

Contents

1	1. hyperparameter tuning	1
1.1	1.1. tuning process	1
1.2	1.2. using an appropriate scale to pick hyperparameters	3
1.3	1.3. hyperparameters tuning in practice: Pandas vs. Caviar	5
2	2. batch normalization	5
2.1	2.1. normalizing activations in a network	5
2.2	2.2. fitting batch norm into a neural network	7
2.3	2.3. why does batch norm work?	9
2.4	2.4. batch norm at test time	9
3	3. multi-class classification	12
3.1	3.1. softmax regression	12
3.2	3.2. training a softmax classifier	13
4	4. introduction to programming frameworks	15
4.1	4.1. deep learning frameworks	15
4.2	4.2. tensorflow	15

contents

- 1. hyperparameter tuning
- 1.1. tuning process
- 1.2. using an appropriate scale to pick hyperparameters
- 1.3. hyperparameters tuning in practice: Pandas vs. Caviar
- 2. batch normalization
- 2.1. normalizing activations in a network
- 2.2. fitting batch norm into a neural network
- 2.3. why does batch norm work?
- 2.4. batch norm at test time
- 3. multi-class classification
- 3.1. softmax regression
- 3.2. training a softmax classifier
- 4. introduction to programming frameworks
- 4.1. deep learning frameworks
- 4.2. tensorflow

1 1. hyperparameter tuning

1.1 1.1. tuning process

如图，各超参的优先级为红 > 黄 > 紫，而 adam 的三个参数基本都不调整。

当参数比较少时，可以用表格，两两组合去尝试，但如果超参很多，可以用随机组合，因为事先并不知道各个参数的重新程度。例如，用一个很重要的参数 α 和一个没那么重要的参数 ϵ 组合，可能会发现试完 25 种组合，不论 ϵ 取什么值，效果都基本只和 α 的取值有关。而如果用随机组合的方法，同样的 25 个模型，可以试 25 种 α 的取值。

coarse to fine(由粗到精) 原则：当发现某个空间里的超参组合表现比较好时，例如图中蓝圈的那几个点，那么，可以在一个小区域（图中的蓝色正方形）内 sample 更多的参数组合。

Hyperparameters

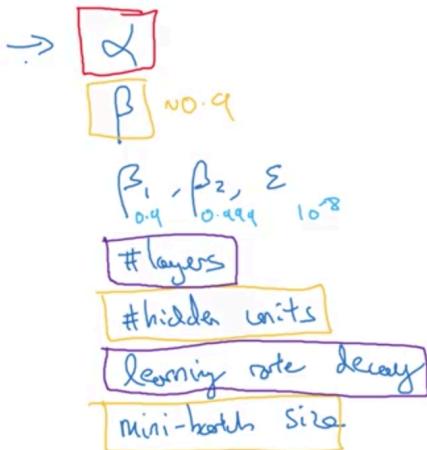


Figure 1: hyperparameters.png

Try random values: Don't use a grid

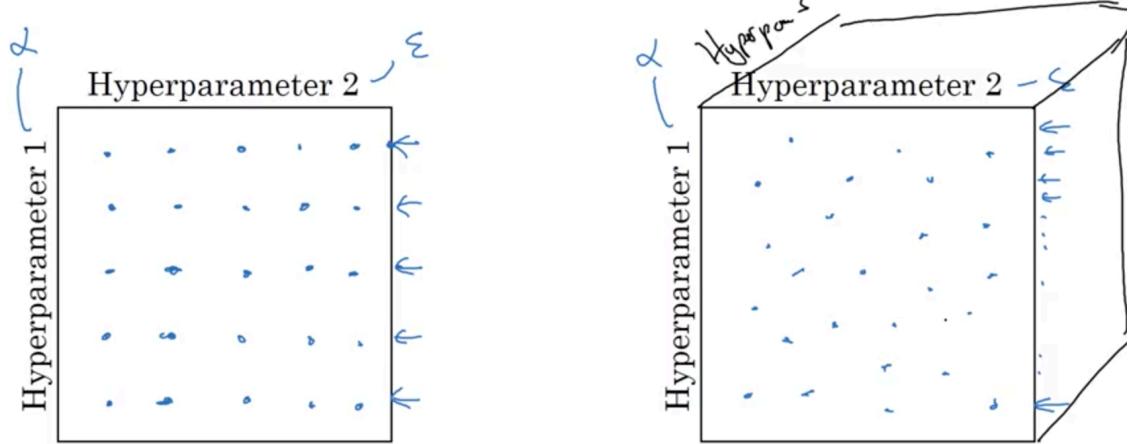


Figure 2: tuning-process-try-random-values.png

Coarse to fine

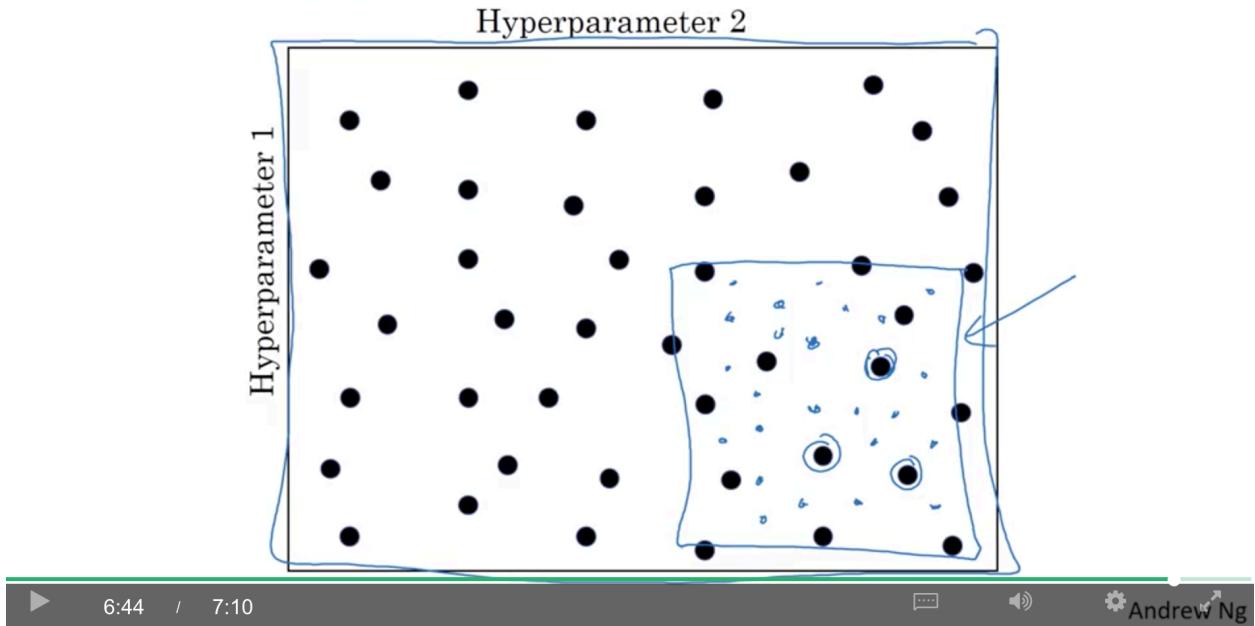


Figure 3: tuning-process-coarse-to-fine.png

1.2 1.2. using an appropriate scale to pick hyperparameters

一种方法是根据均匀分布 (uniform distribution) 来随机地选参数值：

但，如果 α 从 0.0001 到 1 之间均匀地随机采样，会有 90% 的值落在 0.1-1 之间，而 0.001 到 0.1 之间只有 10% 的值。相当于用了 90% 的资源搜索 0.1-1 的值，但只用 10% 的资源搜索 0.0001-0.1 的值。

更合理的方法是，在 log scale 上进行搜索，如图中的第二个数轴，每个区域的起点和终点是 10 倍的关系。

起点是 10^a ，终点是 10^b ，而 $0.0001 = 10^{-4}$ ，所以， $a = \log_{10} 0.0001 = -4$ 。所以

```
r = -4 * np.random.rand() # -4 <= r <= 0
alpha = 10 ** r # 10^-4 <= alpha <= 1
```

而对于 exponentially weight average 的超参 β ，取值范围从 0.9(相当于 10 天的平均)，到 0.999 (相当于 1000 天的平均)。这种情况，就做一个变换，对 $(1 - \beta)$ 在 log scale 上进行采样就行了。即

$$\beta = 1 - 10^r$$

如果想在 0.99 到 0.9 之间采样 β ，则 $r \in [-2, -1]$ 可以用：

```
r = np.random.rand() * (-1) - 1
beta = 1-10 ** r
```

或者

```
r = np.random.rand()
beta = 1-10**(-r - 1)
```

特别地，对于 exponentially weight average 而言，不能使用简单的均匀采样，要用 log scale 的采样。因为，当 β 的取值趋近于 1 时， β 的微小改变，会导致结果特别敏感。取 β 时，相当于 $1/(1 - \beta)$ 天的平均：+ 当 β 比较小时 (如 0.9000 变成 0.9005)，这个变化就是从 10.0 天到 10.05 的变化 + 而 β 比较大时 (如 0.9990 变成 0.9999)，那这个变化就是从 $1/(1-0.999)=1000$ 天变到 $1/(1-0.9995)=2000$ 天了！

Picking hyperparameters at random

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



$$\rightarrow \# \text{layers } L : 2 - 4$$

2, 3, 4



Figure 4: pick-hyperparameters-uniformly.png

Appropriate scale for hyperparameters

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

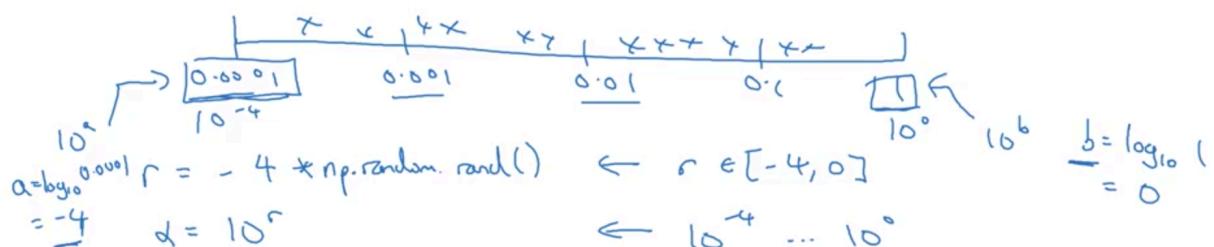
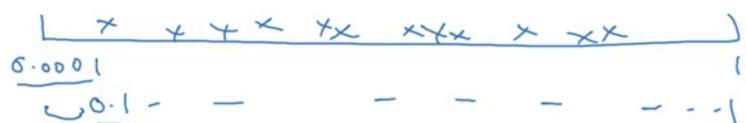


Figure 5: pick-hyperparameters-log-scale.png

Hyperparameters for exponentially weighted averages

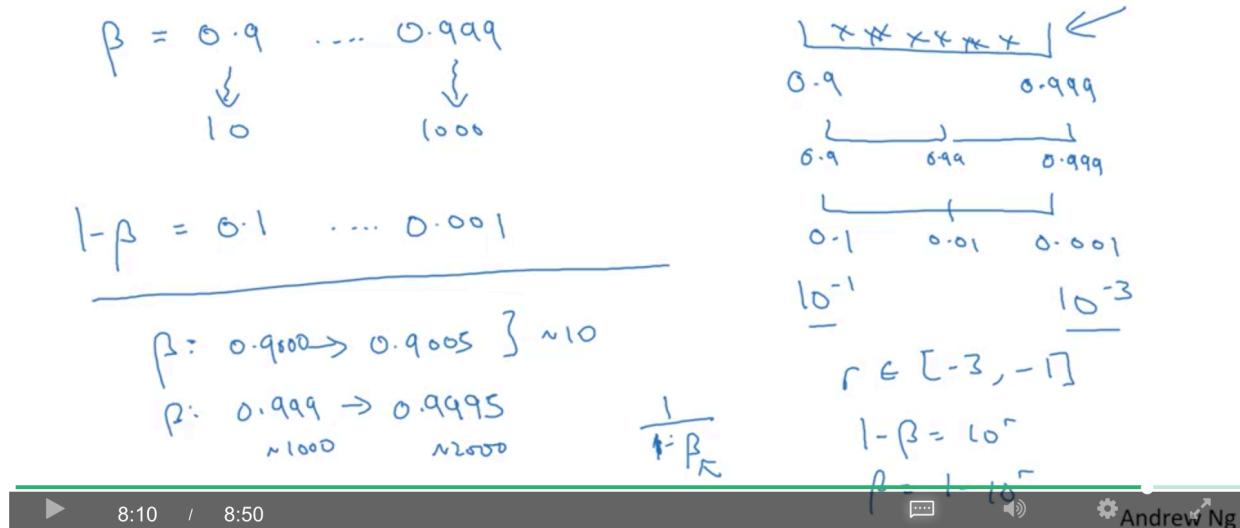


Figure 6: pick-hyperparameters-exponentially-weighted-average.png

1.3 1.3. hyperparameters tuning in practice: Pandas vs. Caviar

深度学习可以用在不同领域，而各领域的经验其实可以相互借鉴，所以 intuitions do get stale(直觉会变得陈腐，需要与时俱进)，所以至少每个月都最好调整下超参，保证总是可以得到最优解。

有以下主要两种流派：

- 精心照料一个模型：当没有足够的资源同时训练多个模型时，会对一个模型天天去微调参数或者发现出问题的话，回滚到上一个 checkpoint，重新调参 (panda approach, 熊猫产子少...)
- 并行训练多个模型：尝试不同的超参组合，选择一个最好的模型 (caviar strategy, 鱼子酱，鱼产子多...)

2 2. batch normalization

2.1 2.1. normalizing activations in a network

normalizing input values 在前面提到过 (c2w1 的 3.1 部分)，对 lr 很有效。

如果是多层神经网络，例如图中，想把 $W^{[3]}, b^{[3]}$ 训练得更好，应该也对 $a^{[2]}$ 进行 normalize。事实上，batch norm 就做的这件事，只是区别在于，bn 的 normalize 针对的是 $z^{[2]}$ ，而非 $a^{[2]}$ 。在学术界上，对 $a^{[2]}$ 还是 $z^{[2]}$ 进行 normalize 是有争议的，但对 $z^{[2]}$ 的 normalize 比较普遍。

给定一系列的中间值 $z^{(1)}, z^{(2)}, \dots, z^{(m)}$ ，这里 $z^{(i)}$ 是 $z^{[l]}(i)$ 的简写。

经过如下变换，得到 0 均值，1 方差的 $z_{norm}^{(i)}$ ：

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

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

Re-test hyperparameters occasionally

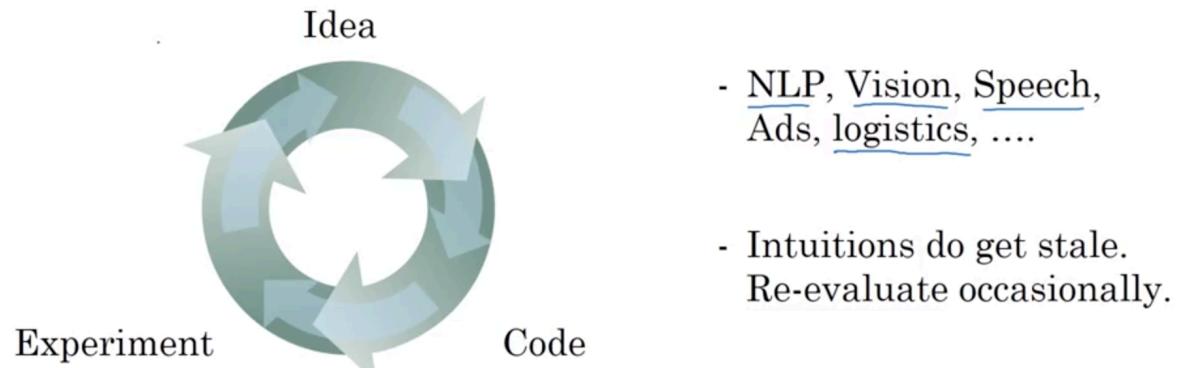
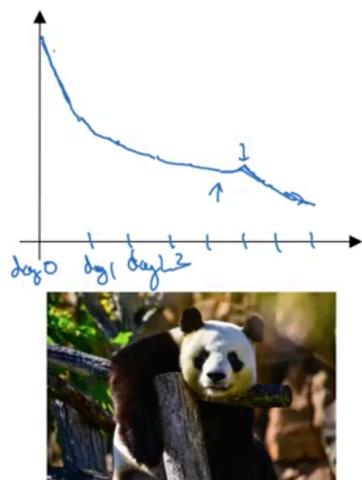


Figure 7: retest-hyperparameters-occasionally.png

Babysitting one model



Training many models in parallel

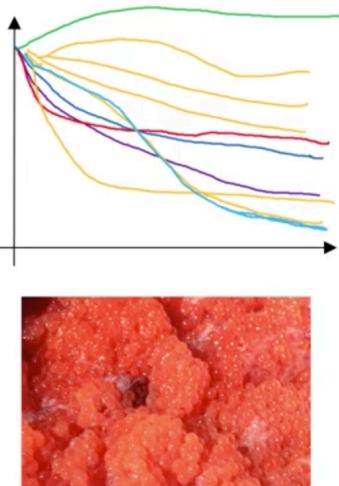


Figure 8: pick-hyperparameters-two-approaches.png

Normalizing inputs to speed up learning

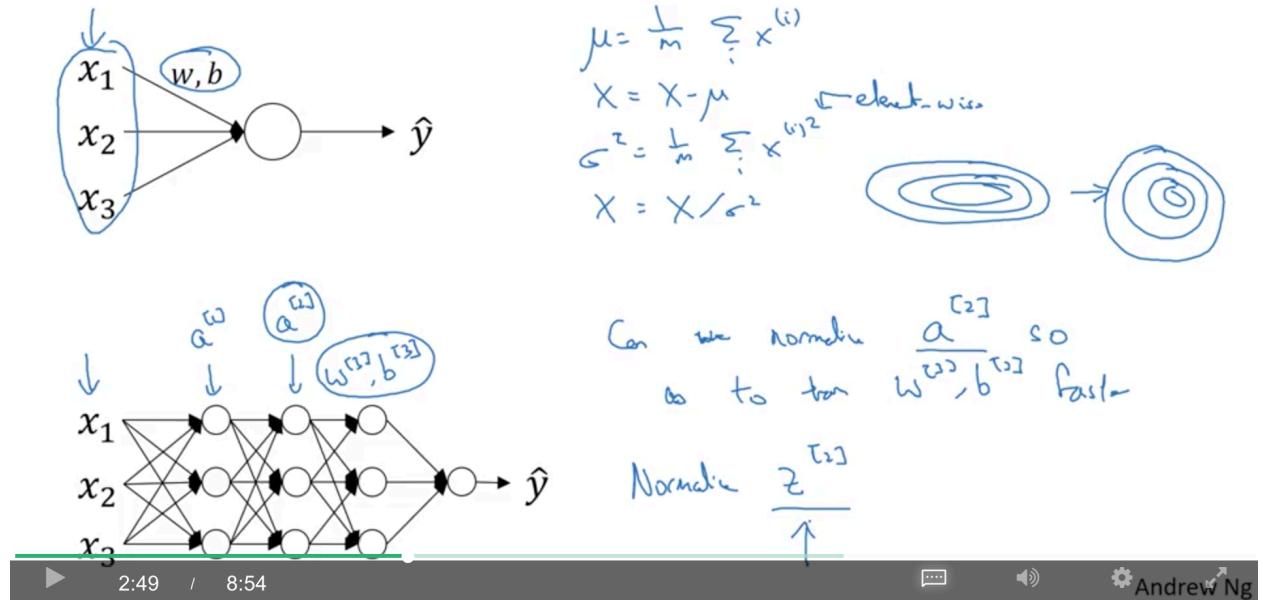


Figure 9: bn-problems.png

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

注意，其中的 ϵ 是为了防止分母为 0。而我们并不希望所有 $z^{(i)}$ 都变成 0 均值 1 方差的结果，也许它们本身的分布就不同 [如图中，假设激活函数是 sigmoid，并不希望数值是 0 均值 1 方差的，因为需要值很大或者很小，这样才能充分利用 sigmoid 的非线性]，所以，可以做如下处理：

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

而其中的 γ 和 β 是像 W 和 b 一样的可以学习的参数

如果

$$\gamma = \sqrt{\sigma^2 + \epsilon}$$

$$\beta = \mu$$

那么， $\tilde{z}^{(i)} = z^{(i)}$

所以，不同层使用不同的 γ 和 β ，可以使各层的分布不一样。最终使用 $\tilde{z}^{[l](i)}$ 代替 $z^{[l](i)}$

2.2 2.2. fitting batch norm into a neural network

针对 mini-batch 的 bn 如下，一次使用一个 mini-batch 的均值和方差去计算 $\tilde{z}^{[l]\{t\}}$ 。

注意：

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

但计算 $\tilde{z}^{[l]}$ 时， $b^{[l]}$ 是个常数，所以所有 mini-batch 里的样本，加不加这个常数，最终生成的 $z_{norm}^{[l]}$ 和 $\tilde{z}^{[l]}$ 都是一样的（方差不变，均值会变但 $-\mu$ 后就没变了，最终的均值和 bias 是 $\beta^{[l]}$ 决定的），所以，其实 $b^{[l]}$ 这个参数可以不用保留和计算，或者，可以直接将 $b^{[l]}$ 永远置为 0。

shape($n^{[l]}$ 是第 l 层的隐层单元数): $+ z^{[l]}: (n^{[l]}, 1) + \beta^{[l]}: (n^{[l]}, 1) + b^{[l]}: (n^{[l]}, 1)$ (可以不要) $+ \gamma^{[l]}: (n^{[l]}, 1)$

Implementing Batch Norm

Given some intermediate values in NN

$$\begin{cases} \mu = \frac{1}{m} \sum z^{(i)} \\ \sigma^2 = \frac{1}{m} \sum (z^{(i)} - \mu)^2 \\ z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ z^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta \end{cases}$$

If $\gamma = \sqrt{\sigma^2 + \epsilon}$ ←
 $\beta = \mu$ ←
then $\hat{z}^{(i)} = z^{(i)}$

x ←
 $\bar{z}^{(i)}$ ←

learnable parameters of model.

Use $\hat{z}^{(i)}$ instl of $z^{(i)}$.



Figure 10: bn-implement.png

Adding Batch Norm to a network

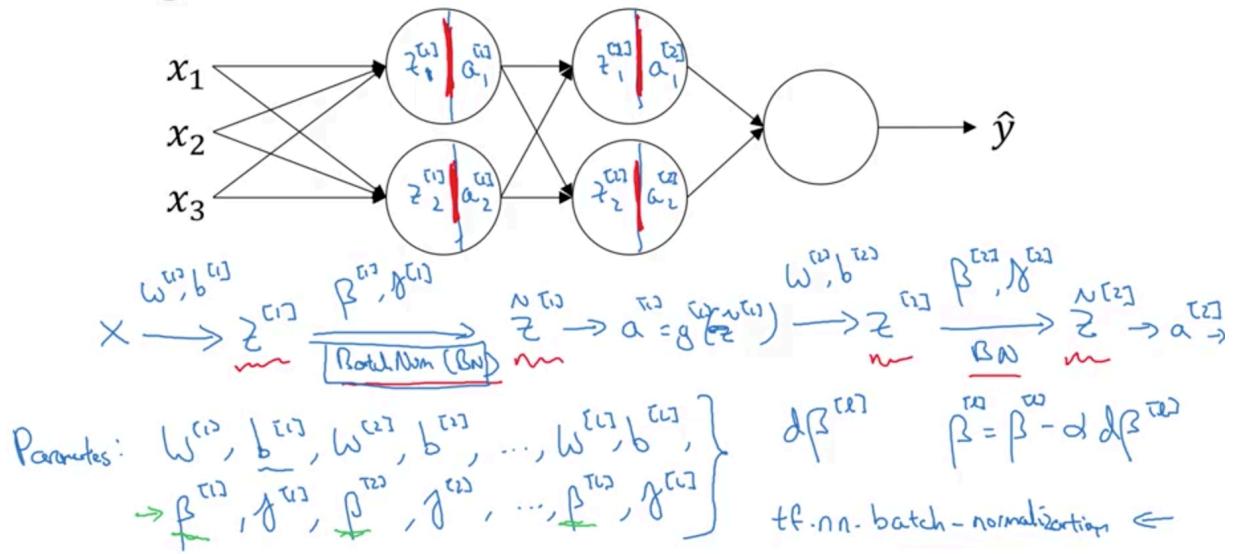


Figure 11: bn-adding-bn-to-a-nn.png

Working with mini-batches

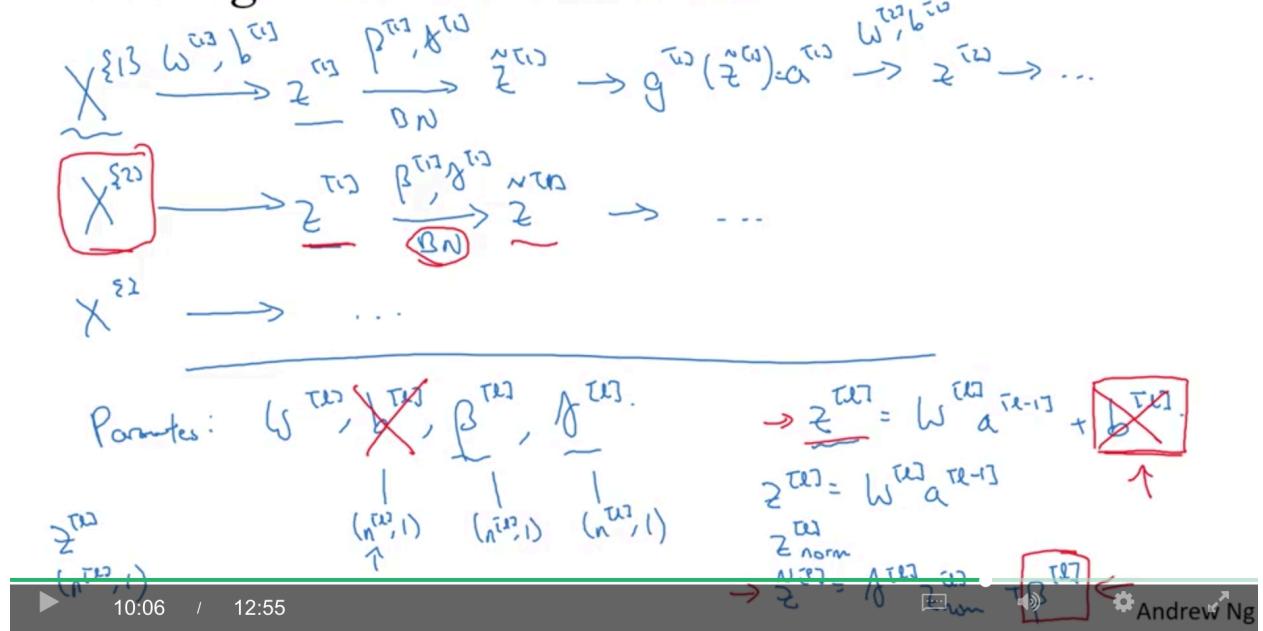


Figure 12: bn-mini-batch-bn-no-bias.png

整体实现流程如下：

2.3 2.3. why does batch norm work?

高层的权重对低层权重的变化更有鲁棒性。例如，训练集里的猫都是黑猫，但希望对其他颜色的猫仍然能很好地识别，即使可能二者用的函数是类似的，决策边界也类似（如图中的绿线），但实际两个的输入分布是不一样的：**covariate(相关变量, 协变量) shift**。

所以，这种情况下，即使模型可能不需要改变，但最好可以重新训练模型。

在神经网络中，例如针对第三层的 $W^{[3]}$ 和 $b^{[3]}$ 而言，输入的是 $a_1^{[2]}, \dots, a_4^{[2]}$ ，然后还要学习 $W^{[4]}$ 和 $b^{[4]}$ ，从而学习到 \hat{y} ；而 $a_1^{[2]}, \dots, a_4^{[2]}$ 是通过 $W^{[2]}, b^{[2]}$ 以及 $W^{[1]}, b^{[1]}$ 生成的，所以输入分布的改变，会引起这些的改变，也会影响到 $W^{[3]}$ 和 $b^{[3]}$ 。

而 batch-norm 可以保证每一层的 $z^{[l]}$ 的均值和方差受 $\beta^{[l]}$ 和 $\gamma^{[l]}$ 控制，从而对实际的输入的变化更加鲁棒。相当于减小了前一层的与当前层的参数之前的耦合，所以它允许网络的每一层独立学习，因此可以加速整个网络的训练。

bn 的第二个效果，就是它具有轻微的正则化效果。

- 每个 mini-batch 的取值是依据当前 mini-batch 的均值/方差进行归一化的
- 所以从 $z^{[l]}$ 到 $\tilde{z}^{[l]}$ ，是增加了当前 mini-batch 的带有噪声（不同于全局的均值和方差）的均值和方差的，而且前面提到了还会减去均值，这样又引入了另一个噪声。类似 dropout，会对每层的激活值增加一些噪声（对激活值以一定概率 *0，以另一个概率 *1）。
- 因为加入的噪声比较小，所以正则化效果并没有非常强大，只是轻微的效果。
- 对 dropout 而言，通过增加 mini-batch size，例如从 64 变成 512，可以减小噪声，从而减小正则化的效果。

所以不要把 bn 看成一种正则化方法，用它来归一化隐层单元的激活函数以加速学习就行了，正则化只是一种无意的副作用。

2.4 2.4. batch norm at test time

训练时，一个 mini-batch 中有 m 个元素的均值和方差，但测试时，只有一个元素，用它的均值和方差并不 make sense。

测试时，用 exponentially weighted average across the mini-batches 来估计（即，对所有 mini-batches，算出一个 μ 和 σ^2 ）。

Implementing gradient descent

for $t = 1 \dots \text{numMinibatches}$
 Compute forward pass on $X^{(t)}$.
 In each hidden layer, use BN to replace $\underline{z}^{(t)}$ with $\hat{\underline{z}}^{(t)}$.
 Use backprop to compute $\underline{dW}^{(t)}, \underline{d\beta}^{(t)}, \underline{d\gamma}^{(t)}$
 Update parameters $\left. \begin{array}{l} W^{(t)} := W^{(t)} - \alpha dW^{(t)} \\ \beta^{(t)} := \beta^{(t)} - \alpha d\beta^{(t)} \\ \gamma^{(t)} := \dots \end{array} \right\} \leftarrow$

Works w/ momentum, RMSprop, Adam.



Figure 13: bn-mini-batch-gradient-descent-implementation.png

Learning on shifting input distribution

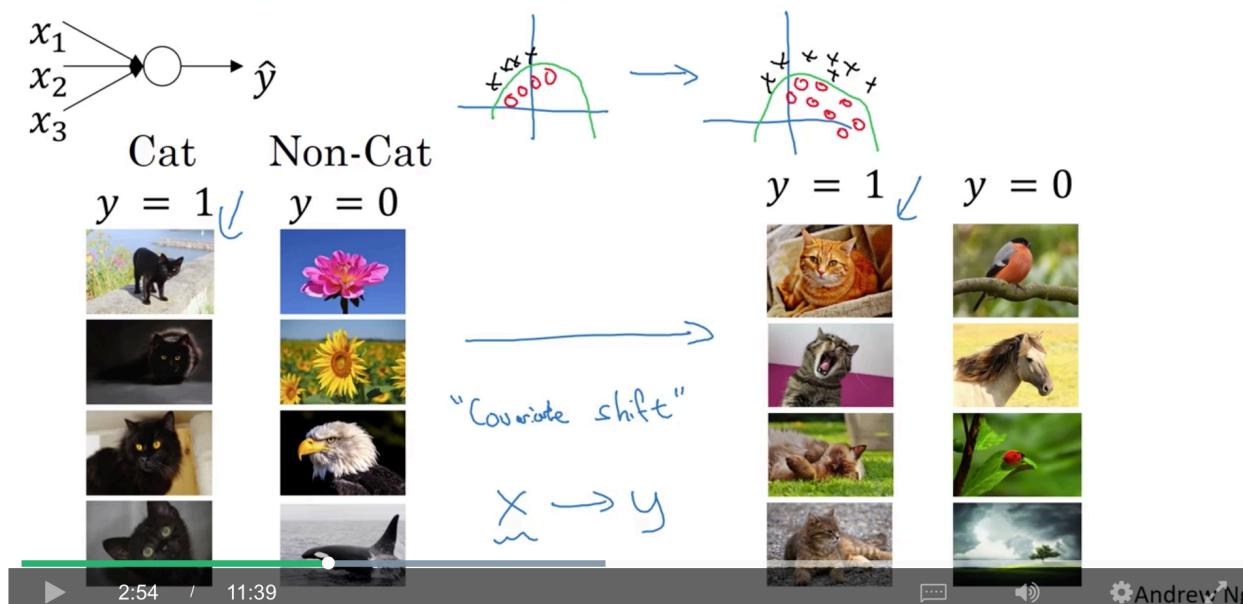


Figure 14: why-bn-works-shifting-input-distribution.png

Why this is a problem with neural networks?

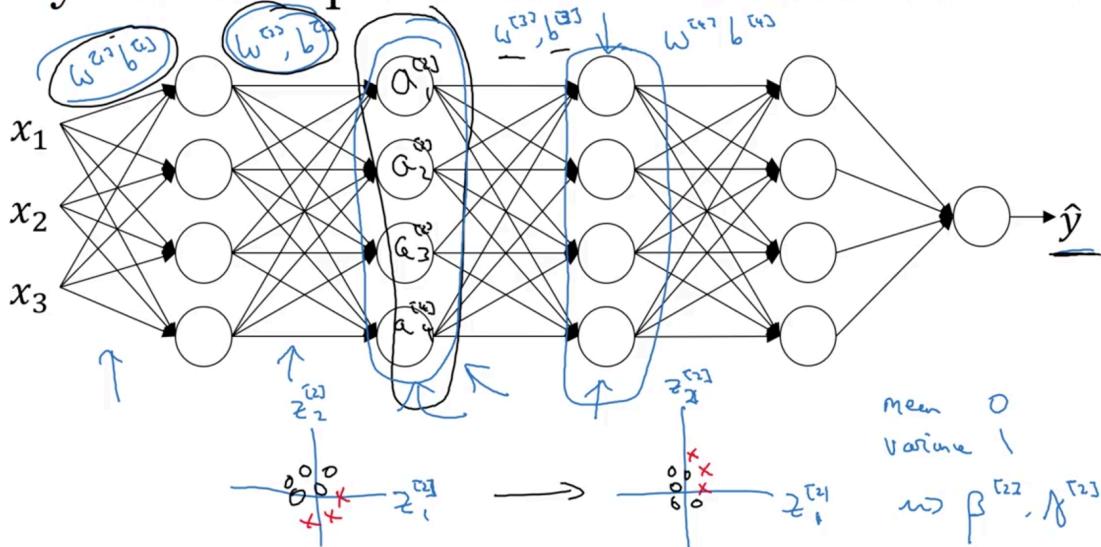


Figure 15: why-bn-works-effect1.png

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.

mini-batch : 64 512

Figure 16: why-bn-works-effect2.png

对于第 l 层的不同的 mini-batch, $x^{(1)}, x^{(2)}, \dots$ 而言, 相应的均值为 $\mu^{(1)[l]}, \mu^{(2)[l]}, \dots$, 即 $\theta_1, \theta_2, \dots$ 。同样地, 针对方差也可以使用 $\sigma^{(1)[l]}, \sigma^{(2)[l]}, \dots$

最后用估算出来的 μ 和 σ^2 来代替训练时的 μ 和 σ^2 计算 $z_{norm}^{(i)}$, 从而计算 $\tilde{z}^{(i)}$ 。

使用指数加权平均的效果一般是可以的, 而实际上任何合理的估算均值和方差的方法在测试时效果应该都是可以的。如果使用一些深度学习框架, 通常有一些默认的方法来估算均值和方差, 效果一般也比较好。

Batch Norm at test time

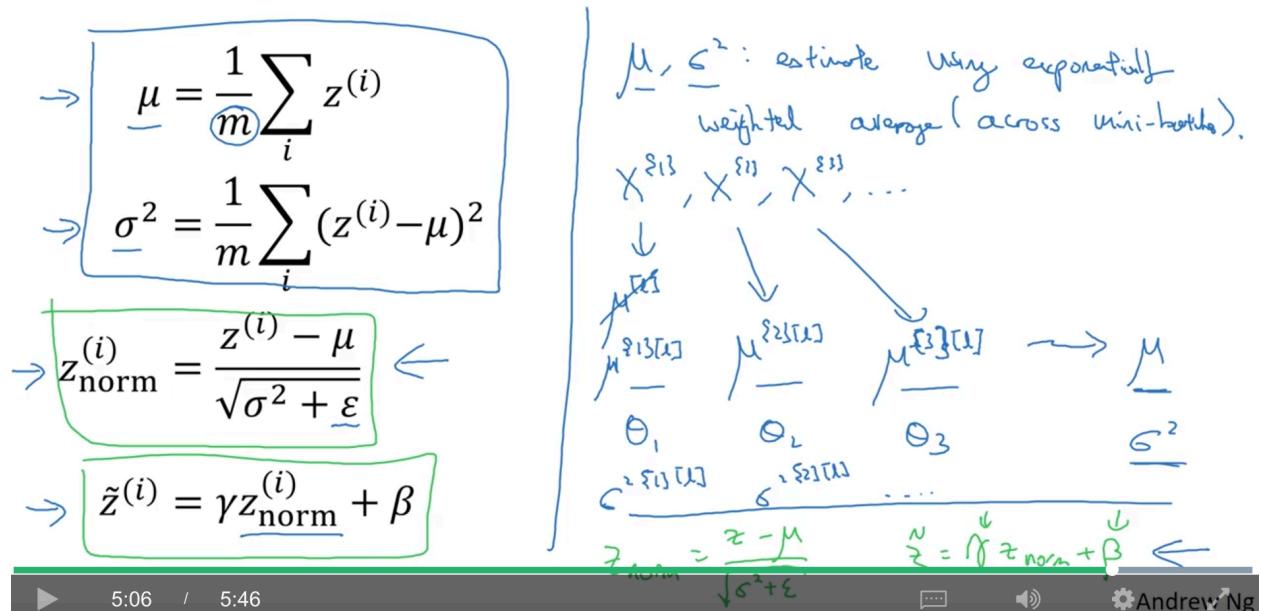


Figure 17: bn-test-time.png

3 3. multi-class classification

3.1 3.1. softmax regression

C 表示类别的数量, 输出层 \hat{y} 是一个 $(n^{[L]}, 1) = (C, 1)$ 的向量, 每个元素表示对应分类的概率, 并且所有分类的概率求和是 1。

对于最后一层 L :

$$\begin{aligned} z^{[L]} &= W^{[L]} a^{[L-1]} + b^{[L]} \\ t &= e^{(z^{[L]})} \\ \hat{y} &= a^{[L]} = g^{[L]}(z^{[L]}) = \frac{e^{(z^{[L]})}}{\sum_{j=1}^C t_j} \end{aligned}$$

对于 $a^{[L]}$ 的第 i 个元素而言, 就是:

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^C t_j}$$

- $z^{[L]}$ 的 shape 是 $(C, 1)$
- t 的 shape 是 $(C, 1)$

Recognizing cats, dogs, and baby chicks , other



3 1 2 0 3 2 0 1

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

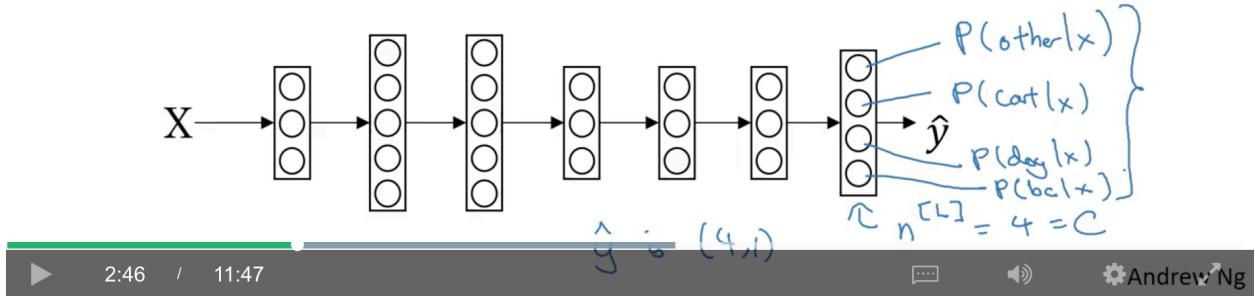


Figure 18: multi-class-classification.png

- $a^{[L]}$ 的 shape 是 $(C, 1)$
- \hat{y} 的 shape 是 $(C, 1)$

区别于二分类，二分类最后一层的输入和输出都是一个实数，而 softmax 都是一个 vector。

对于一个没有隐层的神经网络而言，对于一个多分类器而言，下面几张图的任意两类之间的 decision boundaries 都是线性的：

3.2 3.2. training a softmax classifier

hot max; 只保留最大的元素并置为 1，其他元素置为 0。而对于 softmax 而言，仍然有类似的性质， $z_i^{[L]}$ 最大的元素， $a_i^{[L]}$ 也最大，但其他元素并没有都变成 0，所以叫『soft』max。

另外，可以证明，softmax regression 是 logistic regression 的一般化，而 $C = 2$ 时，softmax regression 就是 logistic regression。loss function 如下：

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

例如，假设这是 $[0,1,0,0]$ 也就是类别 1，那么

$$L(\hat{y}, y) = -y_2 * \log \hat{y}_2 = -\log \hat{y}_2$$

所以，希望 $L(\hat{y}, y)$ 小，就是希望 \hat{y}_2 大。其实就是 maximum likelihood estimation 的一种形式。

对于整个训练集的 cost:

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

Softmax layer

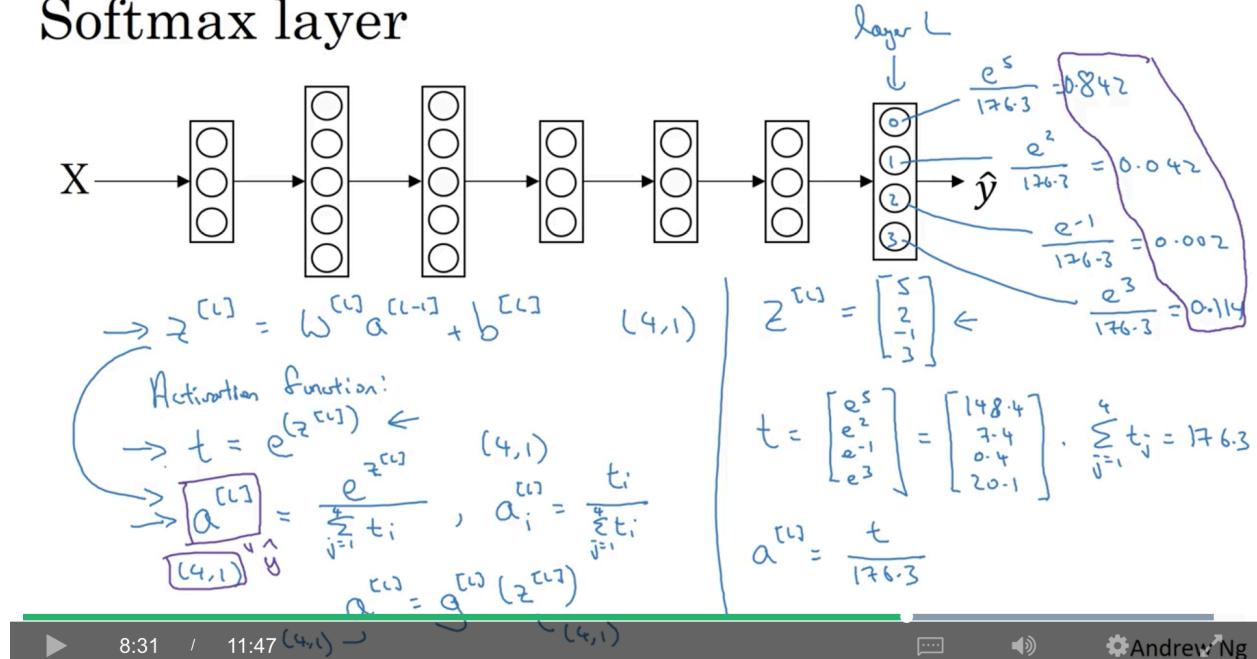


Figure 19: softmax-regression.png

Softmax examples

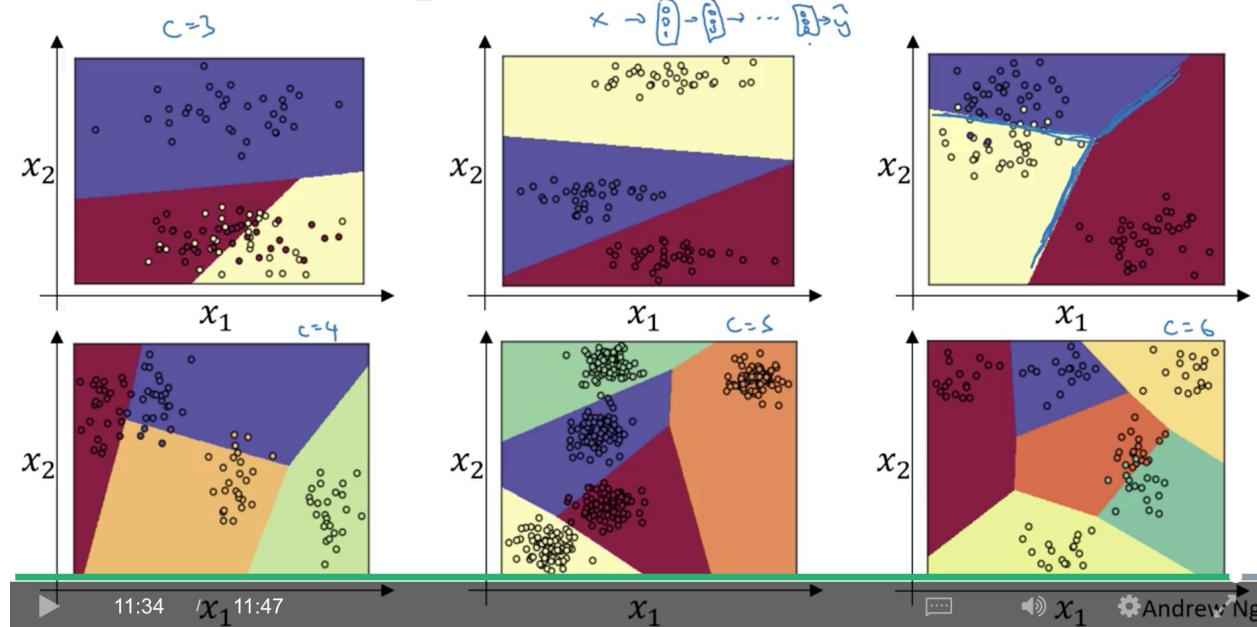


Figure 20: softmax-examples.png

Understanding softmax

(4,1)

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

$\underbrace{\qquad\qquad\qquad}_{\text{"Soft max"}}$

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

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

"hard max"

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

Softmax regression generalizes logistic regression to C classes.

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



Figure 21: understanding-softmax.png

使用 vectorized implementation 时, Y 的 shape 是 (C, m) , \hat{Y} 的 shape 也是 (C, m)

反向传播这里就不细说了, 现有的编程框架都能自动求导。

4 4. introduction to programming frameworks

4.1 4.1. deep learning frameworks

其中的 truly open 指的是, 软件开源, 并且不是被单一公司所掌控。例如某些公司可能开源了某个软件, 但却独占着领导权, 几年后, 当人们开始使用该软件时, 该公司可能会逐渐关闭开源, 或者将一些功能移入私有云服务中

4.2 4.2. tensorflow

例如, 实现求解使得 $cost = w^2 + 25 - 10w$, 即 $(w - 5)^2$, 取得最小值时 w 的取值:

```
import tensorflow as tf
import numpy as np

w = tf.Variable(0, dtype=tf.float32)
cost = tf.add(tf.add(w**2, tf.multiply(-10,w)), 25)

# 因为 tf 重载过了运算符, 所以也可以直接用下面的

cost = w**2 + 25 - 10 * w

train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
```

Loss function

Handwritten notes on softmax loss function:

$y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ y^{(4)} \end{pmatrix}$ - cat
 $y_1 = y_3 = y_4 = 0$
 $y_2 = 1$

$\hat{y} = \begin{pmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \hat{y}^{(3)} \\ \hat{y}^{(4)} \end{pmatrix}$

$a^{(i)} \approx \hat{y}^{(i)} = \begin{pmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{pmatrix}$

$C = 4$

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

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

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

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

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

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

$= \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \dots$

Progress bar: 7:33 / 10:07 Andrew Ng

Figure 22: softmax-loss-function.png

Gradient descent with softmax

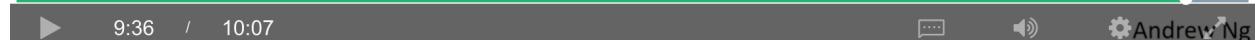
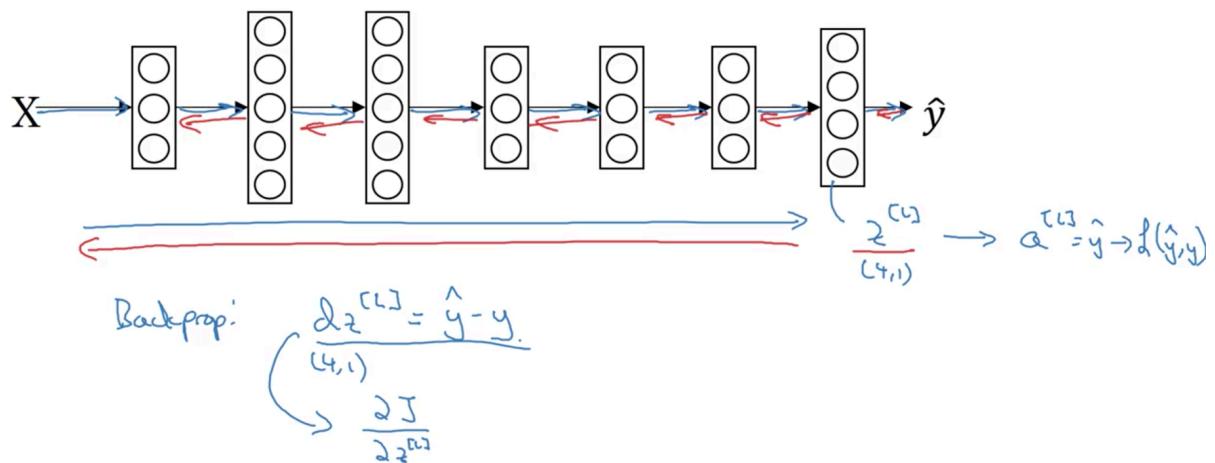


Figure 23: gradient-descent-softmax.png

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)



Figure 24: deeplearning-frameworks.png

```
init = tf.global_variables_initializer()

# 使用 with 的好处，在程序异常退出时，clean up 做得更好
with tf.Session() as session:
    session.run(init)
    for i in range(1000):
        session.run(train) # 每 run 一次 train，训练一轮
    print (session.run(w)) # 打印出此时的 w
```

上面的例子中，cost 函数是固定的（变量只有 w），而对于训练数据 x (特殊的 Variable，其值后续会 assign)：

```
x = tf.placeholder(tf.float32, [3,1]) # 一个 shape 是 (3,1) 的 array
```

而假设 cost function 如下：

```
cost = x[0][0] * w ** 2 + x[1][0] * w + x[2][0]
```

另外，定义 x 的数据来源（训练样本）：

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

然后通过 feed_dict 指定 x 的来源，然后 run：

```
for i in range(1000):
    session.run(train, feed_dict={x: coefficients}) # 每 run 一次 train，训练一轮

print (session.run(w)) # 打印出此时的 w
```

如果要实现 mini-batch gradient descent，每次传给 feed_dict 的变成每个 mini-batch 的数据即可。

注意：tf.multiply 是 element-wise 的乘法，而 tf.matmul 是矩阵乘法。

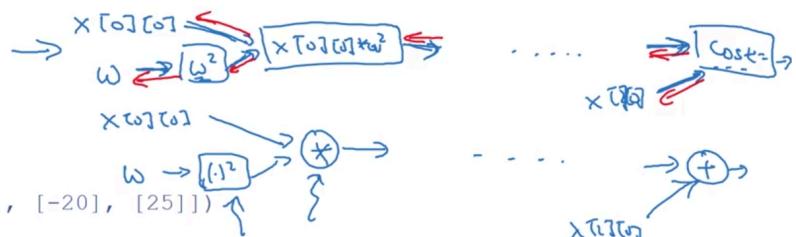
Code example

```

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))

```

▶ pr 15:18 sessi 16:07 run (w) ◀ Andrew Ng

Figure 25: tf-code-example.png