values $a_1^{[2]}, a_2^{[2]}, a_3^{[2]}, a_4^{[2]}$ are changing all the time (the covariate shift problem) because those values are dependent on the forward propagation calculation of the previous hidden layers. What batch norm does is reduce the amount of 'changing' these values go through by ensuring the mean and variance remain the same. Another way of saying this is that batch norm limits the amount to which updating the parameters in earlier layers affects the distribution of the values for the 3rd layer. This layer can learn by itself a bit more independently and it speeds up learning for the whole network.


Why this is a problem with neural networks?
x_1
x_2
x_3
$\hat{y}$
mean 0
variance 1
$\beta^{[2]}, \gamma^{[2]}$

- A second intuition is that back norm has a small regularization effect. Professor Ng does not recommend we use batch norm for regularization. We should only use it to speed up learning for a neural network by normalizing the hidden units' activations.



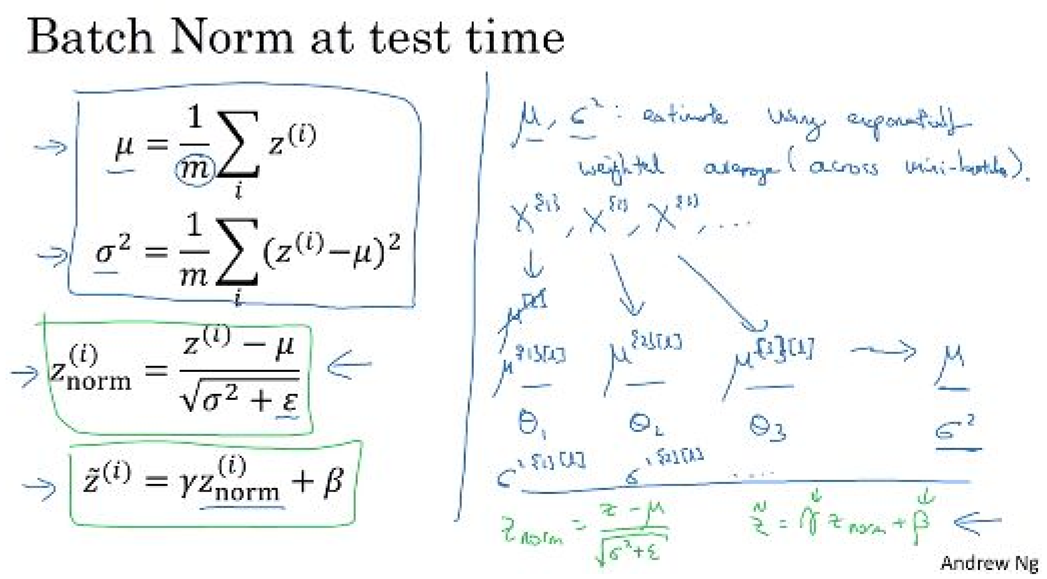
Batch Norm as regularization

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

$X^{\{t\}}$
$\tilde{z}^{[l]}$
64, 128
$z^{[l]}$
μ, σ^2
mini-batch : 64
512

Batch Norm at test time

- Batch norm processes data one mini batch are a time but at test time we might need to process data one example at a time so what do we do at test time then? We need a different way of computing the mean and the sigma. What we do is we use an exponentially weighted average across mini batches. How do we compute it? We start by computing the mean for the first mini batch, $\mu^{\{1\}[l]}$, then the mean for the second mini batch, $\mu^{\{2\}[l]}$, and so on for all mini batches and we end up with a vector of means. We use an exponentially weighted average on that vector and also on the σ^2 vector that has the σ^2 for each mini batch. We then have a running average for both mean and standard deviation. Our z normalized equation then uses the weighted averages we have for mean and sigma squared and then we use that z normalized to compute z tilde as follows: $\tilde{z} = \gamma * z_{norm} + \beta$



Batch Norm at test time

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

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

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

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

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

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

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

$\theta_1, \theta_2, \theta_3$

$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \ldots \qquad \sigma^2$

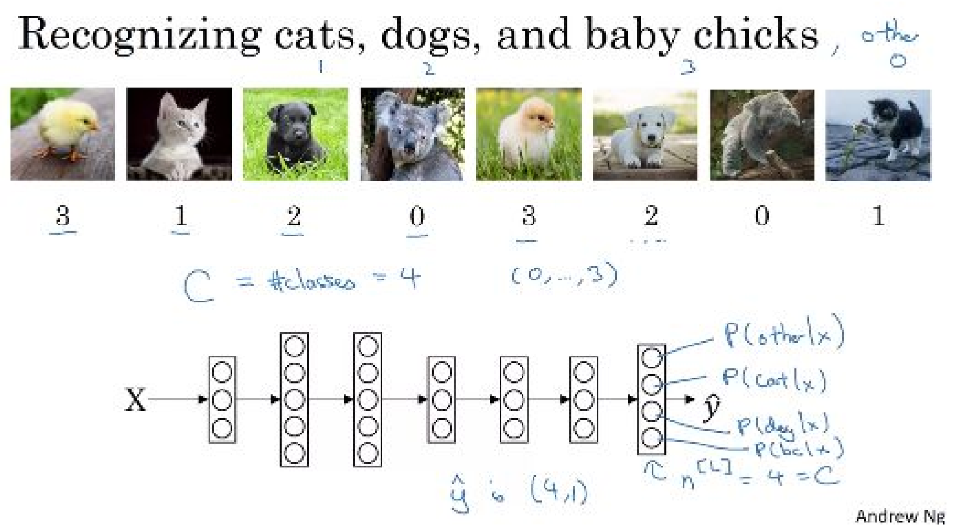
$z_{\text{norm}} = \frac{z-\mu}{\sqrt{\sigma^2+\varepsilon}} \qquad \tilde{z} = \gamma z_{\text{norm}} + \beta$

Andrew Ng

The intuition then is that at training team we compute mean and sigma squared using one mini batch are a time but at test time we might need to process one mini batch at a time. We then keep a running average of the mean and sigma squared we are seeing and those values are the ones we use to compute z_{norm} and $\tilde{z}$.

Multiclass classification: Softmax Regression

- Let’s take a look at Softmax regression (a generalization of logistic regression) which helps us to solve problems where we have to identify multiple classes, not just the binary ones we have seen until now.



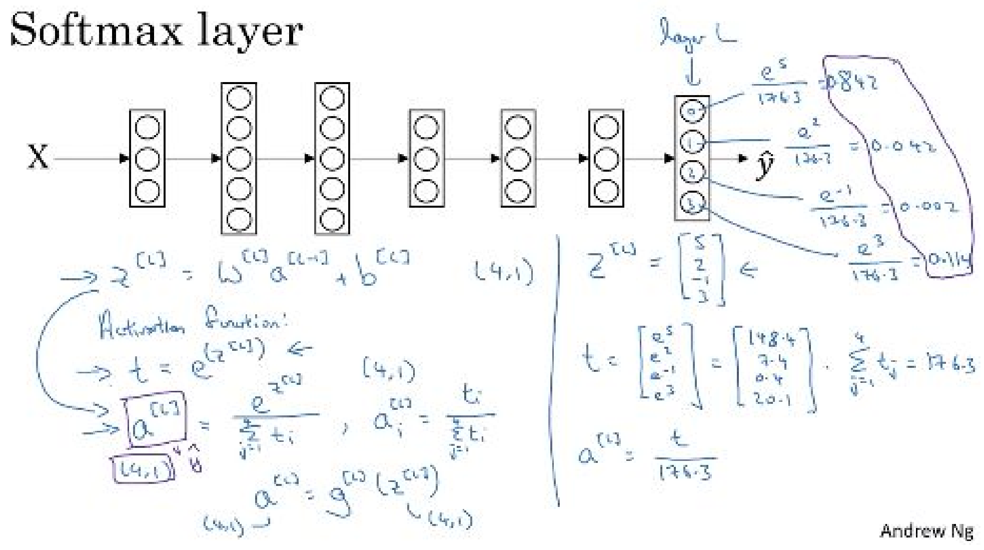
In the example above we will output four probabilities (if the image presented is a chick, a cat, a dog or other). The neural network to accomplish that will have an upper layer L with 4 outputs so $n^{[l]} = 4 = C$ and the probabilities for each unit in that layer are: $P(other \mid x),\ P(cat \mid x),\ P(dog \mid x)\ and\ P(bc \mid x)$ .yhat is then a 4x1 vector

- The standard way to do this is to use a softmax layer. The softmax layer is different from what we have seen before because it takes a vector as an input and outputs a vector of the same dimensions. Let's take a look at the math: we compute $z^{[l]} = w^{[l]} * a^{[l-1]} + b^{[l]}$.

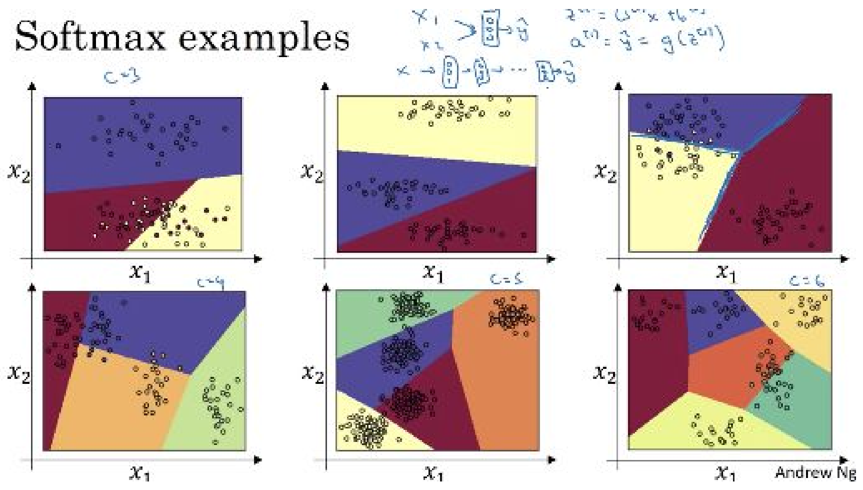
  For the activation function we compute a temporary variable $t = e^{(z^{[l]})}$ and $a^{[l]} = e^{(z^{[l]})} / \sum_{j=1}^{C} t_i$.

  Let's understand this using an example. Suppose $z^{[l]} = [5\ 2\ -1\ 3]$, then $t = [e^5\ e^2\ e^{-1}\ e^3] = [148.4\ 7.4\ 0.4\ 20.1]$ and $\sum_{j=1}^{4} t_i = 176.3$ and the activation $a^{[L]} = t/176.3$.

  Therefore, our output for each layer is then
  $e^5/176.3,\ e^2/176.3,\ e^{-1}/176.3\ \&\ e^3/176.3 = 0.842,\ 0.042,\ 0.002,\ 0.114.$

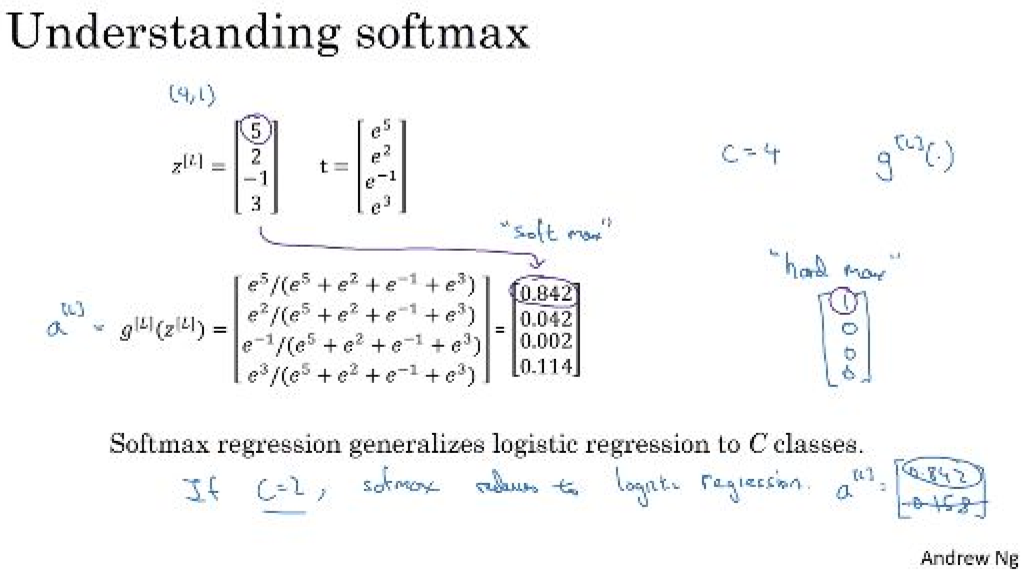


- What are some things that Softmax can represent? Let's look at some examples below. The intuition is that the decision boundary between any two classes will be linear.



Multiclass classification: Training a Softmax Classifier

- Below is our example from the previous session. We can see how we map from a $z^{[L]} = [5\ 2\ -1\ 3]$ *to an activation function* $a^{[L]} = [0.842\ 0.042\ 0.002\ 0.114]$. This is called softmax because a hardmax function would be of the following manner: $[1\ 0\ 0\ 0]$. Softmax is gentler than hardmax. Finally, softmax generalizes logistic regression to C classes. If C=2 then softmax reduces to logistic regression.



Understanding softmax

(4,1)

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

"soft max" "hard max"

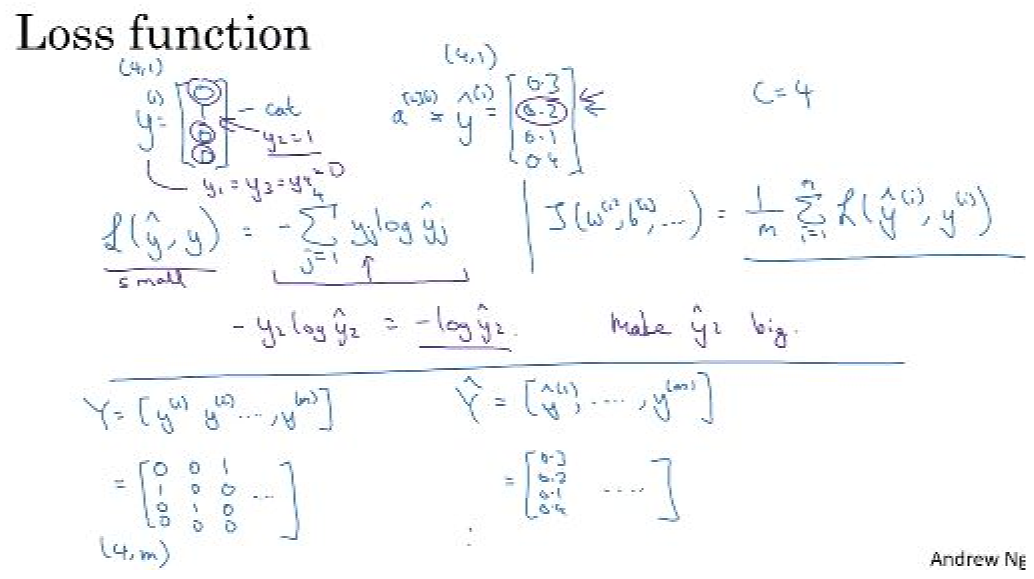
$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} = \begin{bmatrix} \boxed{0.842} \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \qquad \begin{bmatrix} \boxed{1} \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Softmax regression generalizes logistic regression to C classes.

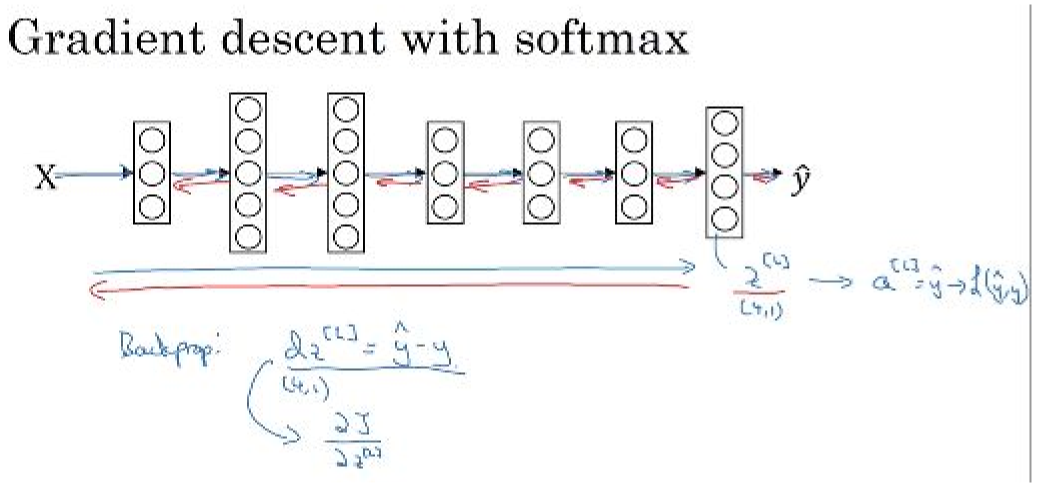
If C=2, softmax reduces to logistic regression. $a^{[L]} = \begin{bmatrix} \boxed{0.842} \\ \cancel{0.158} \end{bmatrix}$

Andrew Ng

- Now let's take a look at the loss function and cost function for softmax. Below we have $y^{[l]} = [0\ 1\ 0\ 0]$ and a corresponding $yhat = [0.3\ 0.2.\ 0.1.0.4]$. Our loss function is $L(yhat,\ y) = \sum_{j=1}^{4} y_j * log(yhat_j)$ and the cost function is $J(w^{[1]}, b^{[1]},\ ...) = (1/m)\ * \sum_{i=1}^{n} L(yhat,\ y)$

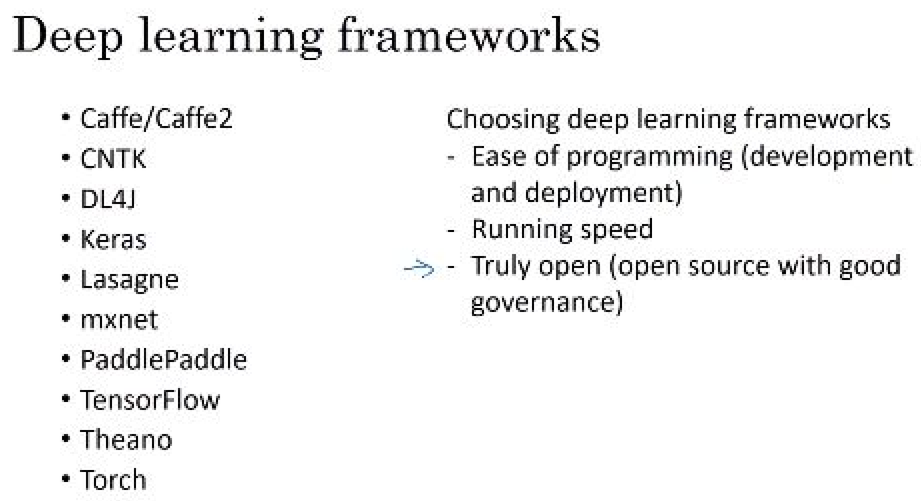

Loss function
(4,1)
$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ — cat
$y_2 = 1$
$y_1 = y_3 = y_4 = 0$
(4,1)
$a^{[L](i)} = \hat{y}^{(i)} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$
C=4
$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j$
small
$J(w^{[1]}, b^{[1]}, \dots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$
$-y_2 \log \hat{y}_2 = -\log \hat{y}_2$
Make $\hat{y}_2$ big.
$Y = [y^{(1)} \; y^{(2)} \dots y^{(m)}]$
$\hat{Y} = [\hat{y}^{(1)} \dots \hat{y}^{(m)}]$
$= \begin{bmatrix} 0 & 0 & 1 & \dots \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
$= \begin{bmatrix} 0.3 & \dots \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$
(4,m)
Andrew Ng

- Finally, let’s take a look at how we implement gradient descent with softmax. The key step within back prog is that the derivative $dz^{[L]} = yhat - y$. Deep Learning programming frameworks will compute the backprop step for us so we don’t have to worry about doing this from scratch.


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

Deep Learning Frameworks

- All the deep learning frameworks are credible choices. Instead of going over them we will focus on the recommended criteria for choosing a framework: ease of programming, running speed & truly open source (open source with good governance)



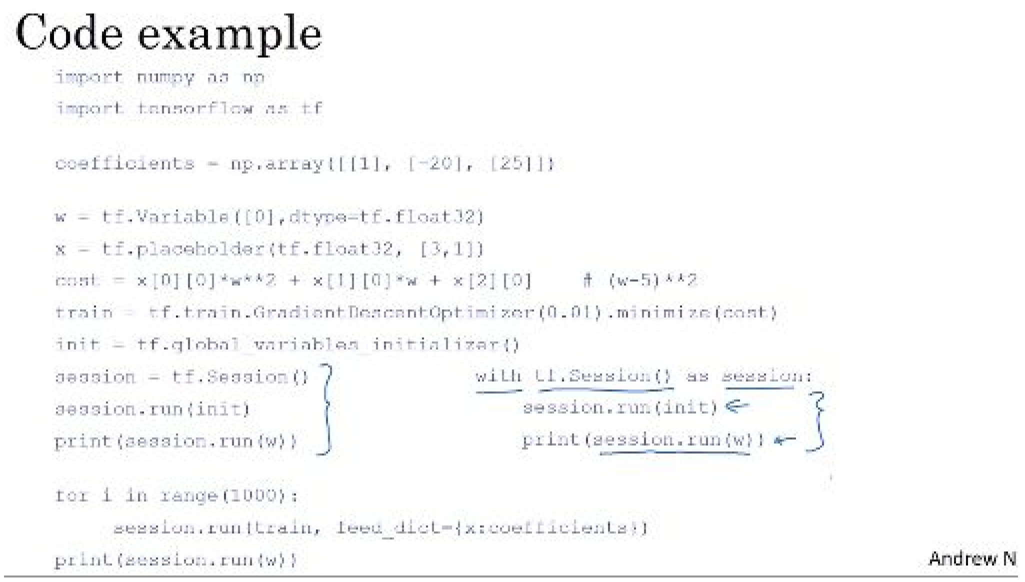
Deep learning frameworks

- 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)

Tensor Flow

Below is a sample TensorFlow program. The heart of a tensorflow program is the cost function; in the example below the cost function is followed by the optimization function (gradient descent in this case).



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

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

Andrew N