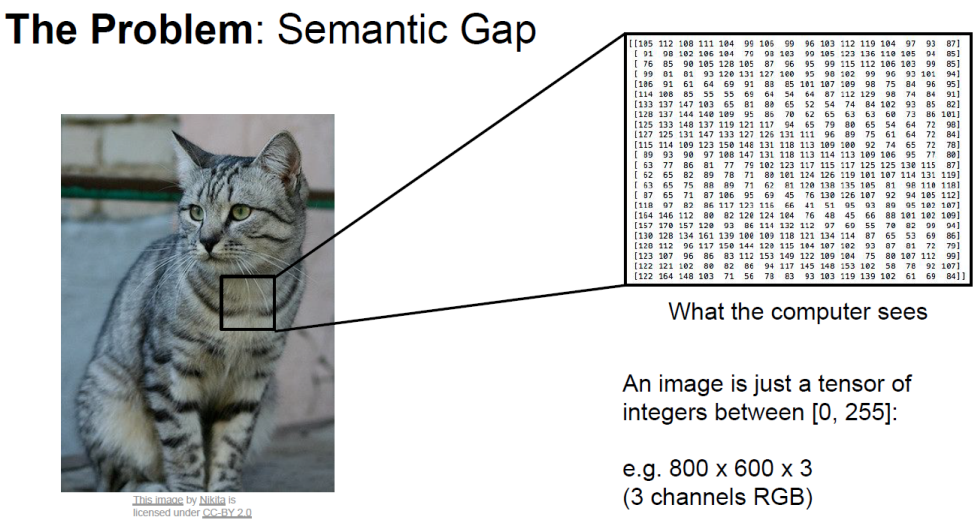
2.1 图像分类 - 数据驱动方法

数据驱动的方法即不使用特定的规则，而是用大量数据.



- 计算机看到的是数字，如果换一个角度，同样的猫但是图片数字可能完全不同，算法需要做到这样的鲁棒：光线，遮挡，猫变形，背景混乱，猫与猫之间的不同等.
- semantic gap:

  一幅图片所表达的意思和计算机所看到的之间的差距.

k - Nearest Neighbor

- train: 仅记录数据.
- test: 计算k个最近邻，投票分类.

Distance Metric to compare images

L1 distance: $d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$

test image					training image				pixel-wise absolute value differences				
56	32	10	18	-	10	20	24	17	=	46	12	14	1
90	23	128	133		8	10	89	100		82	13	39	33
24	26	178	200		12	16	178	170		12	10	0	30
2	0	255	220		4	32	233	112		2	32	22	108

add → 456

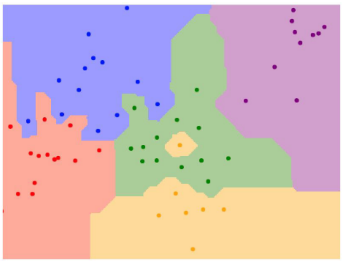
我们希望的是分类器在test的时候非常快，这样可以部署在一些普通的设备上，训练过程可以消耗很大.

不同的度量方式：

K-Nearest Neighbors: Distance Metric

L1 (Manhattan) distance

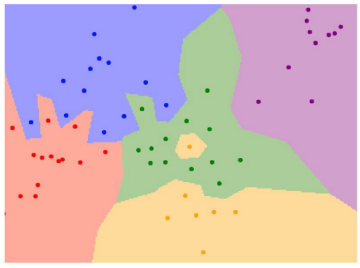
$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$



K = 1

L2 (Euclidean) distance

$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$



K = 1

为什么要引入验证集 / 如何找到好的超参数:

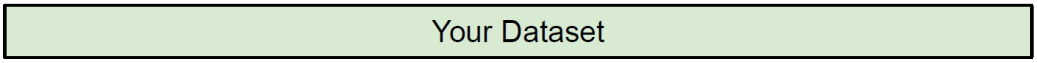
找到好的超参数，如下：

- Idea 1: 关心的是模型在未知的数据上表现得更好，即泛化能力，而不是模型拟合训练数据很好就好了.
- Idea 2: 在这组test set上的训练数据不能代表所有的，而且这样会让模型先见到test set.

Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the data

BAD: K = 1 always works perfectly on training data



Idea #2: Split data into **train** and **test**, choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data

Idea #3: Split data into **train**, **val**, and **test**; choose hyperparameters on val and evaluate on test

Better!

选择在validation set上表现最好的超参数，再到test set上预测并计算各种数据．这才是在未见数据上的预测能力．

一般采用k-fold cross validation.

kNN不用于图像处理的原因

- 之前是图像上 L_1, L_2 距离并用于kNN的例子，但是实际上效果很不好，因为如下all 3 images have same L2 distance to the one on the left:

k-Nearest Neighbor with pixel distance **never used.**

- Distance metrics on pixels are not informative
- Very slow at test time

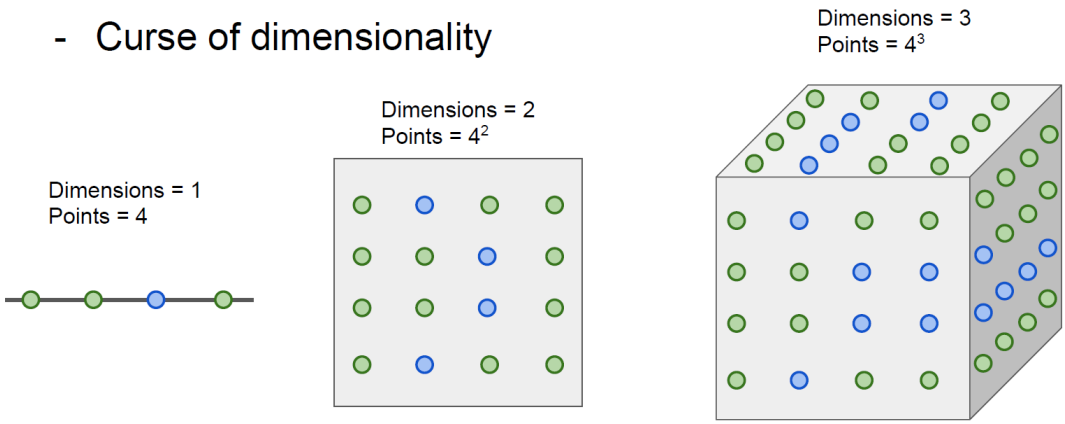
Original image is CC0 public domain

(all 3 images have same L2 distance to the one on the left)

- curse of dimensionality:

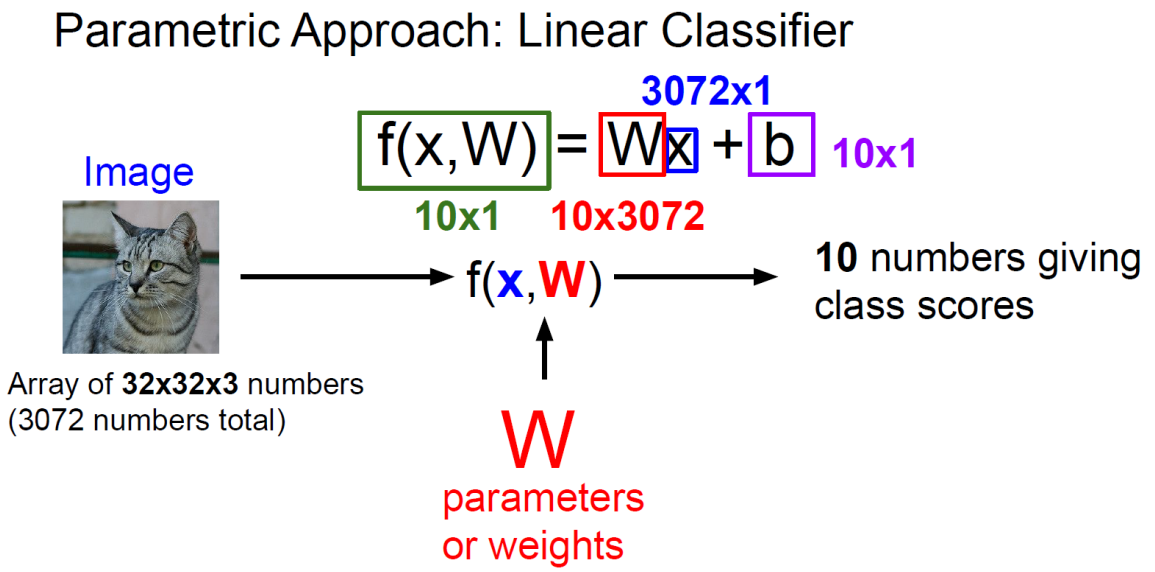
k-Nearest Neighbor with pixel distance **never used.**

- Curse of dimensionality



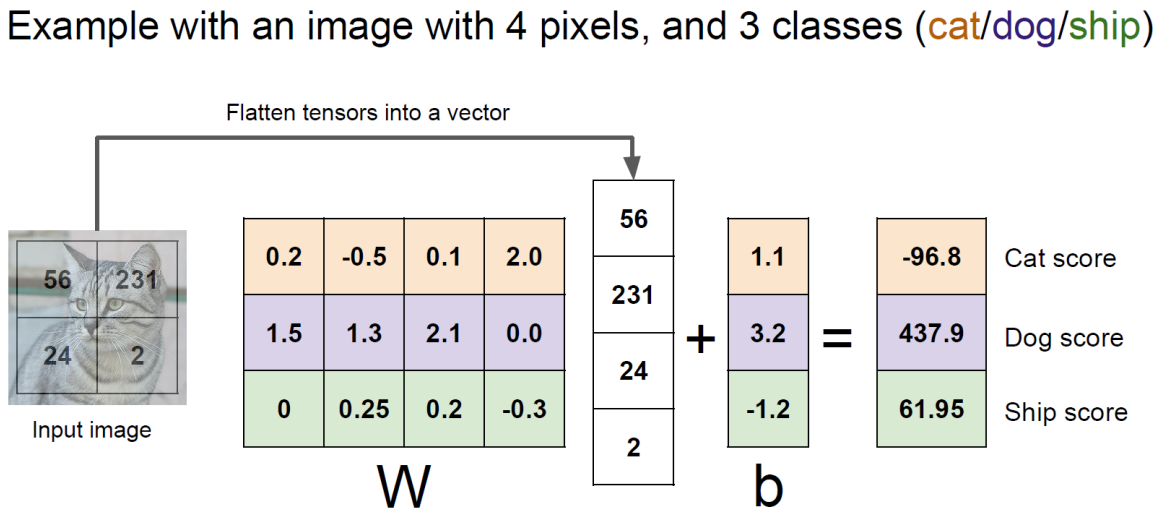
kNN的效果建立在训练数据可以密集地分布在空间中，图片维度高不可能收集到那么多数据．此外高维度带来的指数增长计算复杂度也是不能接受的．

Linear Classifier



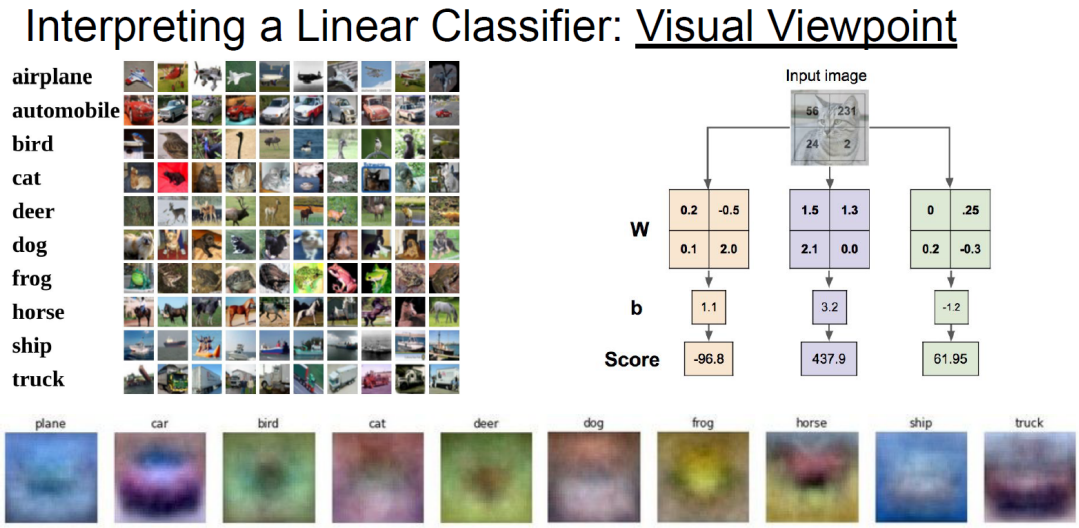
注意维度，图片是 `32 x 32 x 3` 的，然后展开成了 `3072` 的长向量．

例子：这里比较简单，W的每行就代表对于某一类分类的权重：

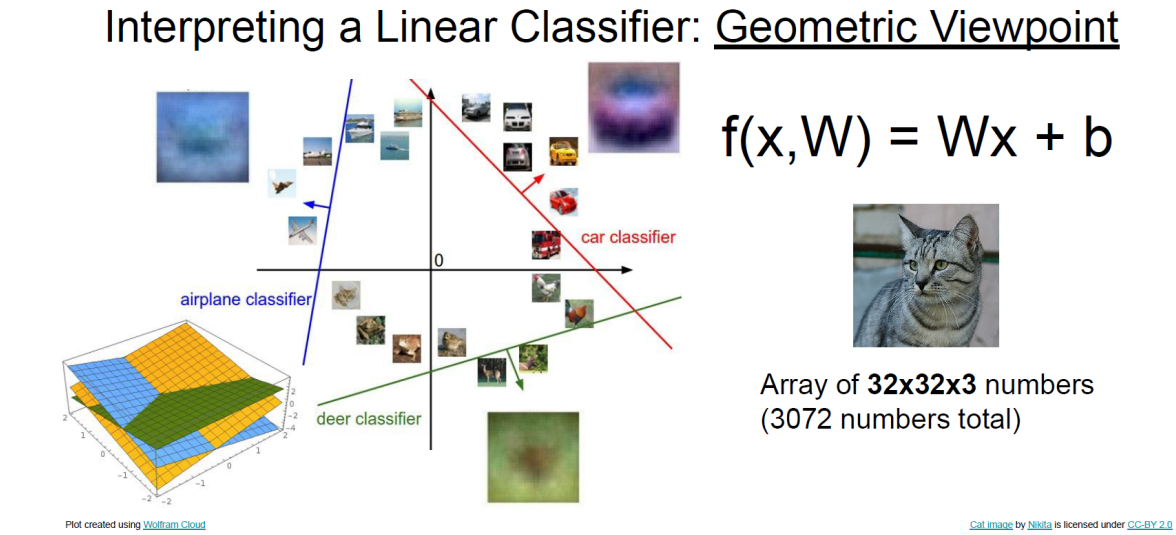


- 第一个角度 来解释线性分类器：W 的行向量 $\in \mathbb{R}^D$ 决定了input属于该类的权重/可能性，线性分类器(W的行向量) 有点像在学习这个类的模板，或者说综合了这个类的均值，行向量也

可以还原为一个图像：



- 另一个角度，线性分类器寻找的是决策面，我感觉这个角度的解释更好：



线性分类器无法处理线性不可分问题.

3.1 损失函数

如何优化 W，首先找到如何判定哪些 W 是好的. 引出loss function.

一些标记：

- $\{(x_i, y_i)\}_{i=1}^N$ 是给定的dataset，x_i 是image，y_i 是label.
- 整个数据集上的loss：$L = \frac{1}{N}\sum_i L_i\left(f\left(x_i, W\right), y_i\right)$ 为所有的平均.

multi-class SVM loss

$$
\begin{aligned}
L_i &= \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} \\
&= \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right)
\end{aligned}
$$

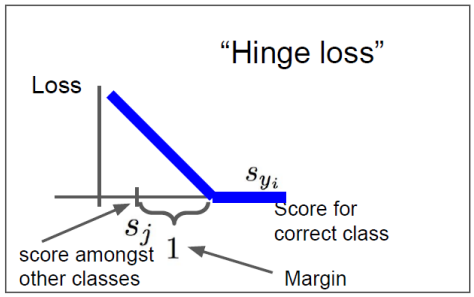
这种 max 啥的损失函数即Hinge Loss:

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Interpreting Multiclass SVM loss:



$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$
$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

如上述图像，对于单个样本讨论，损失函数关注的是 **正确类别得分** 和 **错误类别得分的差值**，尽量让正确类别的得分 高于 其他错误类别. 函数解释为:（注意自变量因变量）

$$L_i(s_{y_i}) = -s_{y_i} + (s_j + 1)$$

正确得分到了一定值，落在 $s_j + 1$ 区域就是一个safety margin了.

- 计算loss:

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9		

the SVM loss has the form:

$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$

= max(0, 5.1 - 3.2 + 1)
+max(0, -1.7 - 3.2 + 1)
= max(0, 2.9) + max(0, -3.9)
= 2.9 + 0
= 2.9

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	12.9

the SVM loss has the form:

$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$

Loss over full dataset is average:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

L = (2.9 + 0 + 12.9)/3
= **5.27**

思考题

Q1: What happens to loss if car scores decrease by 0.5 for this training example?
Q2: what is the min/max possible SVM loss Li?
Q3: At initialization W is small so all $s \to 0$. What is the loss Li, assuming N examples and C classes?
Q4: What if the sum was over all classes?
Q5: What if we used mean instead of sum?
Q6: $L_i = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right)^2$ 如何?

- Q1: 这里 SVM loss 的概念就在于，上述对中间图像的分类，如果car(正确分类)的得分变化了一点，只要还在那个margin里面(这里是大1的)，损失函数都不会改变.
- Q2: 观察上述图像，就会发现SVM loss最大是 $+\infty$，最小是0.

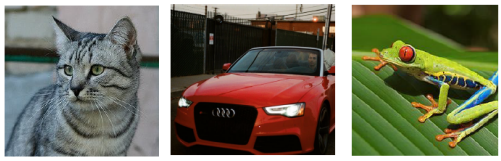
- Q3：注意到 刚开始训练的时候，W 是随机初始化的，s_j, s_{y_i} 都是非常靠近0的类似均匀分布的，此时 $L_i = C - 1$，这样总的loss大概就是 $L = C - 1$.
- Q4：注意到 在计算损失函数时候 $\sum$ 是跳过正确分类类别的，如果加入这个类别，损失函数的值将+1. 就是说 $L = \sum L_i$ 的最小值为1，这样不太直观，一般来说最小损失为0.
- Q5：相同.
- Q6：会有不同，loss function指示了什么error是算法care/trade off的.
- 为什么要加上 1：$s_j + 1$?

  这就是损失函数的一个常数(不重要)，因为我们关心的是正确分类得分和错误分类得分的 差值：$\forall\, j \neq y_i, |s_j - s_{y_i}|$.

模型选择

$L = 0$ 时，W 不唯一，例如 $2W$ 也可以：

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	

$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$

Before:
= max(0, 1.3 - 4.9 + 1)
+max(0, 2.0 - 4.9 + 1)
= max(0, -2.6) + max(0, -1.9)
= 0 + 0
= 0

With W twice as large:
= max(0, 2.6 - 9.8 + 1)
+max(0, 4.0 - 9.8 + 1)
= max(0, -6.2) + max(0, -4.8)
= 0 + 0
= 0

模型不只是需要拟合训练数据，由此引入过拟合，正则项：

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Occam's Razar: Among multiple competing hypotheses, the simplest is the best,
William of Ockham 1285-1347

Simple examples
L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$
L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$
Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

More complex:
Dropout
Batch normalization
Stochastic depth, fractional pooling, etc

- 模型选择：Occam's Razar.
- 有不同的作用，比如 L_1 更鼓励稀疏，用一个例子看 L_1 和 L_2 的偏好，L_2 更倾向于铺开的W，这可能更鲁棒(在x_i有较大变化时)：

Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$
$$w_1 = [1, 0, 0, 0]$$
$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will the L2 regularizer prefer?

L2 regularization likes to "spread out" the weights

Which one would L1 regularization prefer?

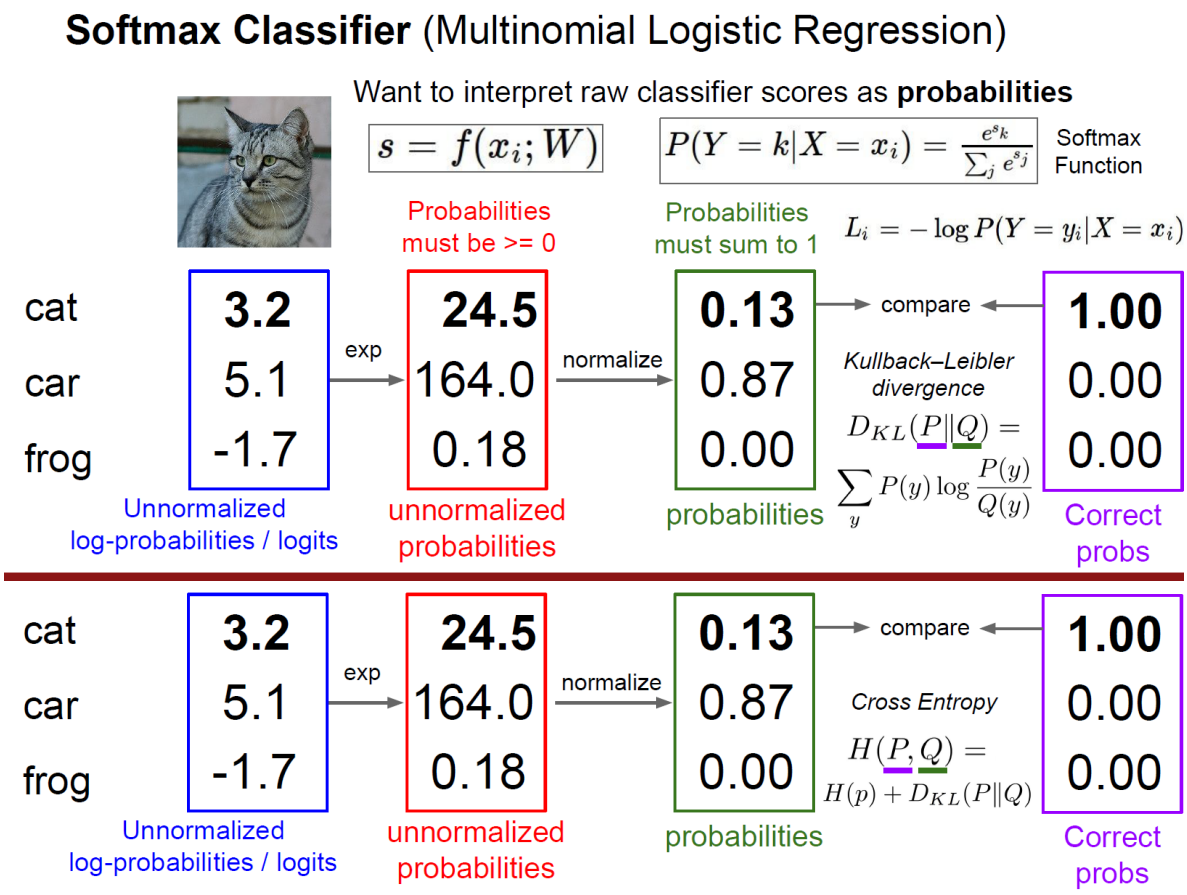
上面还有一些较复杂的正则化方法，都是在试图惩罚模型的复杂度，而不是试图拟合数据.

Softmax Classifier(Multinomial Logistic Regression)

score经过softmax之后，变成概率. 我们希望的是正确类别的预测概率接近1. 所以loss的考虑为:

$$L_i = -\log P(Y = y_i \mid X = x_i) = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

注意这里还有其他的优化目标，比如 转为预测分布和目标分布的比较:



思考题

- L_i（对于一个样本的损失）最大值为 $+\infty$，最小值为0.
- 注意这里softmax loss是不断地优化，让正确的预测概率越来越大，而不像SVM loss一样，只要正确类别的score最高，即落到了那个margin里面就OK.

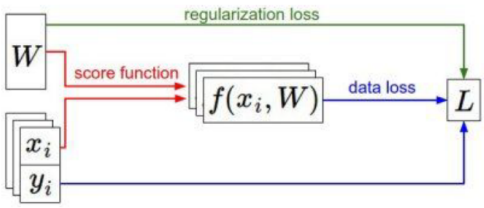
Softmax

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$ Full loss



3.2 优化

梯度下降

Strategy #2: **Follow the slope**

In 1-dimension, the derivative of a function:

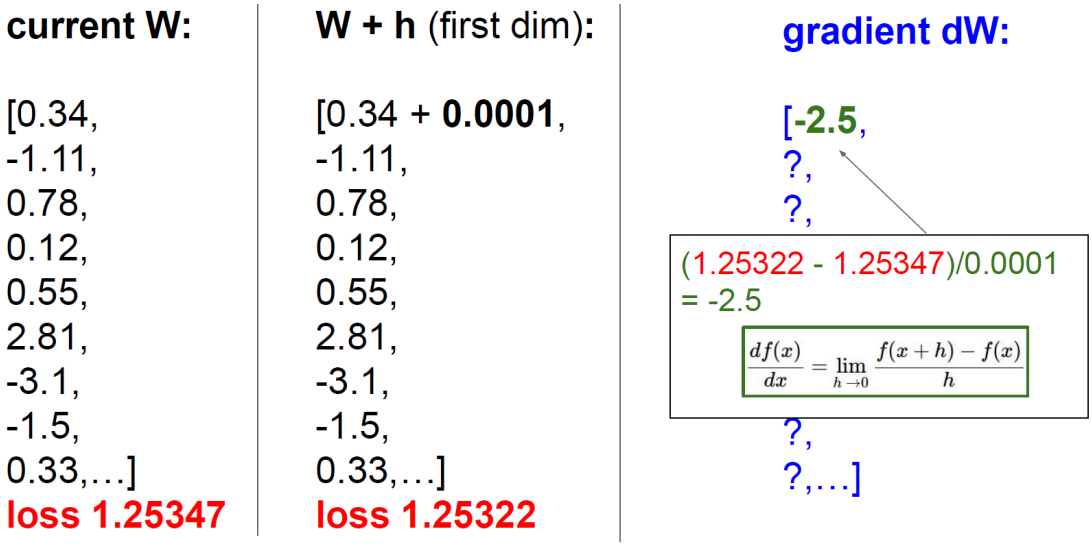
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

计算梯度：

- **numerical gradient:** 有限差分法(the method of finite differences): 这回到了梯度的极限定义：



某一维度上加一个小量 h，由极限定义的梯度计算该维度上的梯度值．实现有限差分逼近．

- 上述有限差分是一种方法，但是实际中我们使用微积分**直接写出梯度表达式**：精确且快速．
- 如何验证代码中实现的 直接计算梯度表达式 是否正确?

  使用 analytic gradient，但是**gradient check**的时候使用**numerical gradient.**

- 代码：

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

优化梯度下降的方法(以后会讲)：1. 带momentum，2. Adam优化.

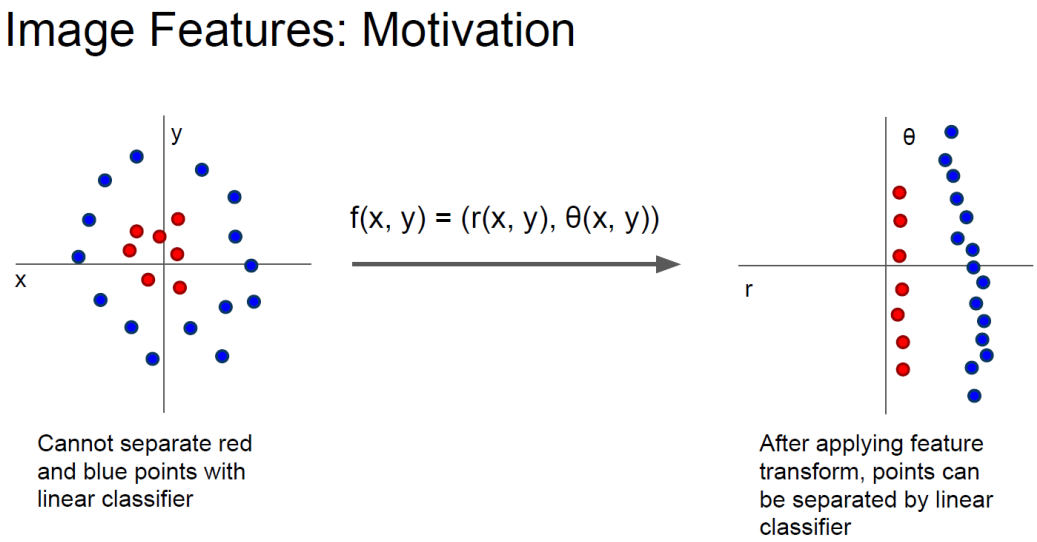
随机梯度下降

并非计算整个训练集的误差和梯度值. 而是在每次迭代中，采样一个**minibatch(**大小**32/64/128)**. 在minibatch上估计误差，实际梯度.

线性分类器交互网站：http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/ 有点意思哈，调整学习率感受一下.

4.1 神经网络 和 反向传播

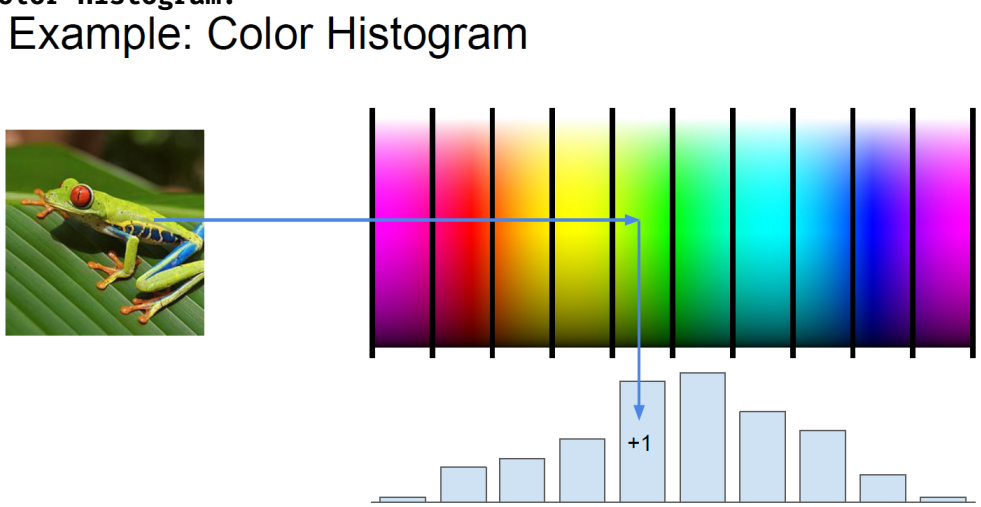
特征提取方法



常常需要先对数据提取特征，上面是一个特征变换：极坐标转换.

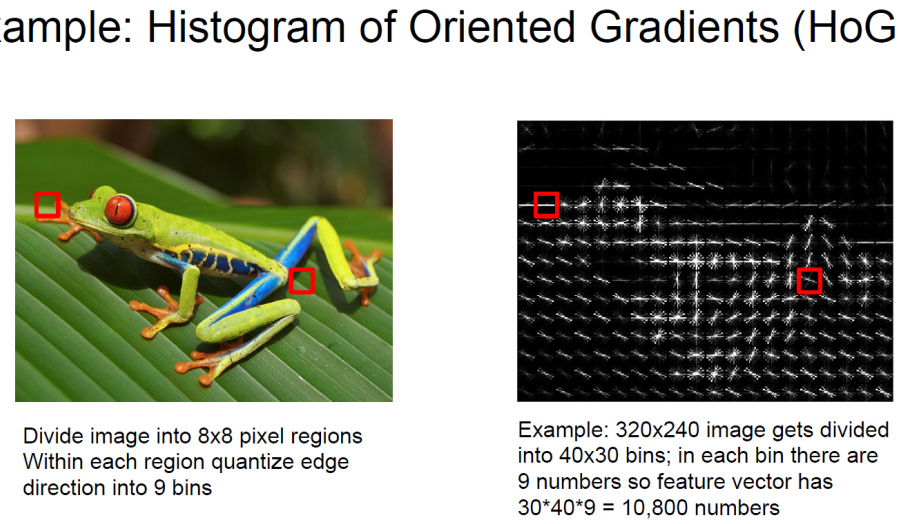
例子：

- **Color Histogram:**



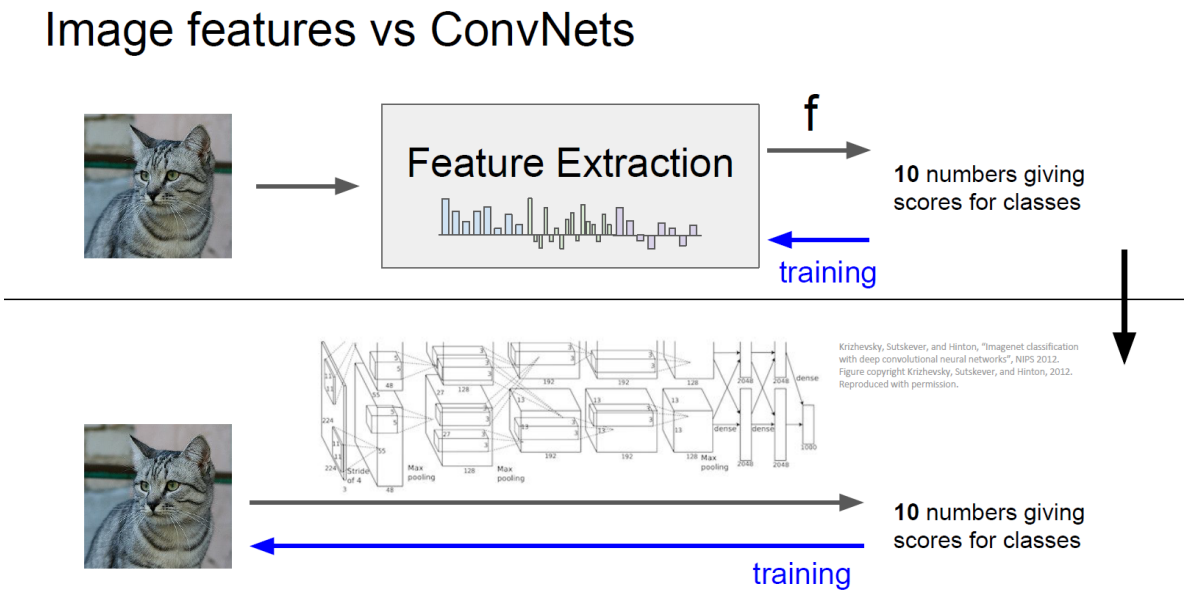
计算图片上每个像素 的颜色柱状图. 柱状图值代表频次.

- **Histogram of Oriented Gradients (HoG) 方向梯度直方图：** 在神经网络之前，提取图片的边缘特征，先一张图片划分成许多许多小块，每个小块里面识别特征，曾经的目标检测特征表示：



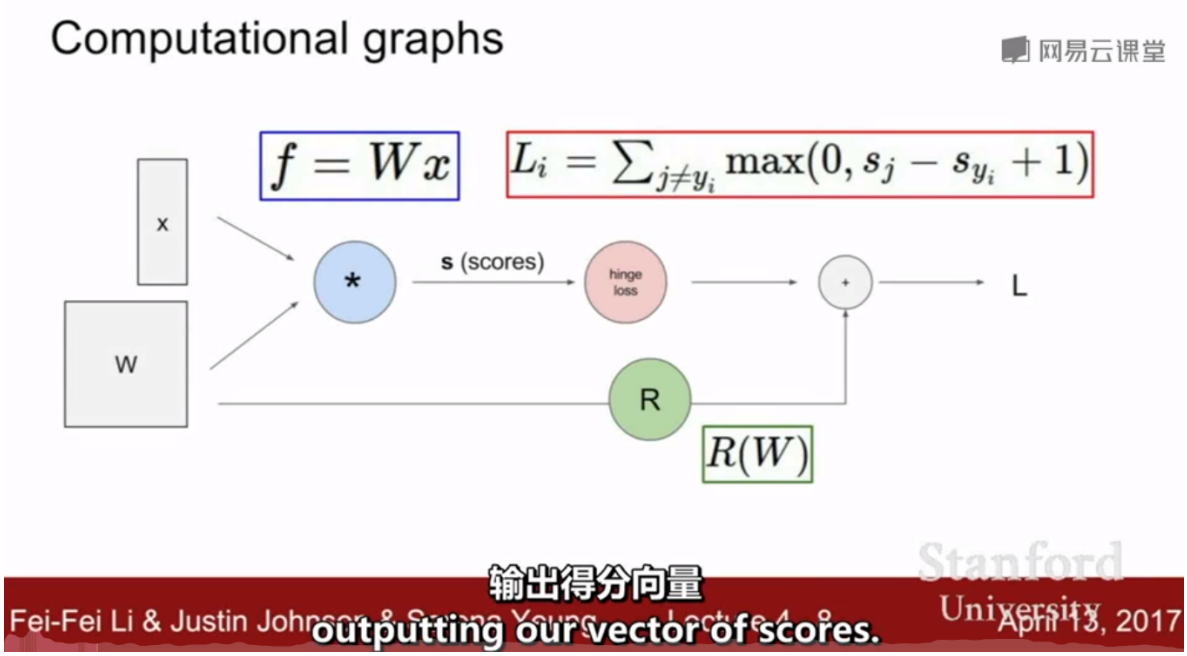
- **Bag of Words:**

  源于NLP，词频率表达，这里对于图像每一块就是视觉单词；随机块采样，K-means聚类得到簇中心，即不同的视觉单词表示.

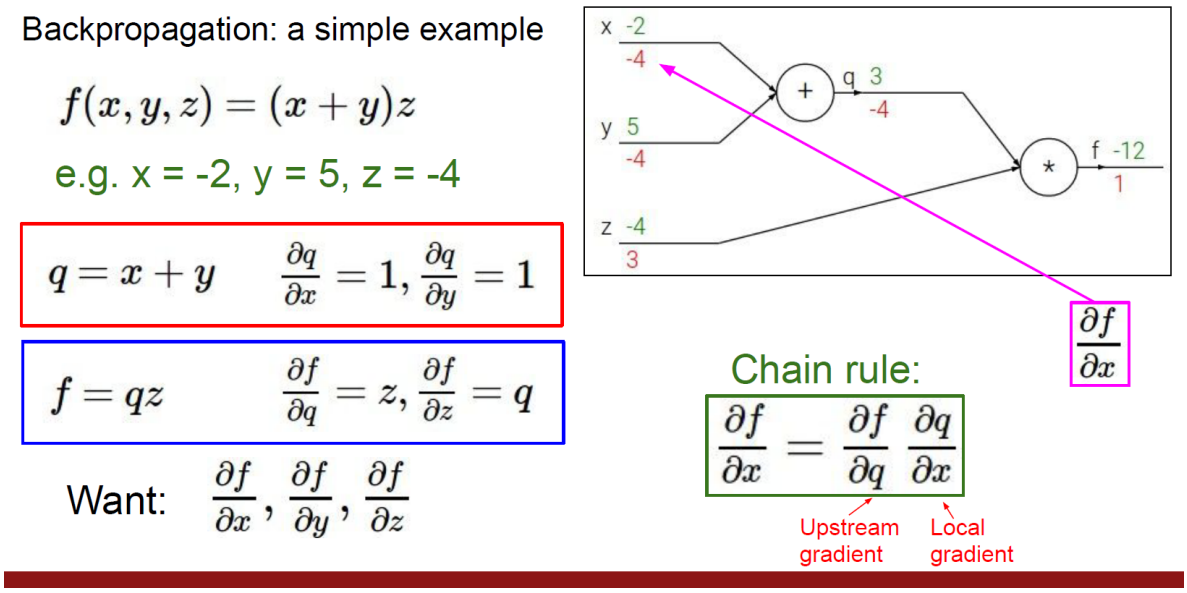


- 如上描述的特征提取方法，是直接 提取特征，并提前记录特征，在之后做分类.
- 深度学习方法是 从数据中学习特征. 在整个网络中训练参数.

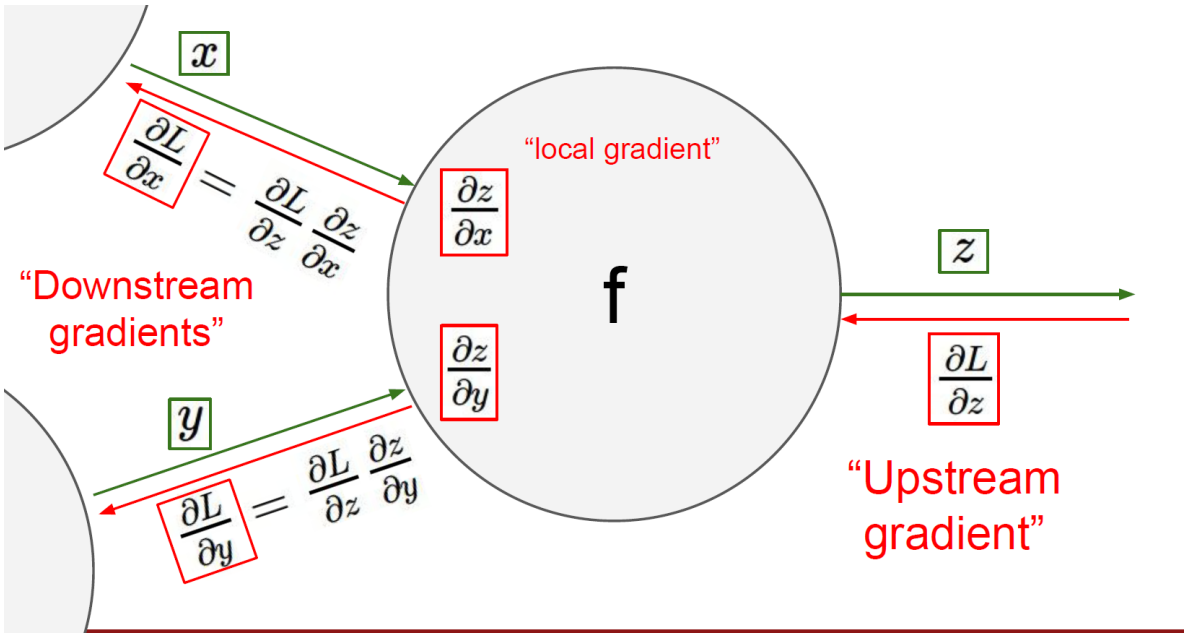
计算图：



反向传播 backpropagation

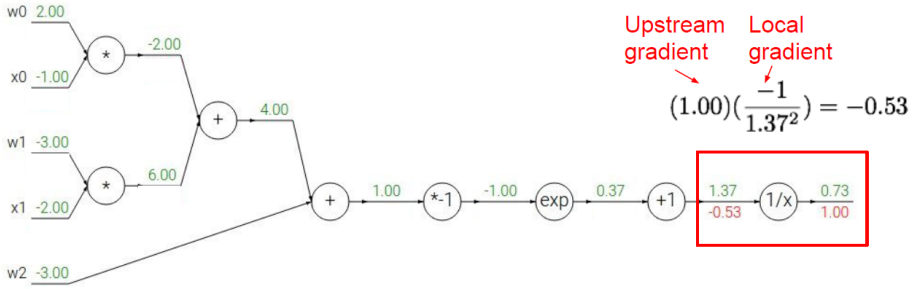


形象：



反向传播将输出对输入的求导做了 **化解成一层层的小问题 的简化**。看下面这个较复杂的例子即可体会：

Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$

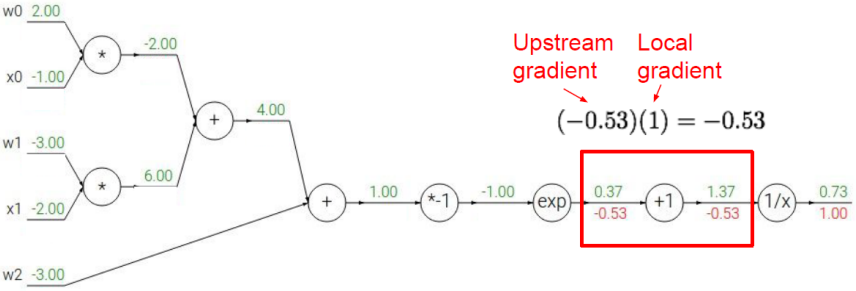


$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$

$f_a(x) = ax \rightarrow \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$

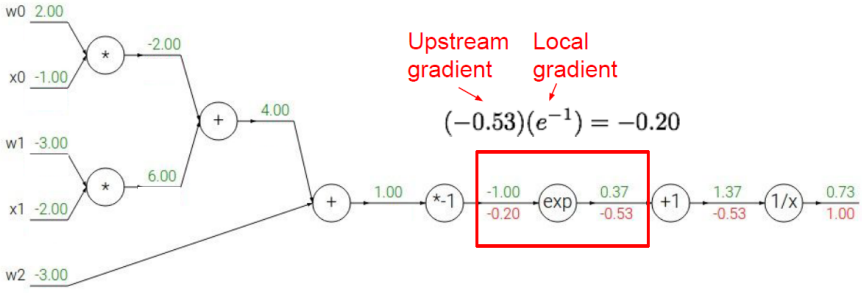


$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$

$f_a(x) = ax \rightarrow \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$



$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$

$f_a(x) = ax \rightarrow \frac{df}{dx} = a$

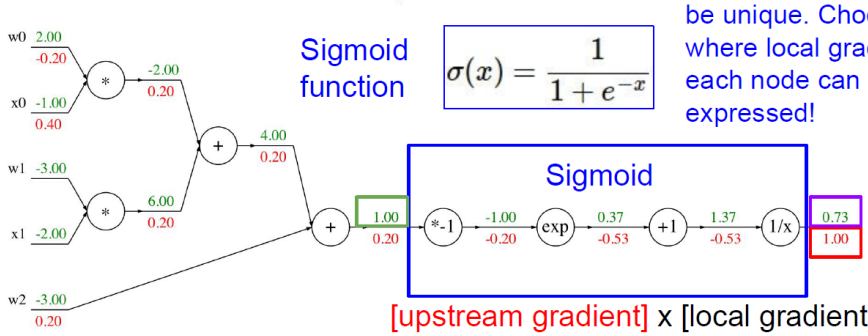
$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$

[upstream gradient] x [local gradient]
[1.00] x [(1 - 0.73) (0.73)] = 0.2

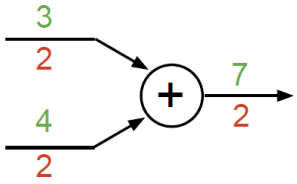
Sigmoid local gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

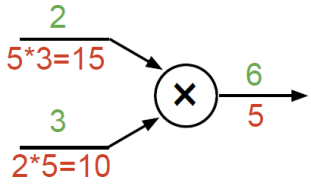
在计算图中不同 门/神经元 的反向传播梯度作用：

Patterns in gradient flow

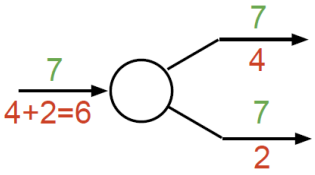
add gate: gradient distributor



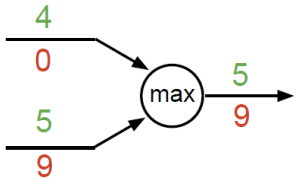
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router



矩阵求导方式与维度关系：

Recap: Vector derivatives

Scalar to Scalar

$x \in \mathbb{R}, y \in \mathbb{R}$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Vector to Scalar

$x \in \mathbb{R}^N, y \in \mathbb{R}$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$$

For each element of x, if it changes by a small amount then how much will y change?

Vector to Vector

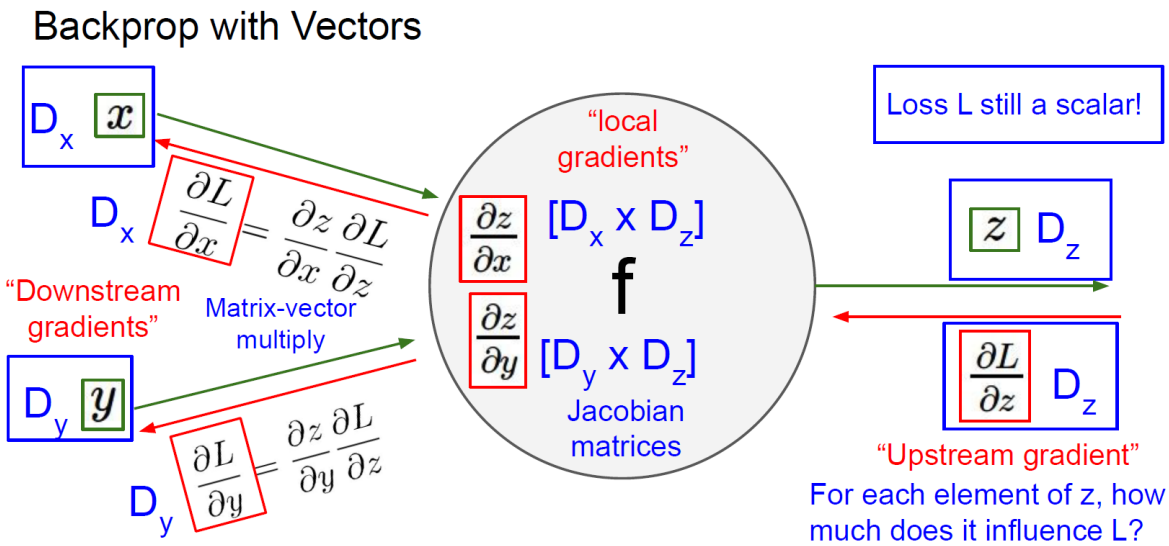
$x \in \mathbb{R}^N, y \in \mathbb{R}^M$

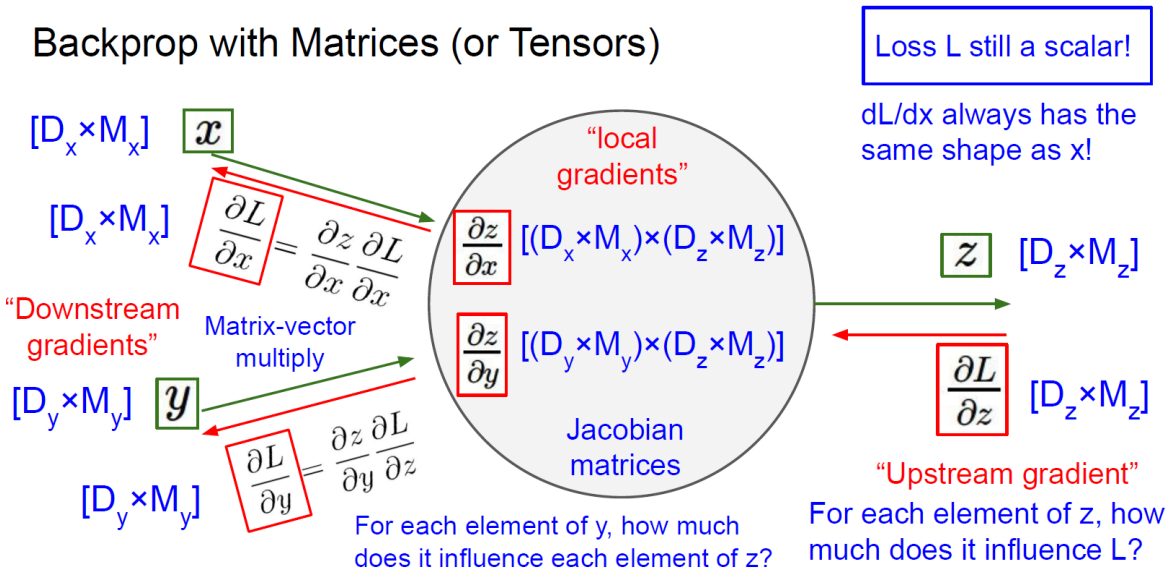
Derivative is **Jacobian**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N\times M} \quad \left(\frac{\partial y}{\partial x}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

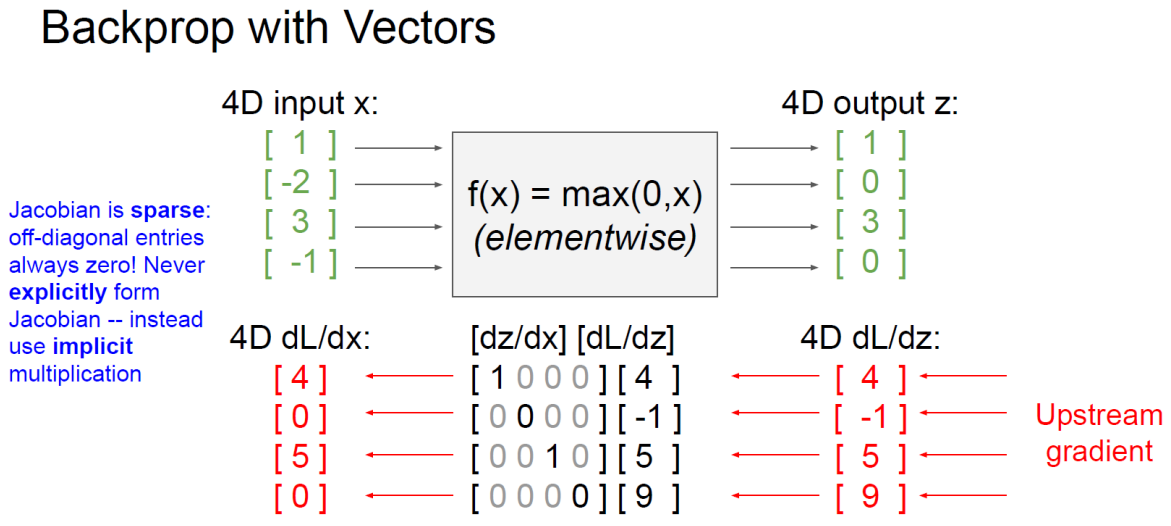
For each element of x, if it changes by a small amount then how much will each element of y change?

推广到高维的神经网络，就是变成梯度矩阵(Jacobian matrices)





Jacobian 矩阵分析

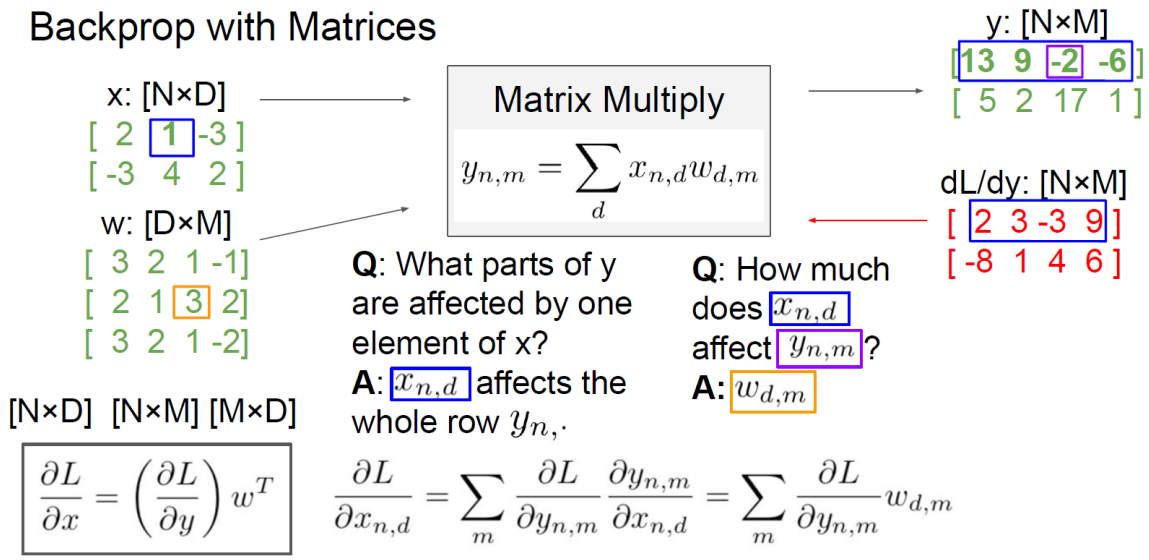


4096 的维度，喂给一个**ReLU**，输入输出都是4096维度，这样它的Jacobian矩阵大小为 `4096 x 4096`，再加上minibatch的大小，计算量太大了.

但是 Jacobian 矩阵的特点:

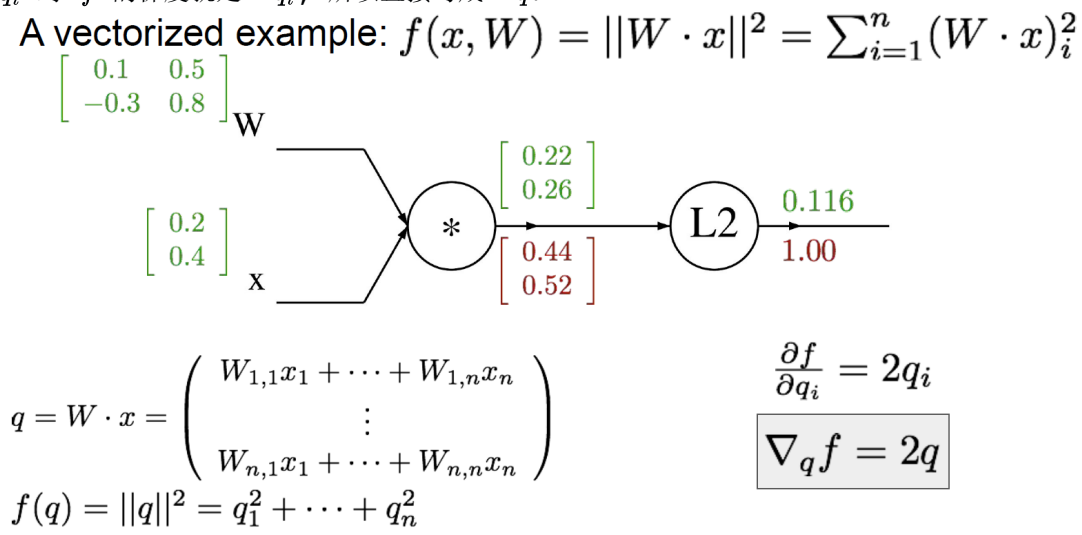
- 对角阵. 因为单层ReLU, 第i个输入仅和第i个输出有关.

一种解决方法:

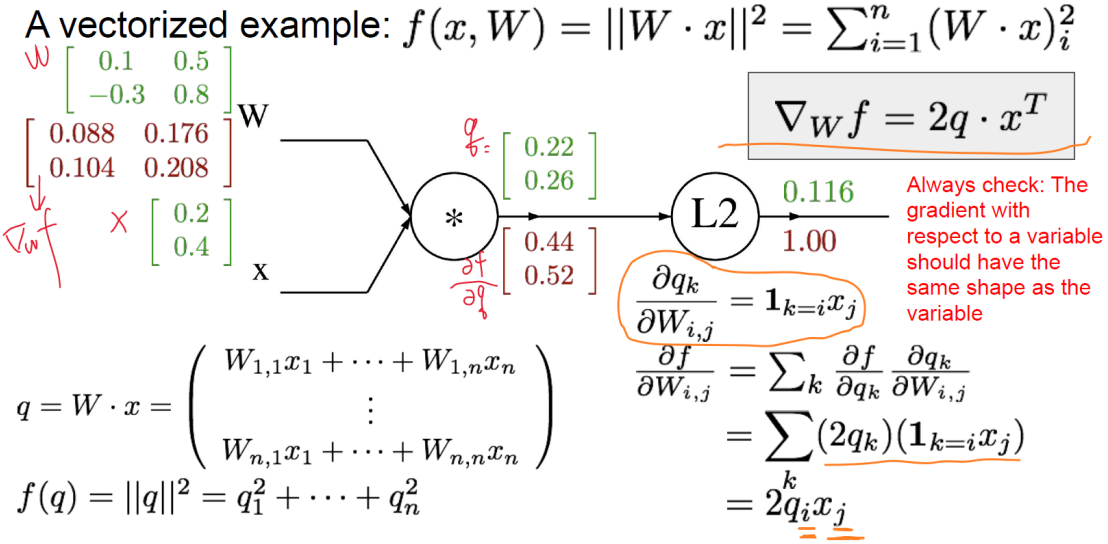


详细的例子 线性+L_2:

- q_i 对 f 的梯度就是 $2q_i$, 所以直接写成 $2q$:



- 再看 q_k 对于 W_{ij} 的影响:

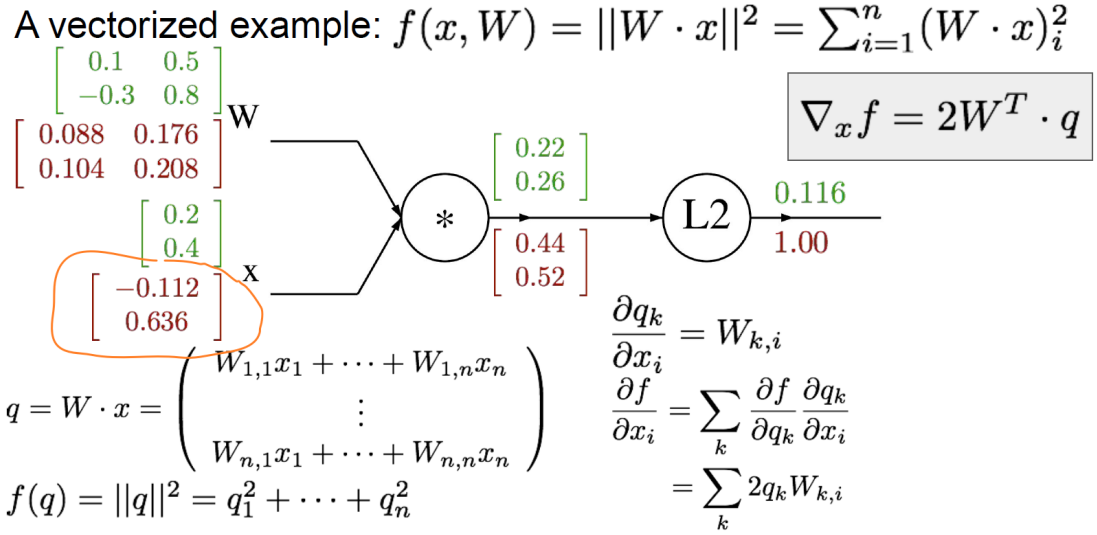


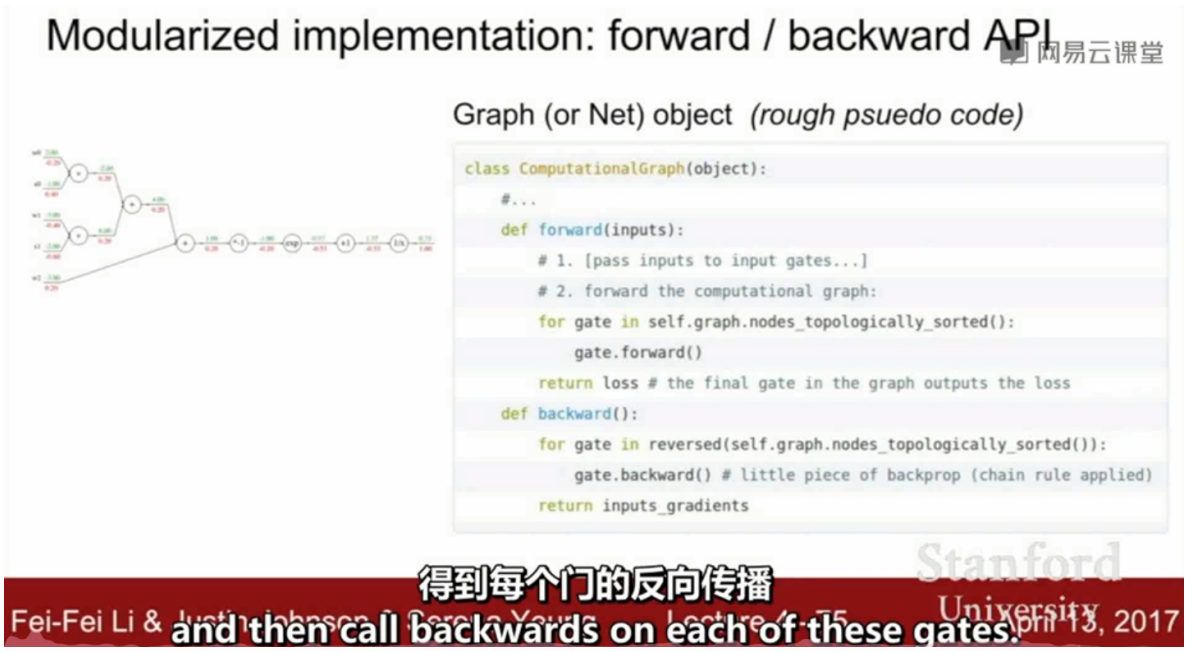
注意上图中各个位置的矩阵是如何求出来的(橙色线).

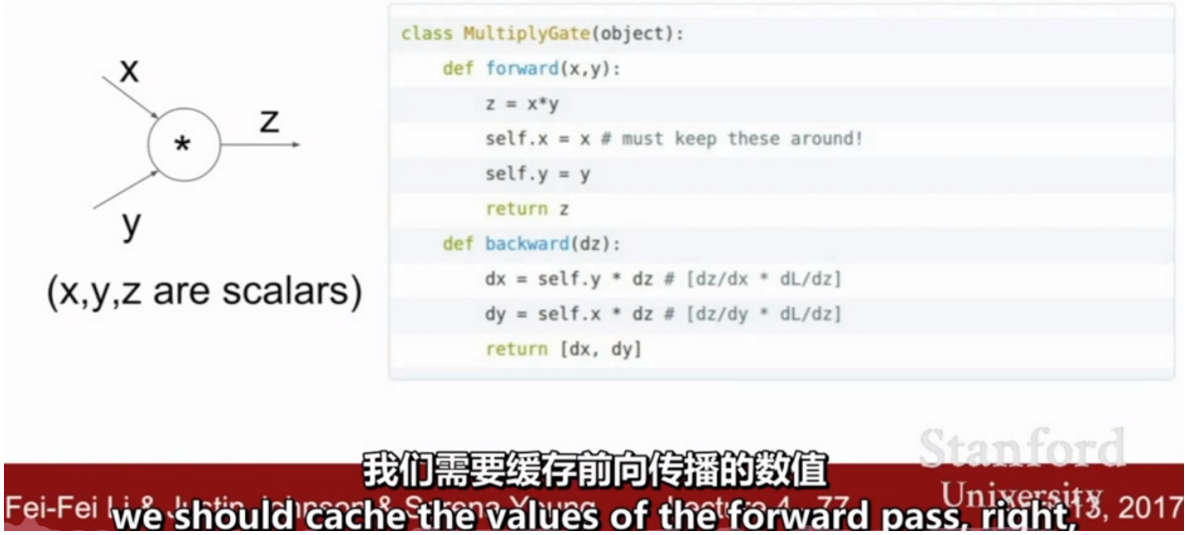
注意 q_k 对 W_{ij} 的梯度有 **指示函数**的限制, 这也说明了得到的Jacobian矩阵是稀疏的概念, 使用链式法则得到$2q_i x_j$(仔细考虑 求梯度时element-wise的影响范围, 可以先用几个特例试试).

最后得出 $2q \cdot x^T$.

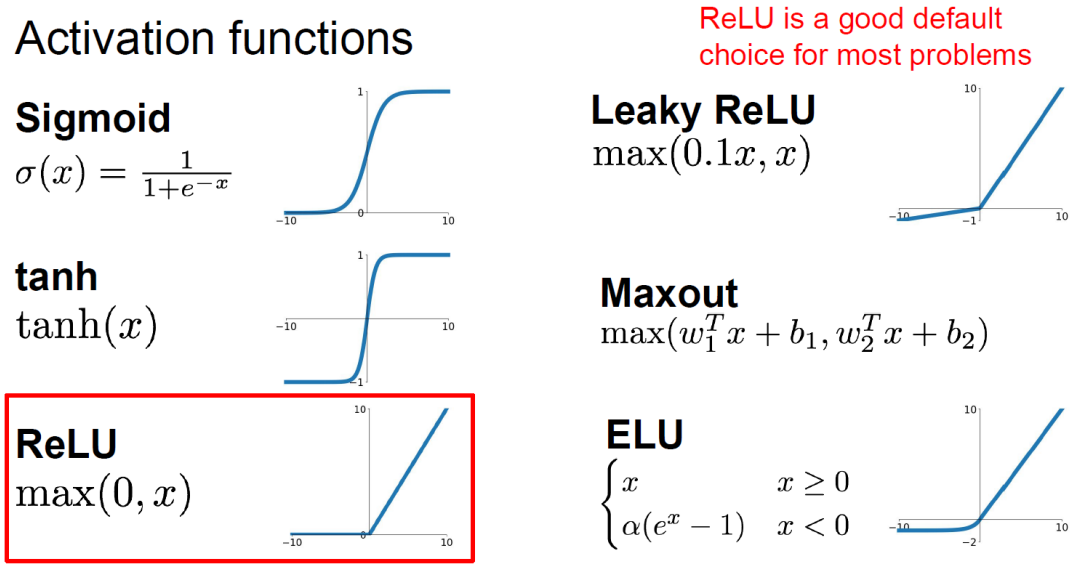
同理有：







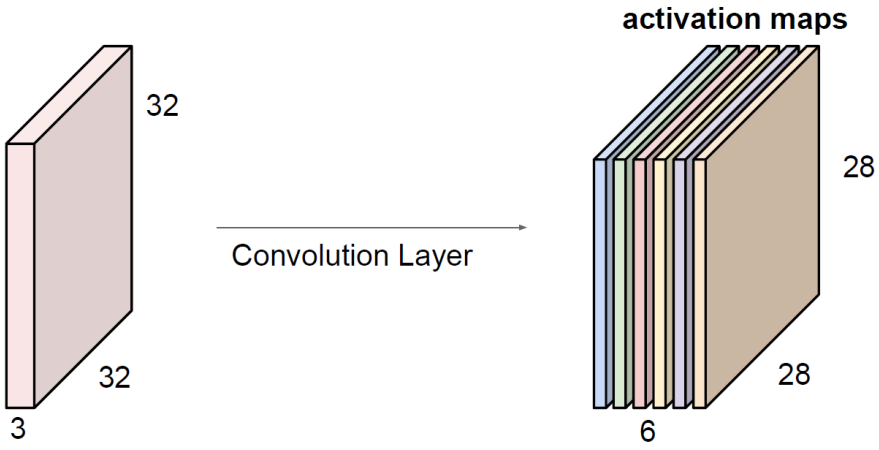
激活函数：



5.1 卷积神经网络

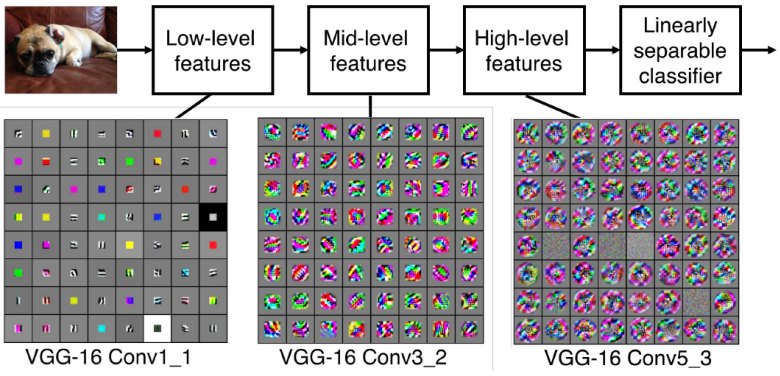
如果有6个卷积核，我们可以得到：

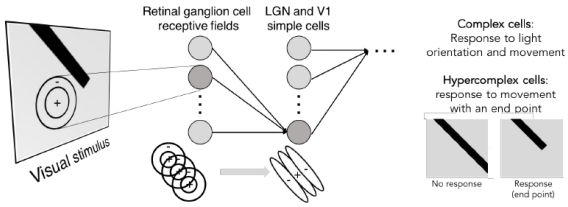
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

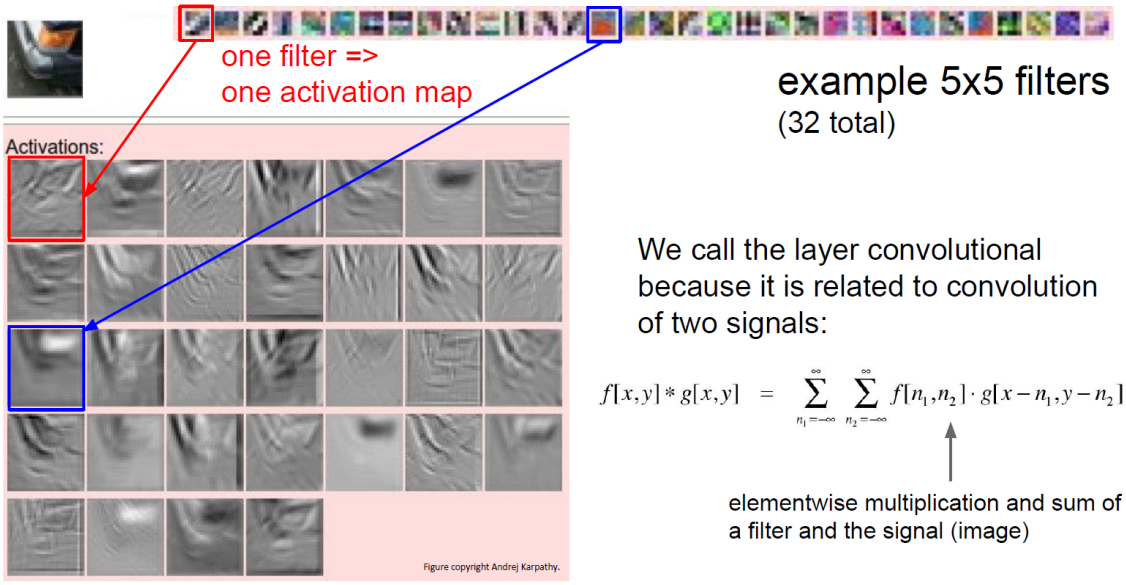
Preview





在上图中，每一个格子都是神经元，一定程度上代表了神经元在寻找什么.

不同的kernel在一幅图像上卷积的结果，比如第一个红色框的，就像在找图片中的边缘，更关注边缘：

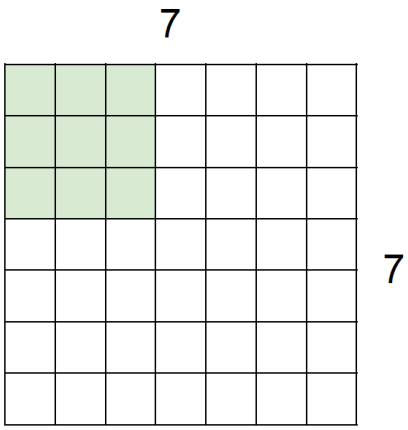


卷积操作，和两个信号之间的卷积操作有关.

filter 及输出大小相关

深入探究 stride:

A closer look at spatial dimensions:

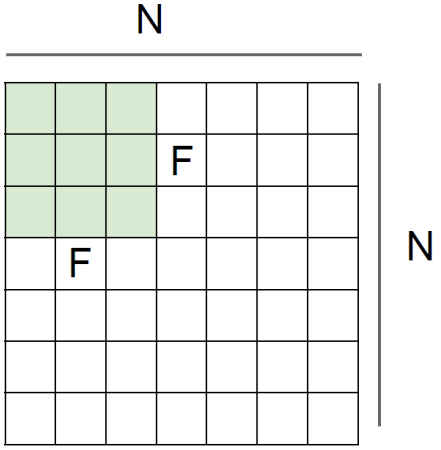


7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.

在 `7 x 7` 上的图不能使用 `3 x 3` 的filter.

计算输出尺寸的公式，同时可以来判断什么stride是OK的:



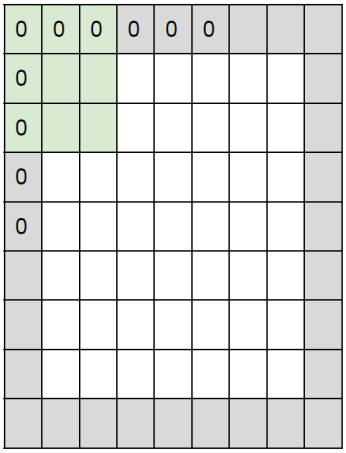
Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

$$Output = (N - F)/stride + 1$$

- 使用**padding:**（zero padding）

In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

(recall:)
(N + 2P - F) / stride + 1

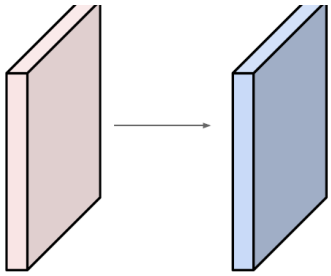
padding了周围一圈 ⇒ `7 x 7` → `9 x 9`，注意是 加 2P.

如上加了一圈padding之后，输出的图片大小就和卷积之前的图片大小相同了．保持图像尺寸．

我们不希望多层卷积之后图片尺寸变得很小，因为这会损失很多信息．

例题：

Examples time:

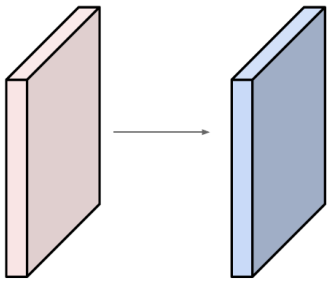


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

参数个数，注意每个filter还有一个bias:

Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

总结关于 `filter` 以及输出尺寸大小的这节：

Convolution layer: summary

Let's assume input is W_1 x H_1 x C
Conv layer needs 4 hyperparameters:
- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of W_2 x H_2 x K
where:
- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

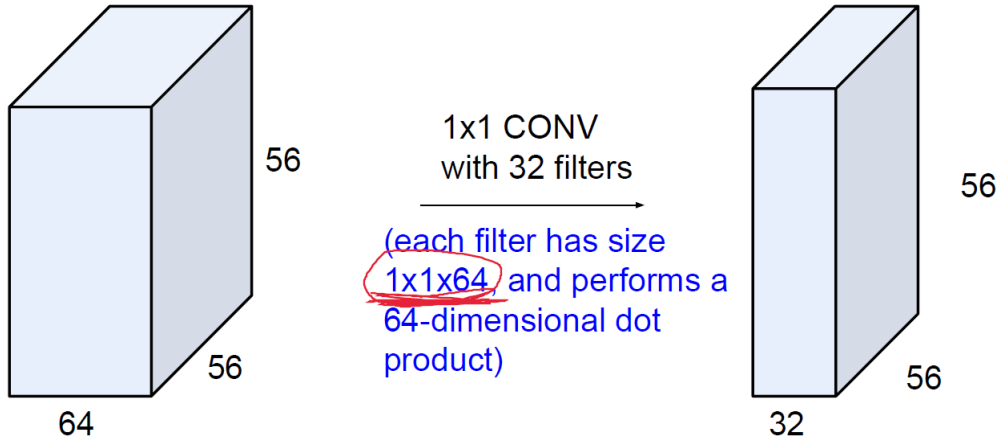
Number of parameters: F^2CK and K biases

Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512
- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

`1 x 1` 的卷积操作也是可以的，注意下面每个`filter`是 `1 x 1 x 64` 的，每次卷积 卷积核都是对图片上所有通道上操作，然后有32个构成了输出深度是32的：

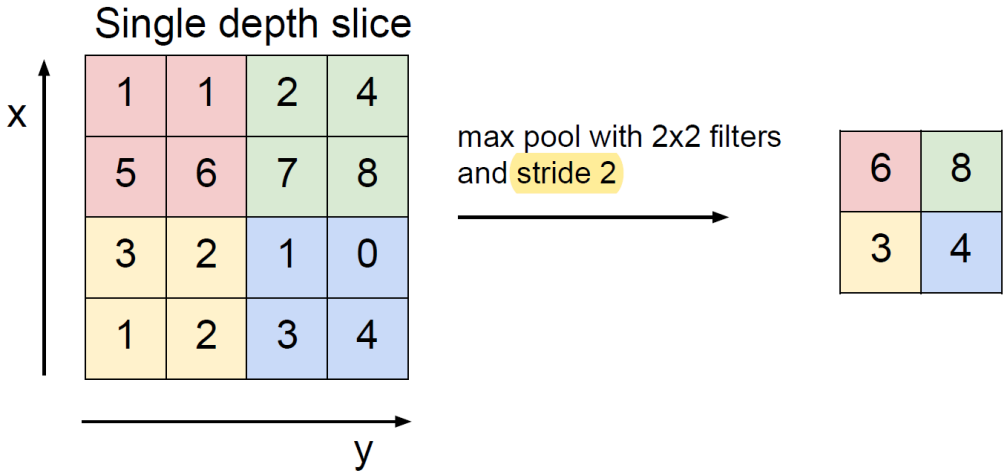
(btw, 1x1 convolution layers make perfect sense)



Pooling layer

- makes the representations smaller and more **manageable.**
- operates over each activation map independently.

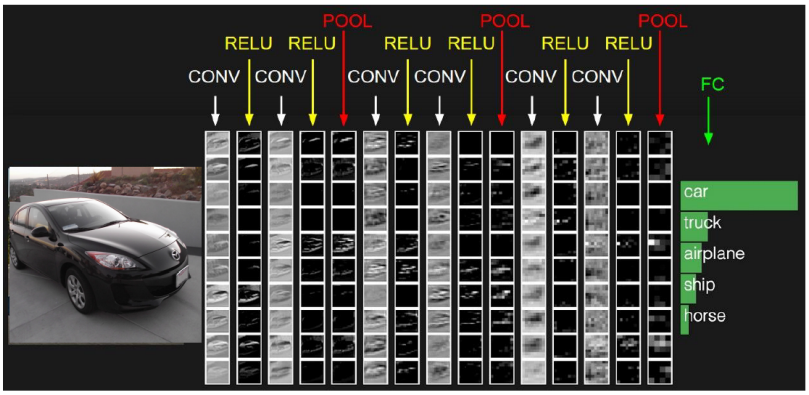
MAX POOLING



- 不做深度上的池化处理，深度不变.
- 通常来说，pooling的filter是没有重叠的.
- 一般不做zero padding，因为pooling本身就是降采样.

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



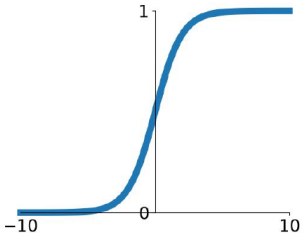
上图每一层的输出都是激活映射的输出.

6.1 激活函数

- **sigmoid**

Activation Functions

$$\sigma(x) = 1/(1+e^{-x})$$



Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

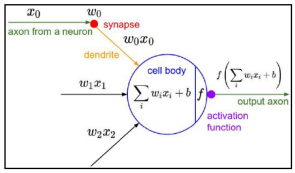
  - 饱和神经元使梯度消失:

    0（梯度）会传递到下游结点. 当值较大或者较小的时候，因为sigmoid那里是平的.

- 输出不是以0为中心的，这将导致：

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on **w**?

We know that local gradient of sigmoid is always positive
We are assuming x is always positive

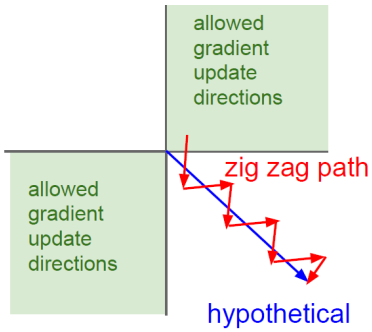
So!! Sign of gradient **for all w_i** is the same as the sign of upstream scalar gradient!

$$\frac{\partial L}{\partial w} = \sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times upstream\_gradient$$

全正或全负的梯度 对于梯度的更新是低效的 zig zag path:

~~always positive...~~

$$f\left(\sum_i w_i x_i + b\right)$$

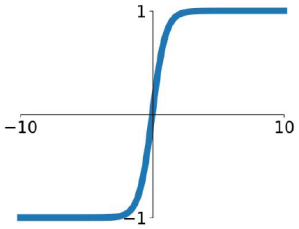


What can we say about the gradients on **w**?
Always all positive or all negative :(

这也是为什么要使用均值为0的输入数据.

接下来引出 以0为中心的 激活函数.

- **tanh**



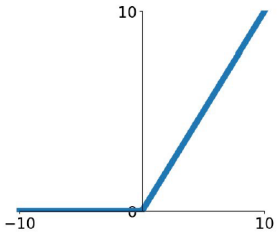
tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

但是仍然会存在梯度消失.

- **ReLU**

Activation Functions



ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

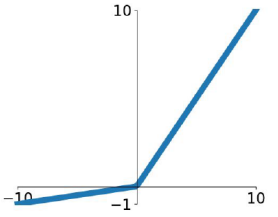
annoyance: <0 的区域出现梯度消失.

dead-ReLU, 学习率太大的时候, 刚开始训练很正常, 后面ReLU结点的参数就不更新. 超平面就不动, 因为负的梯度为0不更新了.

- **Leaky ReLU**

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

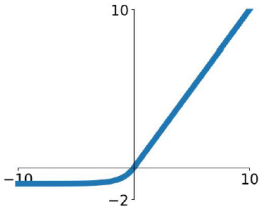
Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

- **ELU**

Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

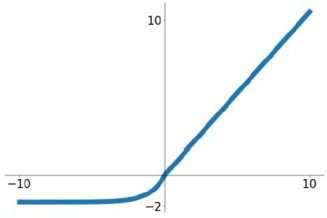
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

(Alpha default = 1)

- Computation requires exp()

- **SELU**

Scaled Exponential Linear Units (SELU)



- Scaled versionof ELU that works better for deep networks
- "Self-normalizing" property;
- Can train deep SELU networks without BatchNorm
  - (will discuss more later)

$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

α = 1.6733, λ = 1.0507

- **Maxout**

Maxout "Neuron"

[Goodfellow et al., 2013]

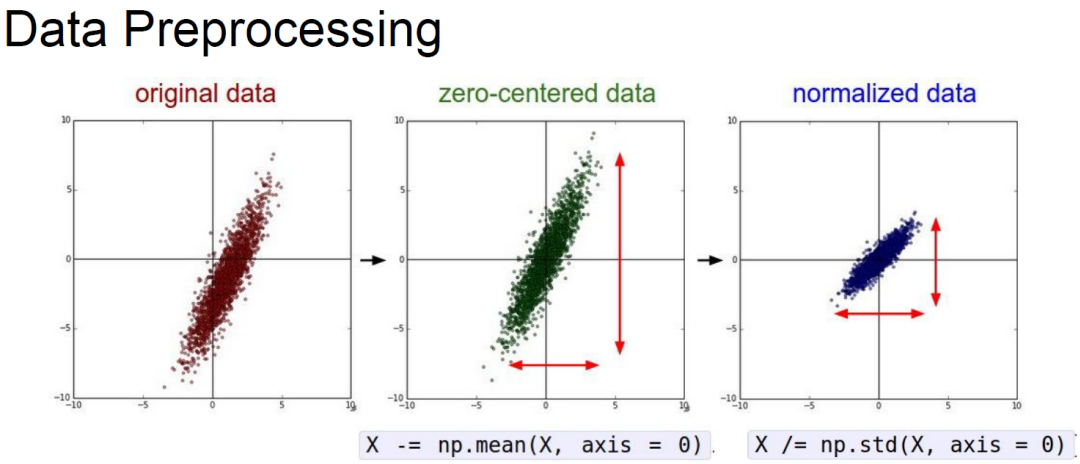
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

- 实践建议：

所以为什么要做数据预处理，在激活函数这里有一定的体现：



(Assume X [NxD] is data matrix,
each example in a row)

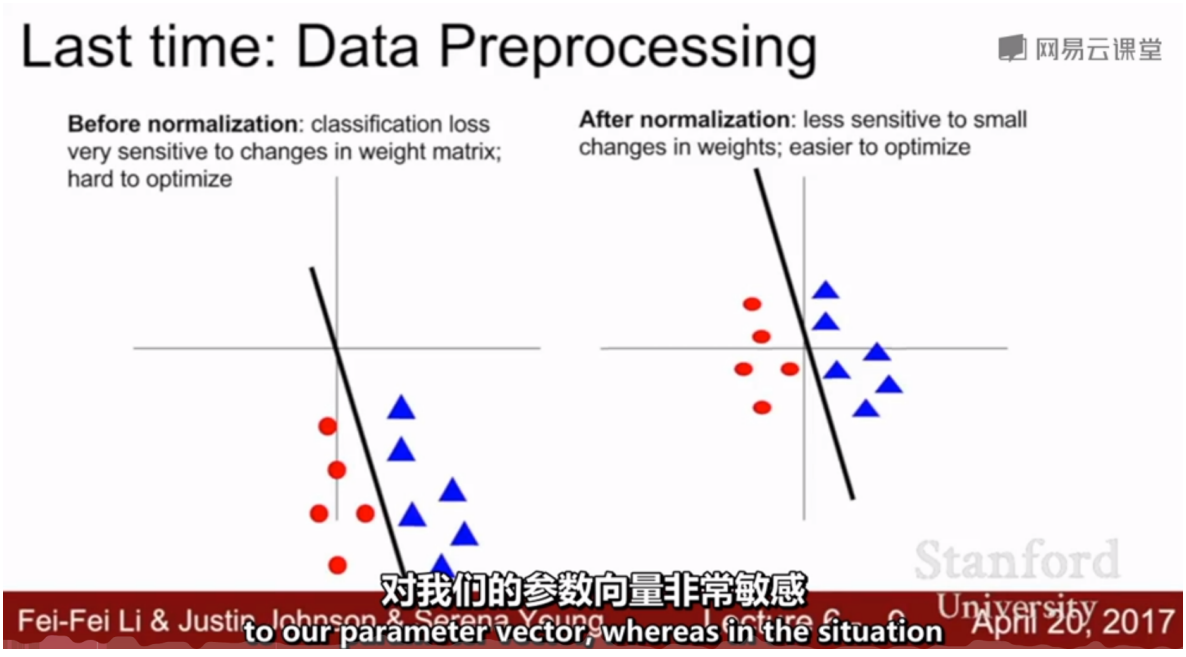
- `zero-centered:`

  比如刚刚ReLU的例子，不能全正/全负的输入.

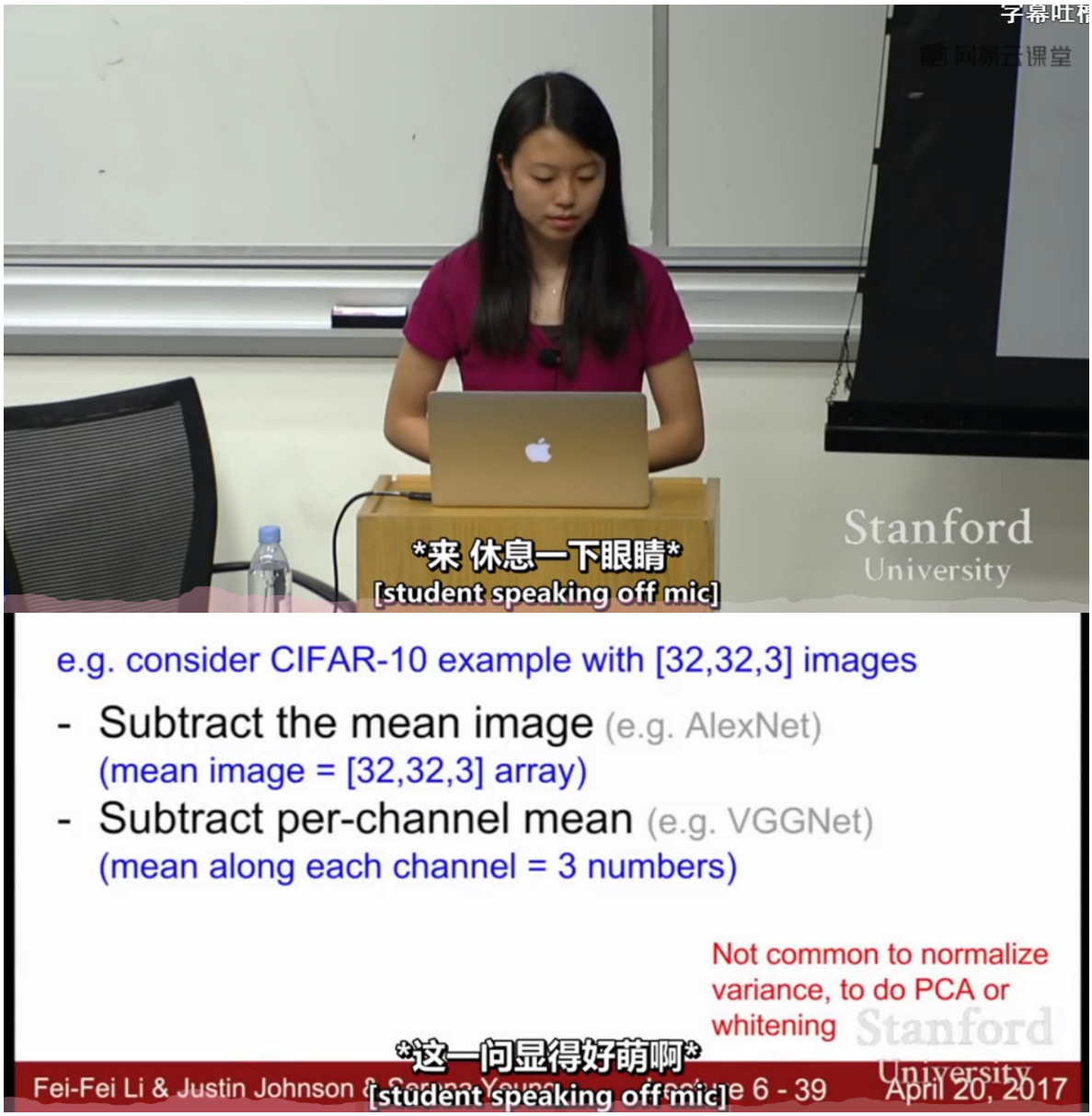
- `normalize:`

  所有特征在相同的值域内，这样每个特征贡献相同.

还有一个角度的解释，移动到中间之后对于分类边界的微小扰动就不那么敏感了：



如上，在神经网络中也是一样的，我们要交叉使用 线性，卷积，非线性激活，如果某一层不是Normalization的，那么该层权值矩阵的微小摄动，就会造成该层输出的巨大变动.



参数初始化

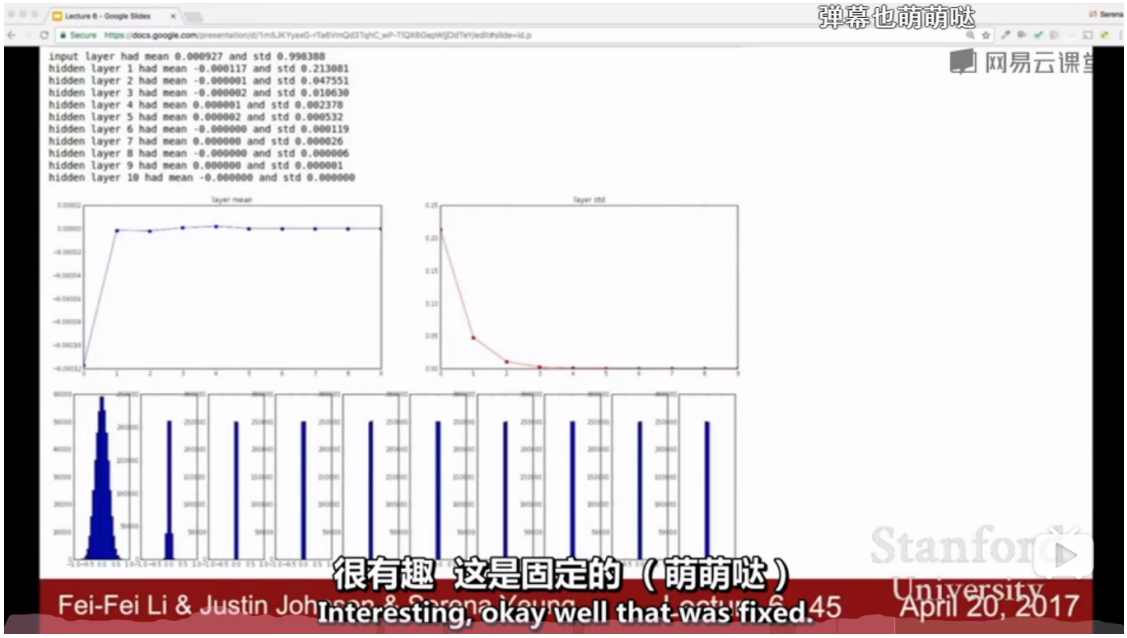
- **Q: what happens when $W = constant$ init is used?** 对于最普通的全连接网络.

  所有神经元会做同样的事情，它们可能不会不变，是否不变取决于输入. 因为梯度啥的都是一样的.

- 所以正确的初始化是从一个概率分布中**随机抽样**:

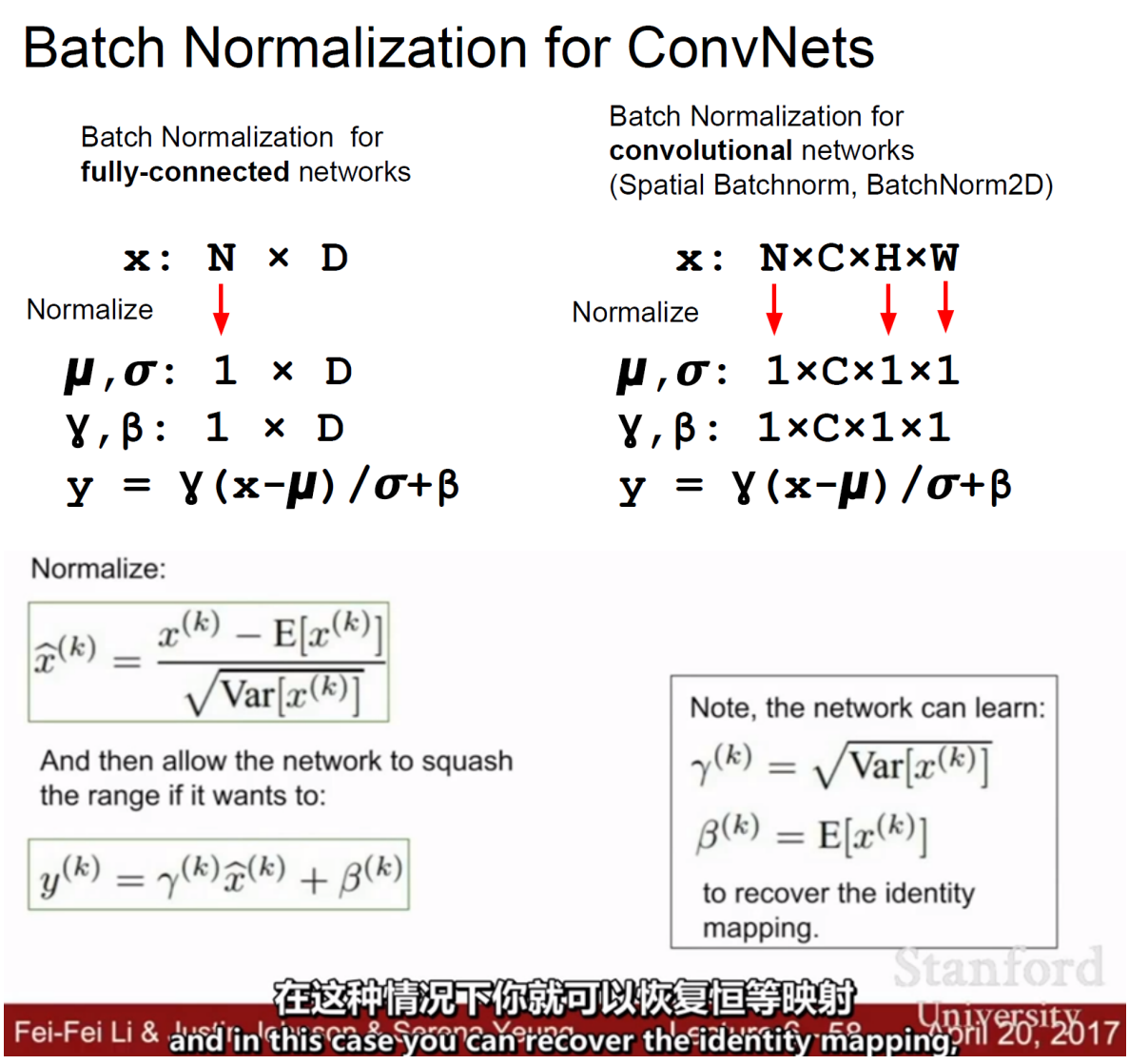
  ```
  1 | w = 0.01 * np.random.randn(Din, Dout)
  ```

  这在小型网络中是适用的，但是**在深度网络中会出现问题**:



Batch Normalization

我们希望在每一层都有很好的单位高斯分布．并在训练时一直保持．



Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$ Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$ Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ Output, Shape is N x D

注意进来计算的都是对于 **minibatch** 的.

使用Batch Normalization, 训练会变得容易, 也可以将其看成正则化方法, 因为这些激活值等前面网络过来的都是源于输入 X.

梯度不更新(或者更新很小很慢), 可能是learning rate设置太低了.

- 可能存在 loss 不怎么变化, 但是accuracy却在下降的情况:
  是因为 参数 的分布不太变化, 但是整体朝accuracy对的方向移动.

出现 cost NaN 意味着已经非常大, 爆掉了. 这时候很可能是因为learning rate设置太大了.

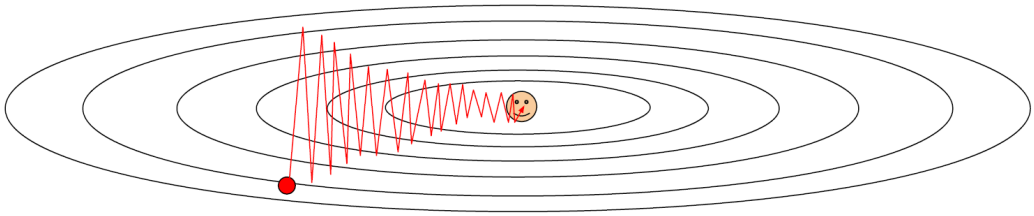
random search 优于 grid search 的原因很可能是对参数空间覆盖得更好.

7.1 优化

SGD的一点问题:

Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

- 

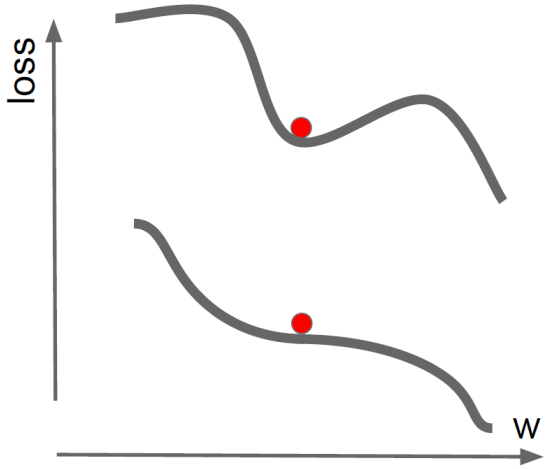
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

- 存在局部最小值点，鞍点，这些位置都是梯度为0的，鞍点在高维时比局部极小更为常见：

Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck



- 请注意 SGD 在每一点都是计算梯度的一个估计，此时存在噪声，寻找最优点的时间就长.

解决方法：

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

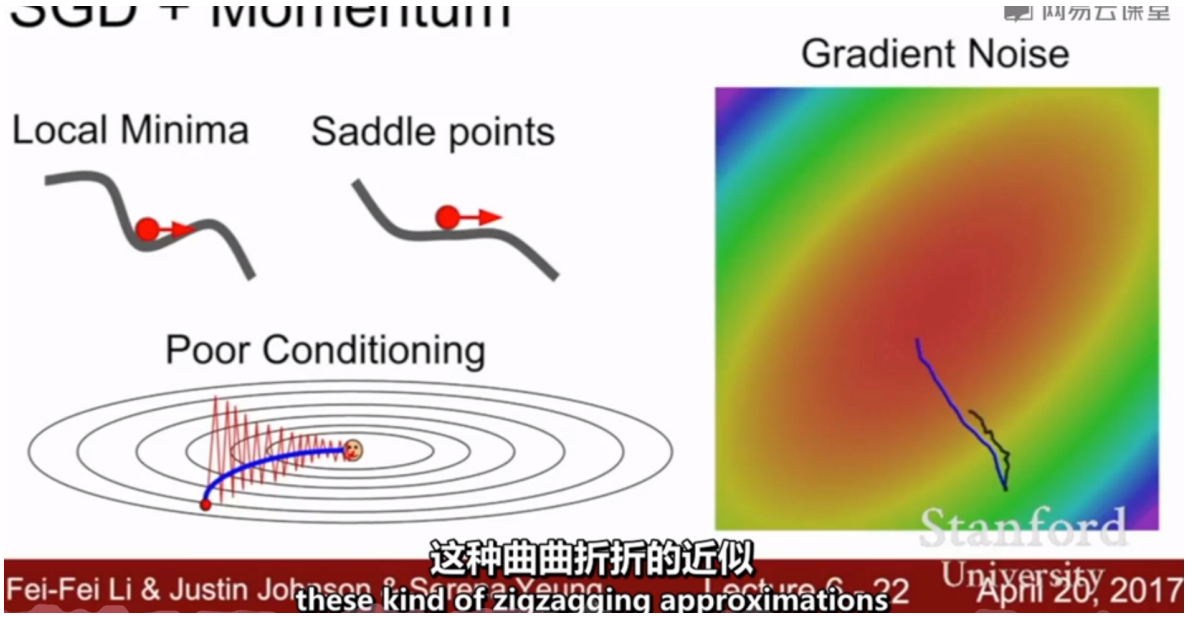
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

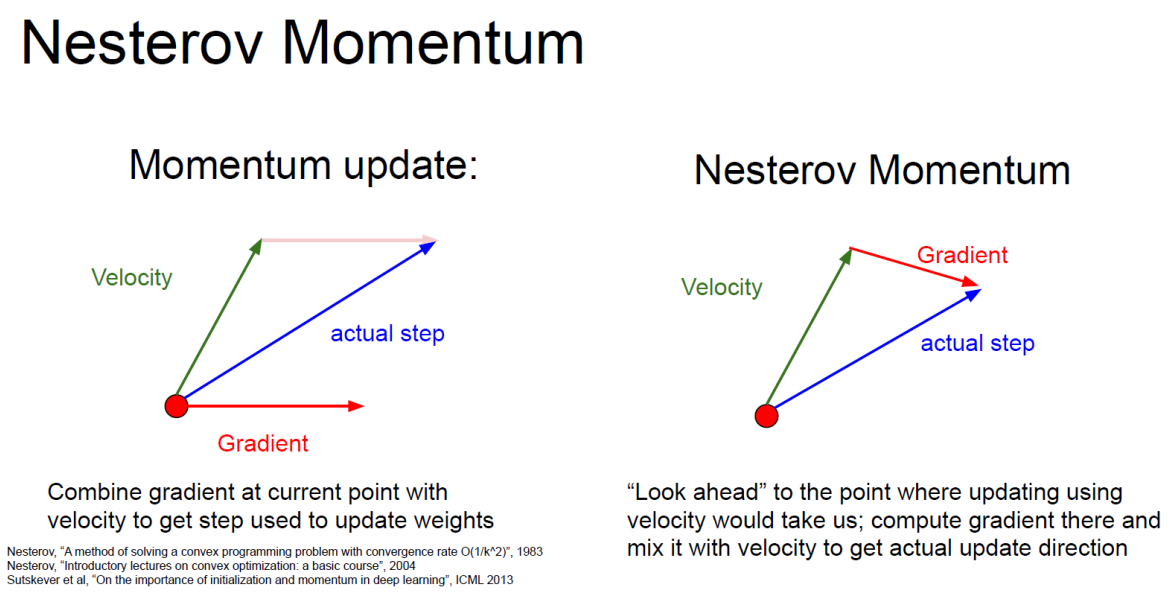
如上，在 v_i 的方向上步进，而不是在梯度方向上步进. 使用了friction constant ρ.

这个例子可以想象成一个球在滚下山的时候有速度，即使是鞍点也能滚过去. 梯度是一个方向，但是带有 Momentum 的方法在调整方向的基础上，保持了速度.



Nesterov Momentum

Velocity 的初始化为0没关系，这不是一个类似 w 的.



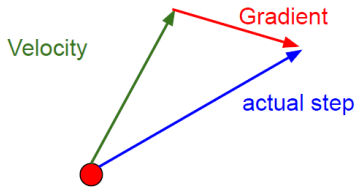
- 左边是带有Momentum的梯度更新，Velocity(上一层来的) 和 Gradient(红点的) 叠加.
- 右边是Nesterov Momentum，注意区别是 Gradient 是按照 Velocity 步进之后计算的，然后和上一层的 Velocity 叠加，再在红点执行这个叠加后的步进.

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$



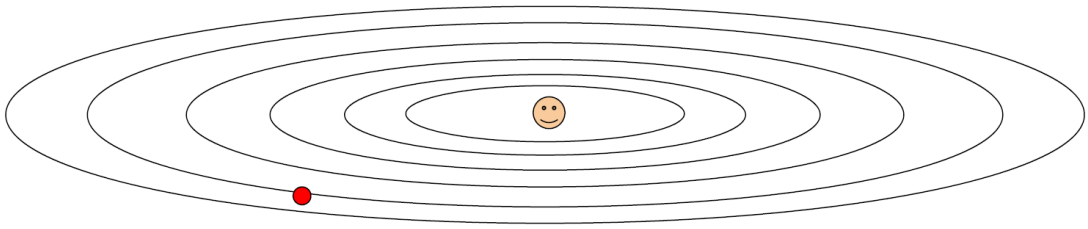
"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

https://cs231n.github.io/neural-networks-3/

AdaGrad

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

Progress along “steep” directions is damped;
progress along “flat” directions is accelerated

- Q2: What happens to the step size over long time?

  Decays to zero.

AdaGrad 凸函数时表现很好，但非凸情况不行.

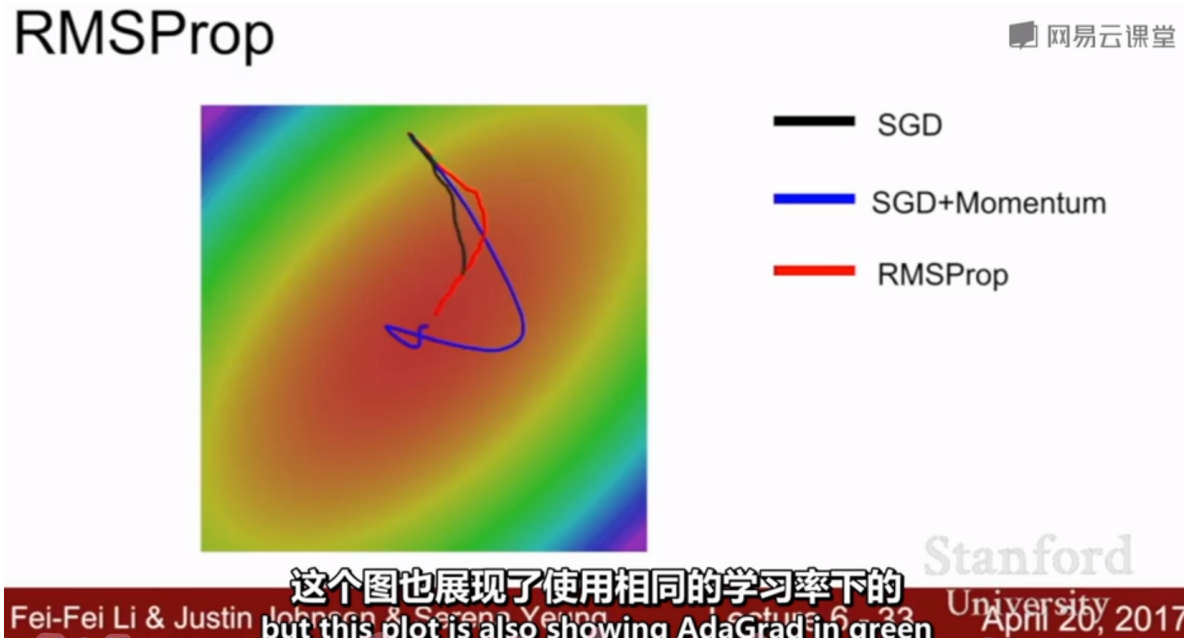
RMSProp: “Leaky AdaGrad”

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



在训练神经网络时我们倾向于不使用 `AdaGrad.`