

Lecture 1: Image Classification

The image classification task

Challenges

An image classifier

Machine Learning: Data-Driven Approach

First classifier: Nearest Neighbor

Hyperparameters

Lecture 2: Loss Functions and Optimization

loss function

Regularization

Softmax classifier (Multinomial Logistic Regression)

交叉熵(cross-entropy)

K-L散度 (相对熵) = 信息熵 - 交叉熵

Optimization

Stochastic Gradient Descent (SGD)

Lecture 3: Neural Networks

Activation function

Backpropagation

Lecture 4: Convolutional Neural Networks

Convolution Layer

convolutional kernel

1x1 conv kernel

Pooling layer

FC layers

Summary

Lecture 5: Training Neural Networks

Activation function

Sigmoid() and tanh()

ReLU()

Leaky ReLU() and ELU()

Data Preprocessing

standardization

PCA主成分分析

Weight Initialization

Xavier Initialization

Kaiming / MSRA Initialization

Batch Normalization

Train

Test

Optimization

SGD

SGD + Momentum

AdaGrad and RMSProp

Adam

First-Order and Second-Order Optimization

Learning rate schedules

Overfitting

Early Stopping

Model Ensembles

Regularization

Drop Connect

Data Augmentation

Choosing Hyperparameters

Transfer learning

Lecture 6: CNNs in Practice

How to stack them (The power of small filters)
How to compute them
 im2col
 FFT
Lecture 7: CNN Architectures and Recurrent Neural Networks
 CNN Architectures
 Recurrent Neural Networks
Lecture 8: Semantic Segmentation
 Sliding Window
 Fully Convolution Network (FCN)
 下采样
 上采样
Lecture 9: Object Detection
 多目标检测总体方法
 Two-stage object detector
 RCNN
 Fast R-CNN
 Faster R-CNN
 Single-Stage Object Detectors
 YOLO
 SSD

Lecture 1: Image Classification

The image classification task

Challenges

- Viewpoint variation
- Background Clutter
- Illumination
- Occlusion
- Deformation

An image classifier

```
def classify_image(image):
    # Some magic here?
    return class_label
```

no obvious way to hard-code the algorithm for recognizing classes. 即不能显式编码

Machine Learning: Data-Driven Approach

1. Collect a dataset of images and labels
2. Use Machine Learning algorithms to train a classifier
3. Evaluate the classifier on new images

```
def train(images, labels):
    # Machine learning!
    return model
```

→ Memorize all data and labels

```
def predict(model, test_images):
    # Use model to predict labels
    return test_labels
```

→ Predict the label of the most similar training image

First classifier: Nearest Neighbor

L1 (Manhattan) distance

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

L2 (Euclidean) distance

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

- L1或L2距离衡量两个图片的距离，选择最近的一类
- 效果很差
- L1对坐标轴的旋转比较敏感，L2则不会

K-Nearest Neighbors:

take **majority vote** from K closest points (投票)

K是一个需要人工去指定的超参数

Hyperparameters

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

Better!

train	validation	test
-------	------------	------

validation验证集用于评估用的

Idea #4: Cross-Validation: Split data into **folds**, try each fold as validation and average the results

fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test

Useful for small datasets, but not used too frequently in deep learning

Lecture 2: Loss Functions and Optimization

loss function

A loss function tells how good our current classifier is.

Loss over the dataset is a average of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Multiclass SVM loss:

Given an example (x_i, y_i)
where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

铰链损失函数 (hinge loss) 的思想就是让那些未能正确分类的和正确分类的之间的距离要足够的远，如果相差达到一个阈值 Δ 时，此时这个未正确分类的误差就可以认为是0，否则就要累积计算误差。

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:

$$\begin{aligned} L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\ &= \max(0, 2.2 - (-3.1) + 1) \\ &\quad + \max(0, 2.5 - (-3.1) + 1) \\ &= \max(0, 6.3) + \max(0, 6.6) \\ &= 6.3 + 6.6 \\ &= 12.9 \end{aligned}$$

Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions}} + \lambda R(W) \underbrace{\text{should match training data}}$$

Data loss: Model predictions should match training data

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

正则化防止过拟合。

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:

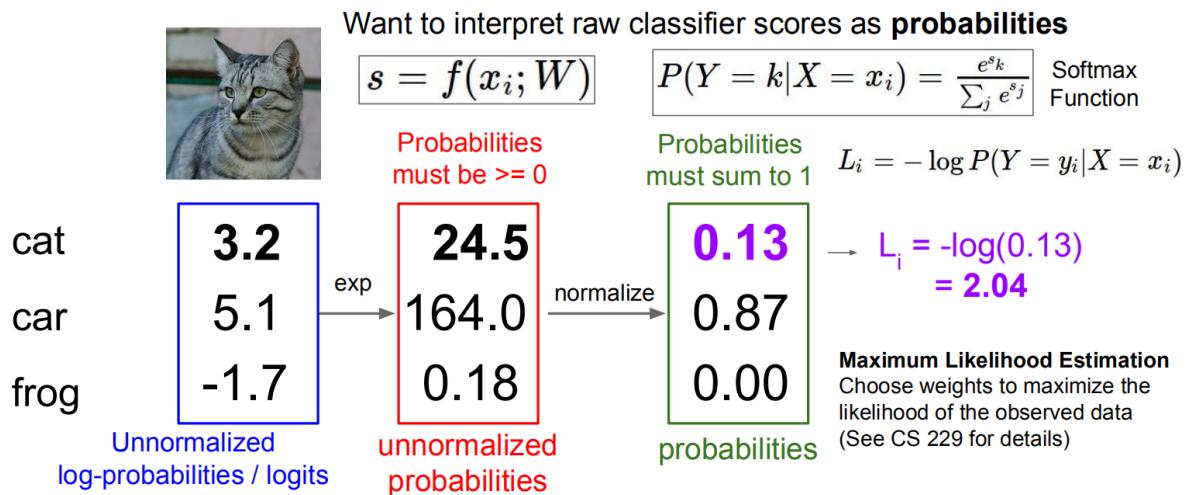
Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

L2正则化较L1更可以将权值分散到各个维度，防止某个权值过大，造成过拟合。

Softmax classifier (Multinomial Logistic Regression)



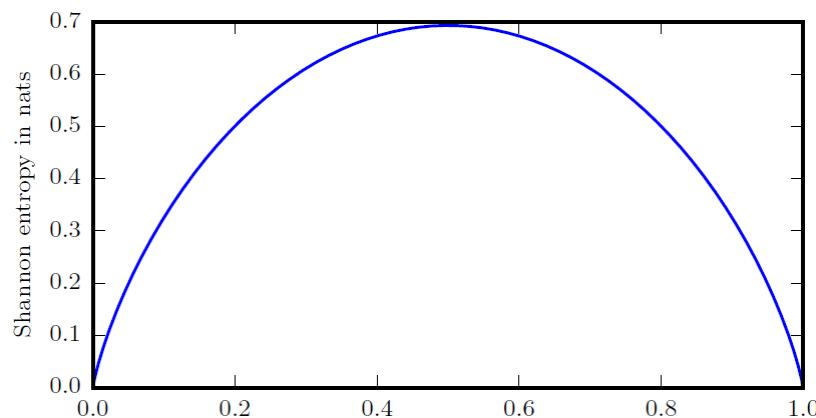
对于多分类问题用 *softmax* 输出对应的概率，再根据这个概率值用交叉熵损失函数计算loss。

交叉熵(cross-entropy)

$$L = - \sum_{c=1}^M y_c \log(p_c)$$

其中， L 为loss， y_c 为标签， p_c 为 *softmax* 预测的概率分布。

而我们希望在训练数据上模型学到的分布 (p_c) 和真实数据的分布 (y_c) 越接近越好



交叉熵损失函数常用于分类问题中，下面以图像分类问题来举例说明。

为了计算网络的loss，模型的输出要确保归一化到0到1之间，二分类问题通常使用sigmoid函数来进行归一化，多分类问题通常使用softmax函数来归一化。

假设我们需要对数字1, 2, 3进行分类，它们的label依次为：

[1,0,0], [0,1,0], [0,0,1]

当输入的图像为数字1时，它的输出和label为：

[0.3,0.4,0.3], [1,0,0]

接下来我们就可以利用交叉熵计算网络的 $loss = -(1 * \log(0.3) + 0 + 0) = 1.20$

随着训练次数的增加，模型的参数得到优化，这时的输出变为：[0.8,0.1,0.1]

则 $loss = -(1 * \log(0.8) + 0 + 0) = 0.22$

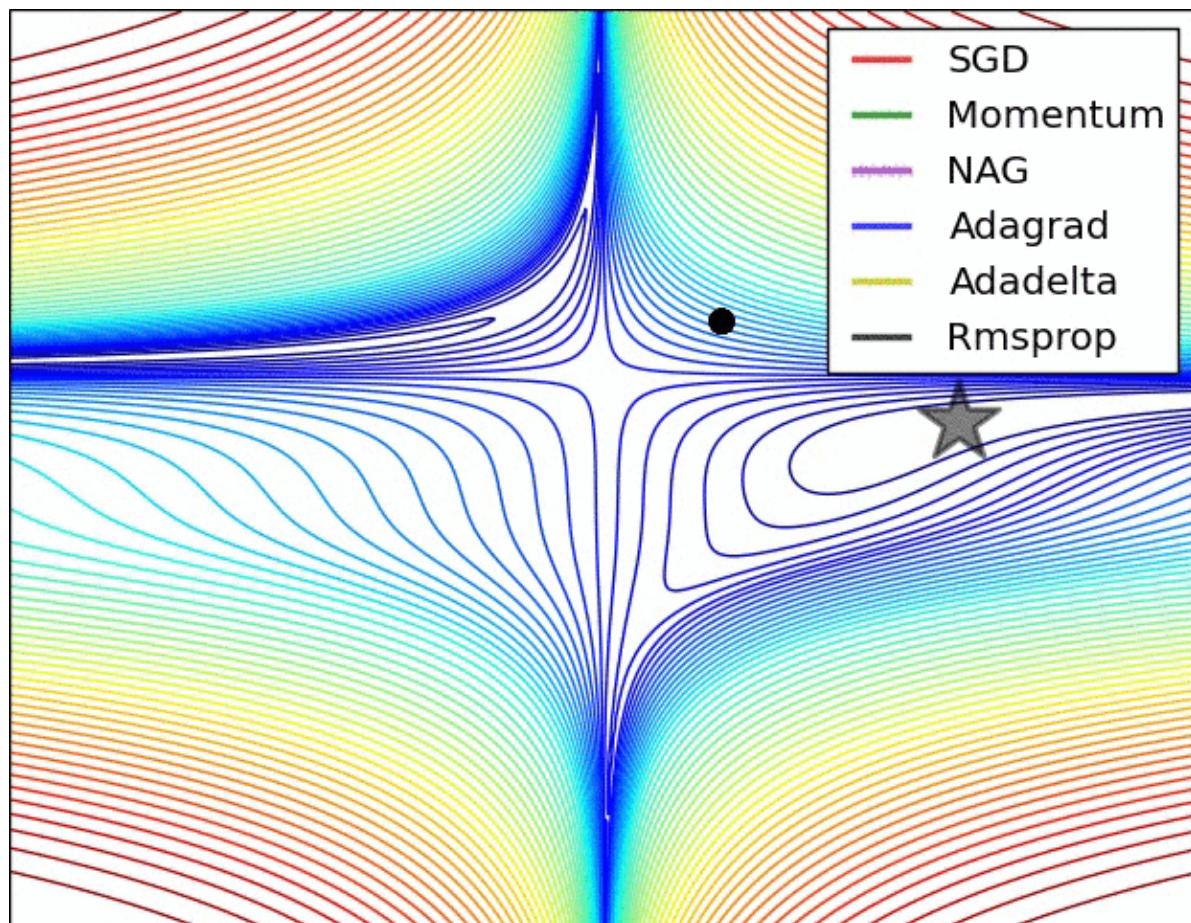
可以发现loss由1.20减小为0.22，而判断输入图像为数字1的概率由原本的0.3增加为0.8，说明训练得到的概率分布越来越接近真实的分布，这样就大大的提高了预测的准确性。

K-L散度（相对熵）= 信息熵 - 交叉熵

Optimization

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

- **Numerical gradient(数值解):** approximate, slow, easy to write
- **Analytic gradient(解析解):** exact, fast, error-prone



Stochastic Gradient Descent (SGD)

- 所有数据一次全都扔进去，根据损失函数的和求梯度更新一次（计算成本太高）
- 一个个数据扔进去，每一次都求一次梯度更新权重（但是可能更新会振荡）
- 折中：选择mini-batch，每次投入一个batch的数据（加快了收敛速度，同时节省了内存）

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive
when N is large!

Approximate sum
using a **minibatch** of
examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent

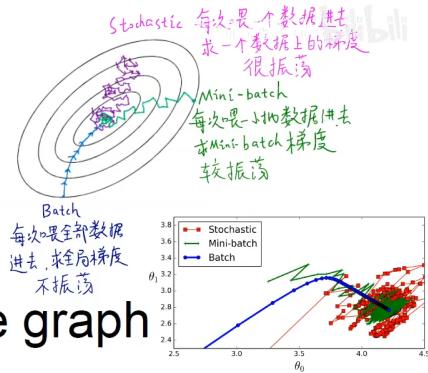
while True:
    data_batch = sample_training_data(data, 256) # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad # perform parameter update
```

Where we are now...

Mini-batch SGD

Loop:

1. **Sample a batch of data**
2. **Forward prop it through the graph (network), get loss**
3. **Backprop to calculate the gradients**
4. **Update the parameters using the gradient**



Lecture 3: Neural Networks

Activation function

Neural networks: why is max operator important?

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

The function $\max(0, z)$ is called the **activation function**.

Q: What if we try to build a neural network without one?

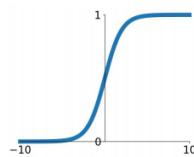
$$f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

A: We end up with a linear classifier again!

激活函数的作用：非线性激活函数是用来加入非线性因素的，因为线性模型的表达能力不够。线性的表达能力太有限了，无论叠加多少层最后仍然是线性的，与单层网络无异

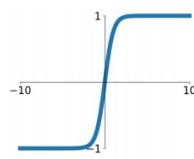
Sigmoid

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



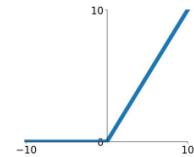
tanh

$$\tanh(x)$$



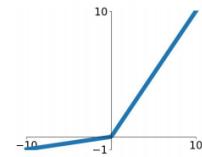
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

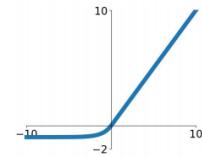


Maxout

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

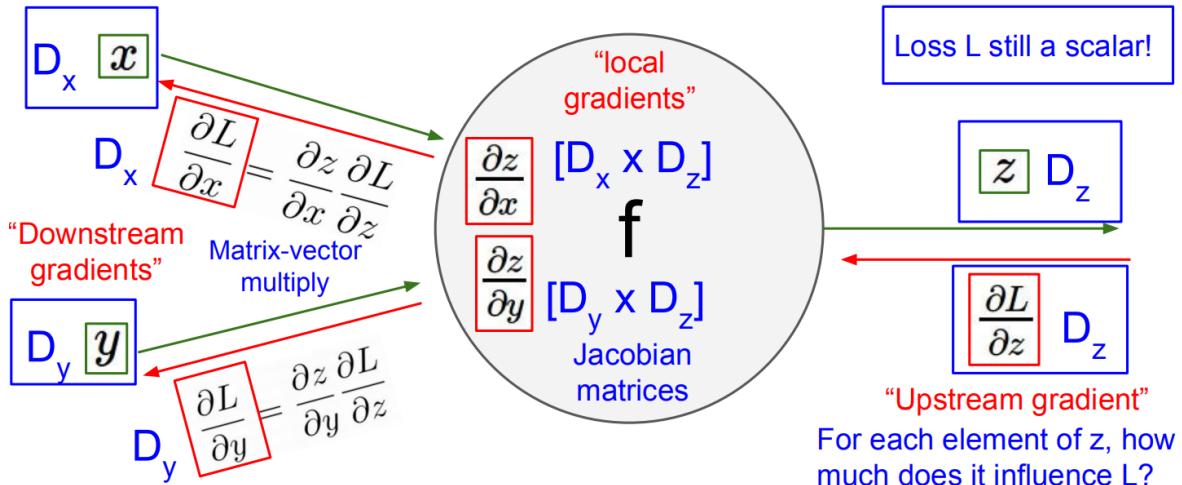
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Backpropagation

Backprop with Vectors



这里我希望给出矩阵形式的 FP 和 BP 算法的公式，因为在后续编程时遇到了一定的困难，特意去搜索了 BP 算法的矩阵形式，非常便于编程的实现。

设输入层为第 0 层，输出层为第 l 层，隐藏层分别为第 i 层 ($i = 1, 2, \dots, l - 1$)，设第 l 层权重矩阵为 W_l ，数据到第 l 层经过权重计算的输出结果为 α_l ， O_l 为第 l 层经过激活函数 f 后的输出，则有 FP 算法的计算公式：

$$\alpha_l = W_l O_{l-1}$$

$$O_l = f(\alpha_l)$$

设 y 为标签， η 为学习率，定义损失函数为 $L = \frac{1}{2} \|O_l - y\|^2$ ，则第 l 层更新算

法为：

$$W_l = W_l - \eta \frac{\partial L}{\partial W_l} = W_l - \eta \xi_l O_{l-1}^T$$

其中：

$$\frac{\partial L}{\partial W_l} = \frac{\partial L}{\partial \alpha_l} \frac{\partial \alpha_l}{\partial W_l} = \frac{\partial L}{\partial \alpha_l} \frac{\partial W_l O_{l-1}}{\partial W_l} = \xi_l O_{l-1}^T$$

ξ_l 是第 l 层即输出层的误差向量，该误差项逐步往后传播，其求解方式为：

$$\xi_l = \frac{\partial L}{\partial \alpha_l} = \frac{\partial \left(\frac{1}{2} \|O_l - y\|^2 \right)}{\partial \alpha_l} = (O_l - y) \circ f'(\alpha_l)$$

在确定后一层的误差向量跟权值矩阵后，我们就能求出当前层的误差向量，以第 $l - 1$ 层为例，有：

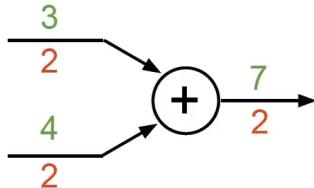
$$\xi_{l-1} = \frac{\partial L}{\partial \alpha_l} \frac{\partial \alpha_l}{\partial O_{l-1}} \frac{\partial O_{l-1}}{\partial \alpha_{l-1}} = (W_l^T \xi_l) \circ f'(\alpha_{l-1})$$

至此，得到了每一层的误差向量求解的迭代公式，即可对每层的权重进行更新。

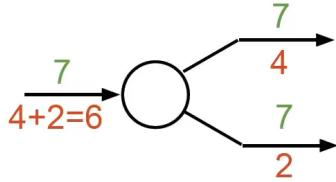
Patterns in gradient flow

加法：梯度均分

add gate: gradient distributor

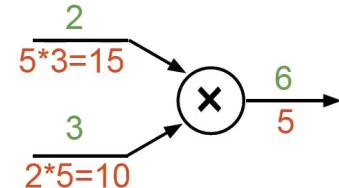


copy gate: gradient adder



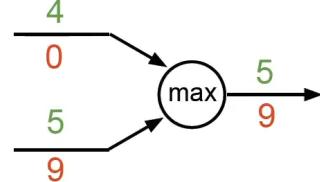
乘法：梯度交换

mul gate: "swap multiplier"

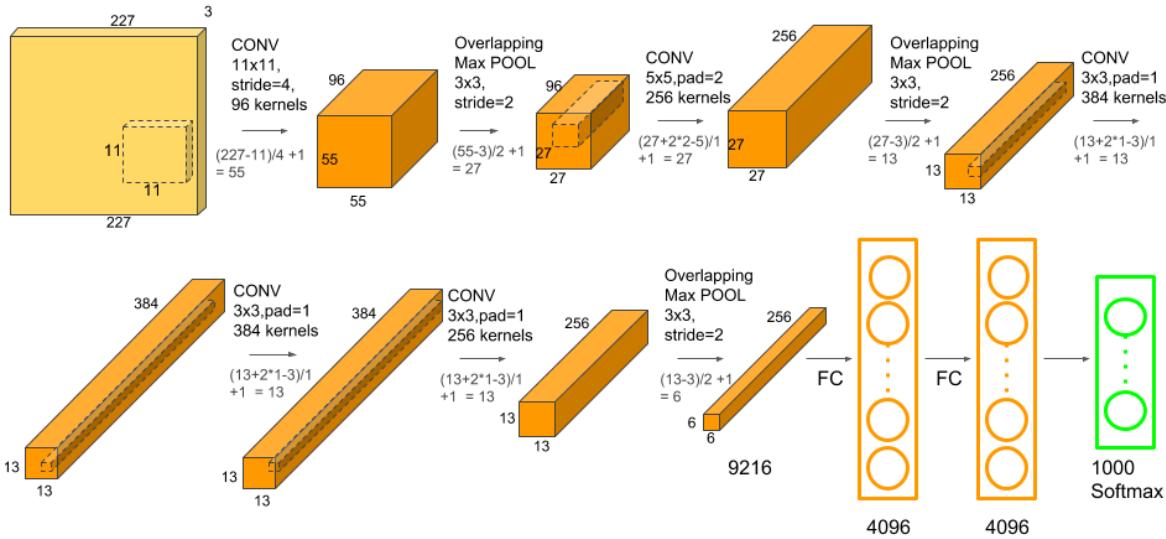


max : 梯度路由 (只走最大所在路)

max gate: gradient router

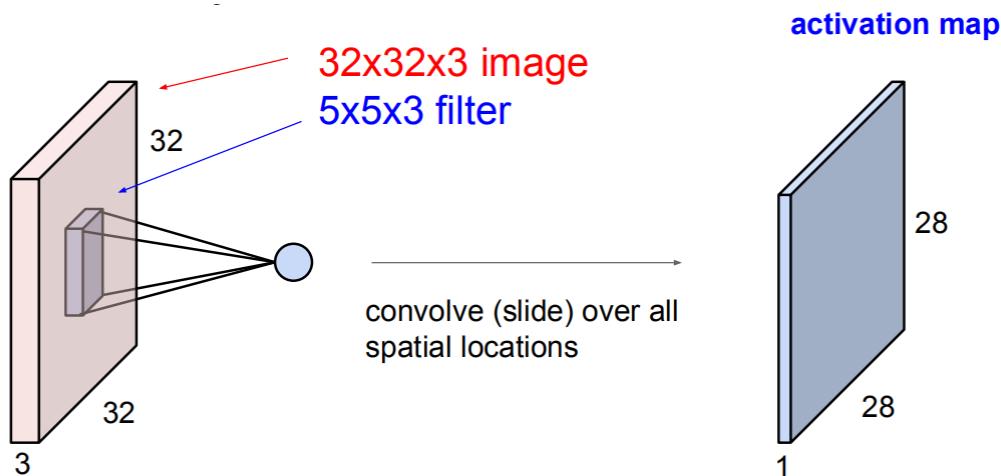


Lecture 4: Convolutional Neural Networks



Convolution Layer

convolutional kernel



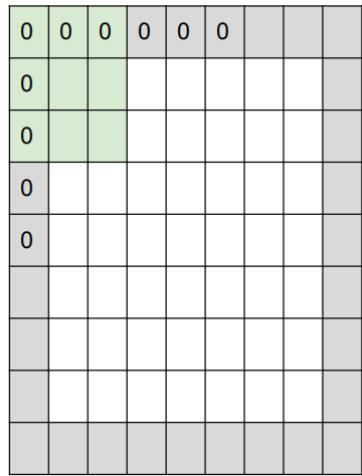
Feature Map (activation map)是输入图像经过神经网络卷积产生的结果

主要选取的是卷积核的权重，大小（ 3×3 、 5×5 等），滑动的步长stride。

上图中卷积核大小 $5 \times 5 \times 3$ ，步长为1，因此宽度 $32 - 5 + 1 = 28$

Feature Map的大小 = $(N - F) / \text{stride} + 1$

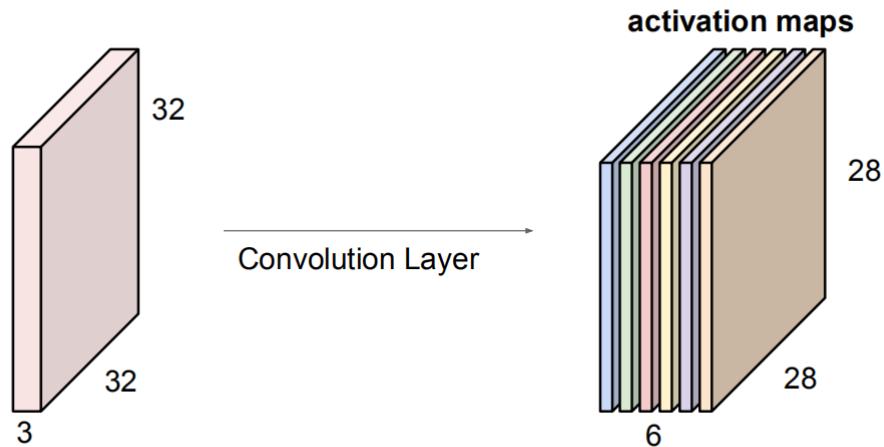
如果padding一圈（补0）： Feature Map的大小 = $(N+2P-F) / \text{stride} + 1$



即使输入时3通道，但是经过模板计算以后，每个模板卷积计算得到的是“一个”数。

下一层的通道数由上一层所用的卷积核的个数决定。

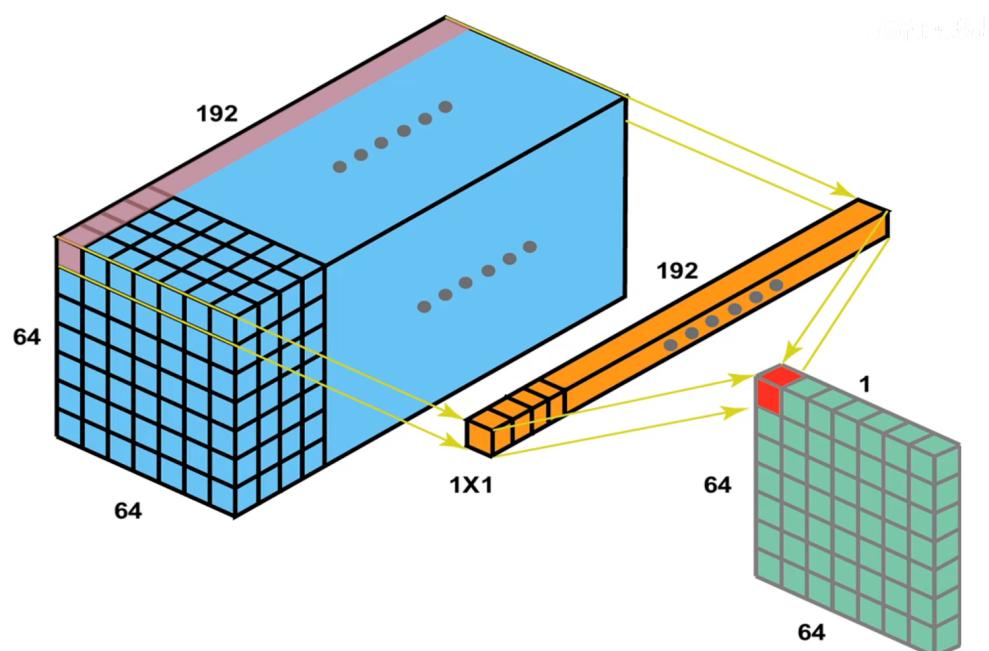
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



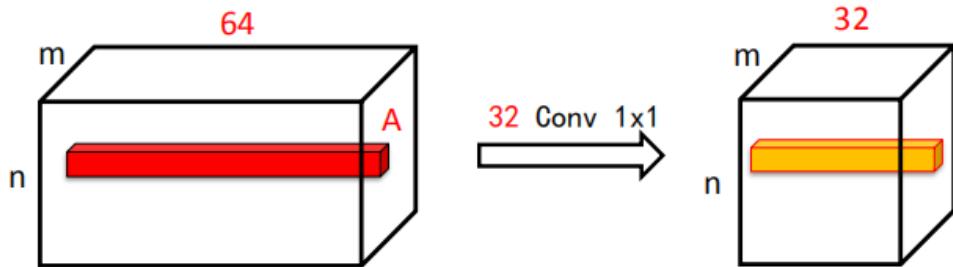
We stack these up to get a “new image” of size 28x28x6!

使用不同的卷积核得到多通道的activation map，而每个卷积核的权重矩阵一般是随机初始化的，因此不同的卷积核可以提取到不同的特征。

1x1 conv kernel



- 降维或升维
- 跨通道信息交融（因为把原图多个通道的值相加）
- 减少参数数量
- 增加模型深度，提高非线性表示能力
- 利用 1×1 卷积进行非线性压缩通常不会损失信息（原向量一般是非常稀疏向量，很多位置的响应都是0）

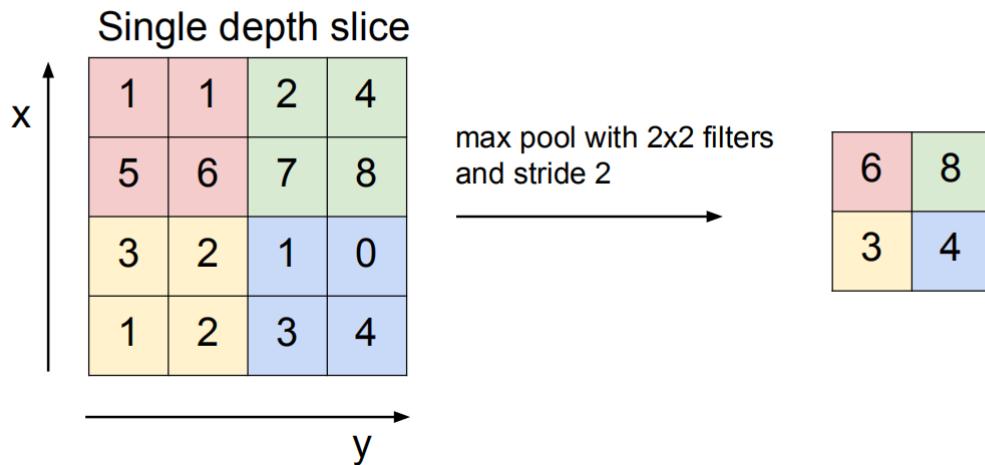


压缩的尺度与选择的 1×1 卷积核的个数有关。

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently
- 是一种下采样

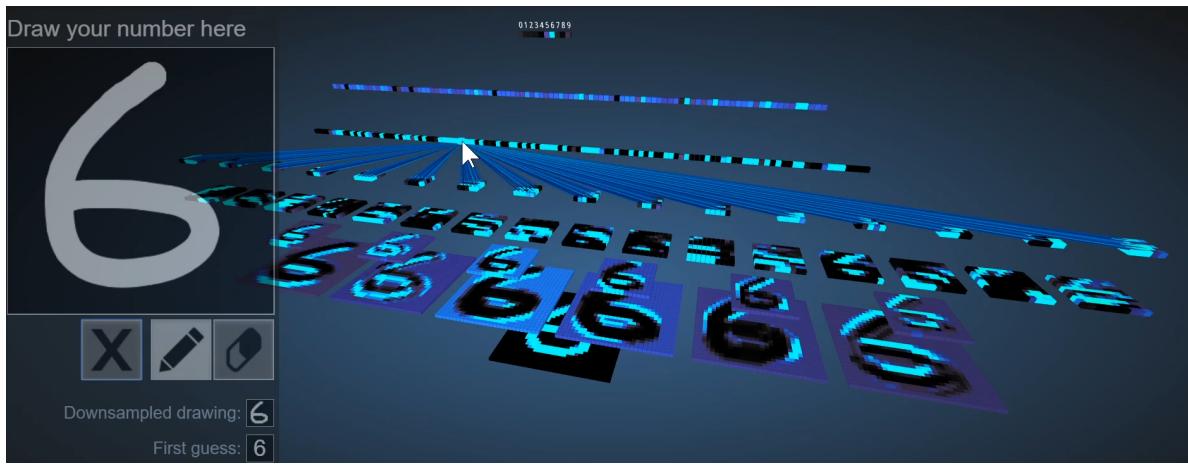
MAX POOLING



max pooling 用的比较多，相当于选出激活最大，最显著的。

FC layers

其实就是之前用过的MLP，不过特别注意一点是，将最后一个pooling完的layer拉成一维向量，然后作为全连接层的输入层。



(FC的隐藏层与最后一个layer的每个元素都有连接)

Summary

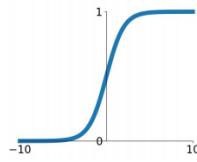
- Trend towards smaller filters and deeper architectures.
- Trend towards getting rid of POOL/FC layers (just CONV) (不用pool和fc)

Lecture 5: Training Neural Networks

Activation function

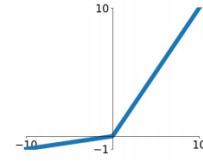
Sigmoid

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



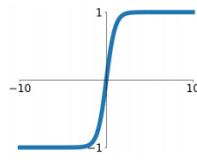
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

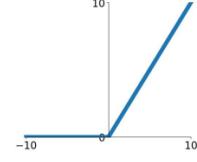


Maxout

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

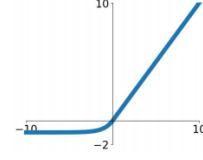
ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid and tanh activation functions have saturation and gradient消失的问题。

Leaky ReLU and ELU are designed to improve the situation where ReLU's gradient is zero for x < 0.

Sigmoid() and tanh()

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

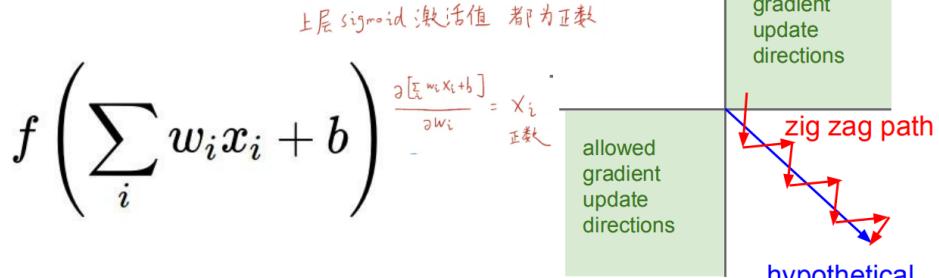
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

This function squashes numbers to range [0,1].

Three problems:

- 1、Saturated neurons “kill” the gradients (x过大或过小造成函数饱和，使得梯度接近0)
If all the gradients flowing back will be zero and weights will never change.
- 2、Sigmoid outputs are all positive and not zero-centered.

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



What can we say about the gradients on w ?

Always all positive or all negative :(

(For a single element! Minibatches help)

如果上一层的输出全正（或全负），则对 w_i 的偏导数符号相同，根据更新的法则所有的权重会同时增大或减小，会出现上图所示的Z型优化路径。

- 3、`exp()` is a bit compute expensive

tanh() 相较Sigmoid()解决了问题2，但依然存在梯度消失。

ReLU()

$$f(x) = \max(0, x)$$

- 一定不会发生饱和以及梯度消失，不消耗什么计算资源，收敛很快。
- 输出不关于0对称， $x < 0$ 时梯度为0，相当于有一些神经元永远不会被更新。
 - 初始不良
 - 学习率太大

Leaky ReLU() and ELU()

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

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

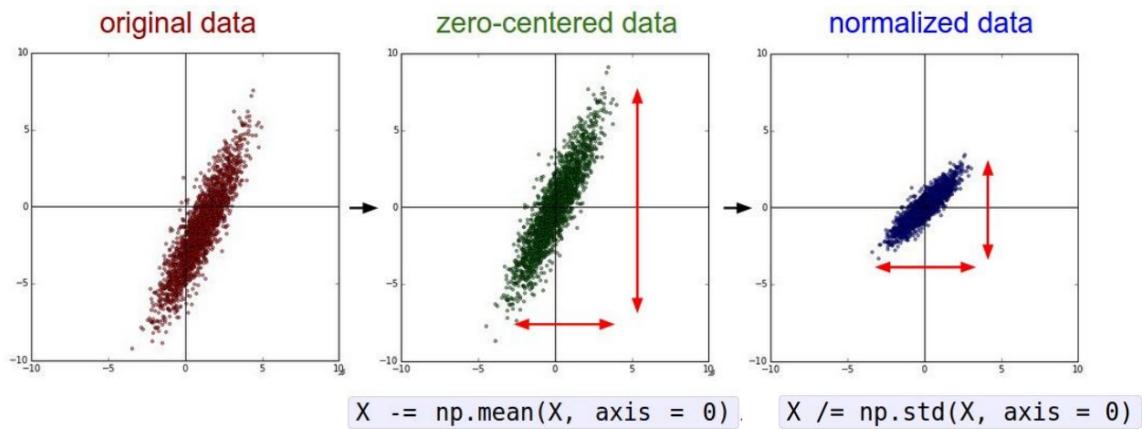
- Computation requires `exp()`
(Alpha default = 1)

ELU()改善了Leaky ReLU()关于零点不对称的问题。

- Use **ReLU**. Be careful with your learning rates
- Try out **Leaky ReLU / Maxout / ELU / SELU**
 - To squeeze out some marginal gains
- Don't use **sigmoid** or **tanh**

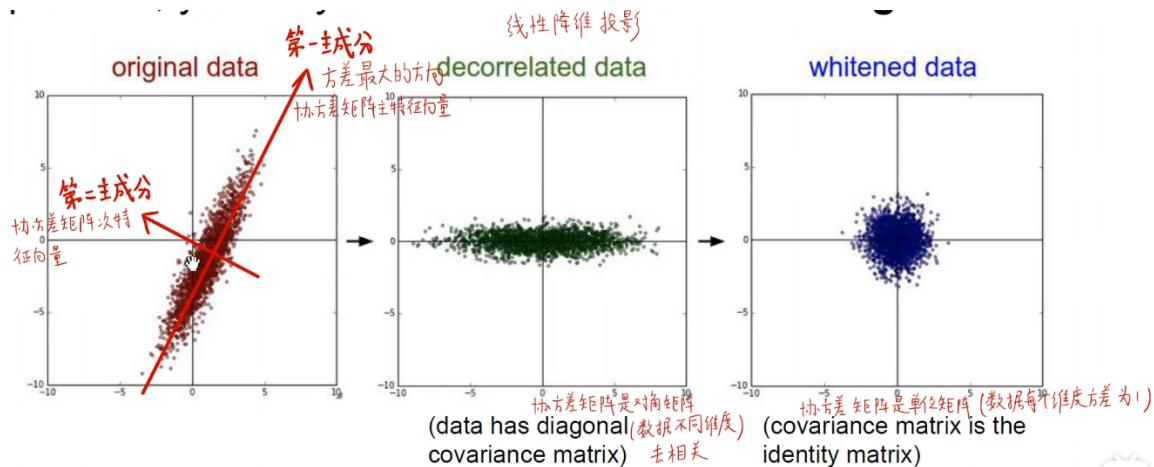
Data Preprocessing

standardization



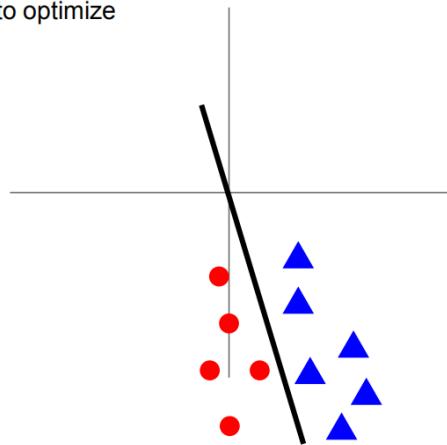
处理后的值将近似服从标准正态分布

PCA主成分分析

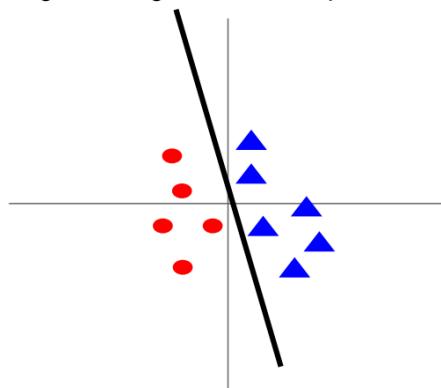


- 第一主成分对应方差变化最大的方向，即协方差矩阵主特征向量的方向；
- 第二主成分对应方差变化第二大的方向，即协方差矩阵次特征向量的方向；
- 线性降维以后协方差矩阵是对角矩阵，去相关。
- 在主成分上除以标准差，转换成白化数据，协方差矩阵是单位矩阵（每个维度方差为1）

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



标准化处理后，损失函数对w的微小改变不那么敏感，更容易优化。

PCA和白化用的其实没有标准化多。

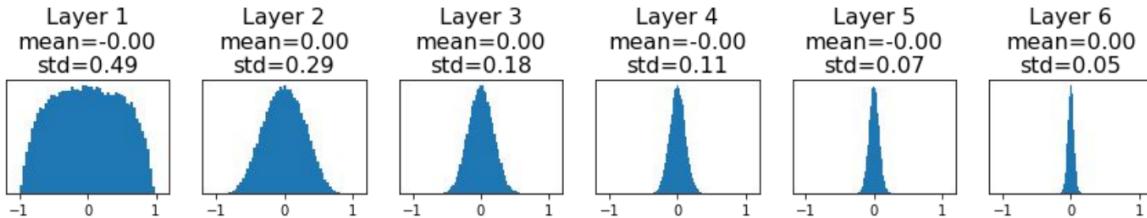
Weight Initialization

- 多层神经网络不能将权重初始化为同一个数，否则无法打破“对称性” (symmetry)
- 权重应该选大还是选小？

a.g. 取6层，4096个神经元， $\tanh()$ 为激活函数，有：

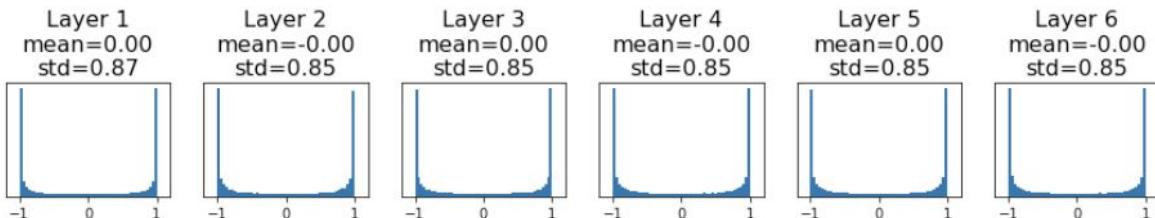
$$\frac{\partial f}{\partial \omega_i} = f' \times x_i$$

```
W = 0.01 * np.random.randn(Din, Dout)
```



到后面层输入 x_i 的值过小（集中在0）， $\frac{\partial f}{\partial \omega_i} = 0$ ，会造成梯度消失

```
W = 0.05 * np.random.randn(Din, Dout)
```



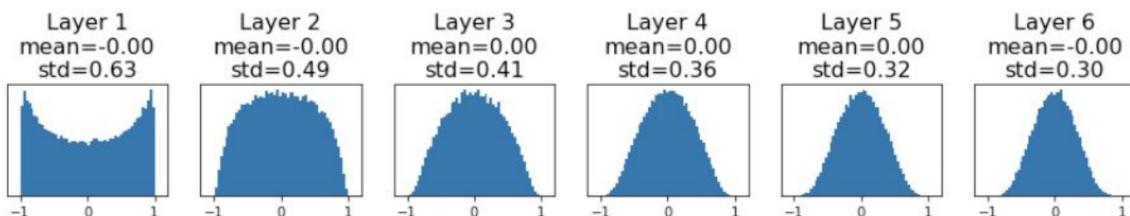
每一层的输出都集中在饱和区， $f' = 0$ ， $\frac{\partial f}{\partial \omega_i} = 0$ ，会造成梯度消失

Xavier Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is filter_size² * input_channels



根据输入和输出的维度，可以自适应的调整权重的初始化幅度。

输入维度越多，权重初始化时的幅度应该越小。（对于CNN，Din = 卷积核大小）

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{Din})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din})$$

$$= D_{in} \text{Var}(x_i w_i)$$

$$= D_{in} \text{Var}(x_i) \text{Var}(w_i)$$

[Assume all x_i, w_i are iid]

So, $\text{Var}(y) = \text{Var}(x_i)$ only when $\text{Var}(w_i) = 1/D_{in}$

Kaiming / MSRA Initialization

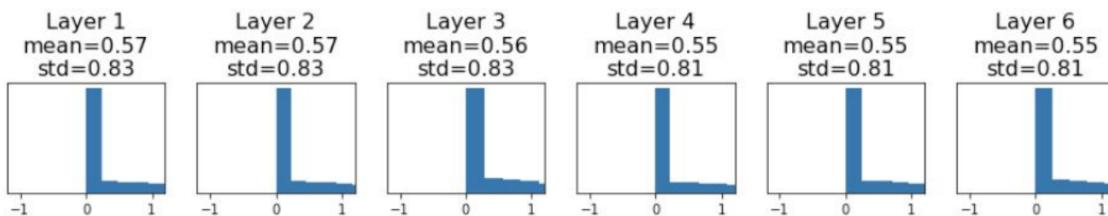
但是如果用ReLU()激活函数，也可能出现梯度消失的问题。（后面层 x_i 集中在0）

```
dims = [4096] * 7  ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

欲使 $\text{Var}(Y_i) = \text{Var}(X_j)$:

若 w_{ij} 服从正态分布，则 $w_{ij} \sim \text{Normal}(0, \frac{2}{d})$



Batch Normalization

Train

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

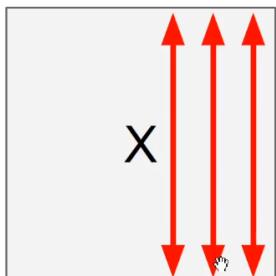
this is a vanilla differentiable function...

Input: $x : N \times D$ 一个batch中有N个数据 每个数据D维

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

求出batch中每一个特征维度的均值
Per-channel mean, 得到D个均值
shape is D

N



D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

求出batch中每一个特征维度的方差, 得到D个
Per-channel var, 方差
shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

对batch中的每个数据进行批归一化
Normalized x,
Shape is $N \times D$

这三步就是normalization 工序

Problem: What if zero-mean, unit variance is too hard of a constraint?

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

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

但是公式的后面还有一个反向操作, 将 normalize 后的数据再扩展和平移. 原来这是为了让神经网络自己去学着使用和修改这个扩展参数 gamma, 和 平移参数 beta, 这样神经网络就能自己慢慢琢磨出前面的 normalization 操作到底有没有起到优化的作用, 如果没有起到作用, 我就使用 gamma 和 beta 来抵消一些 normalization 的操作.

我们需要将训练阶段的总均值和总方差保存下来。

Test

Input: $x : N \times D$

$\mu_j = \text{(Running) average of values seen during training}$

Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta : D$

$\sigma_j^2 = \text{(Running) average of values seen during training}$

Per-channel var, shape is D

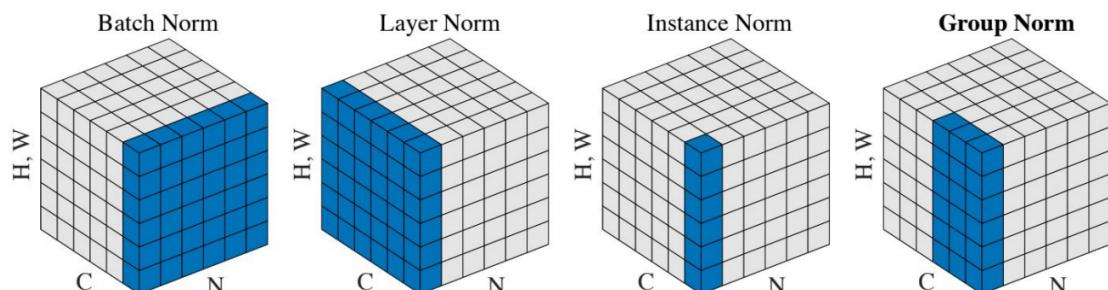
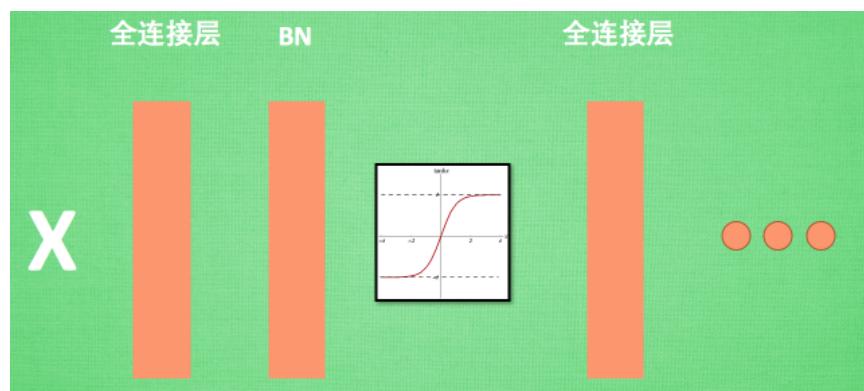
During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

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

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

用训练时的总均值代替mini-batch的均值, 训练时的总方差代替mini-batch的方差。

Usually inserted after Fully Connected or Convolutional layers, and **before nonlinearity**.

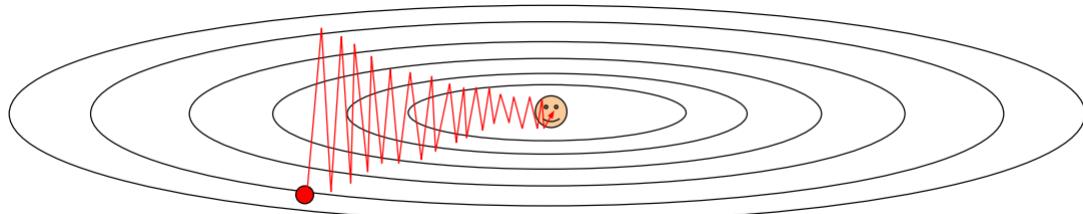


Optimization

SGD

Problems:

- 在梯度较大的方向上产生振荡，且不能单纯通过减小学习率解决。



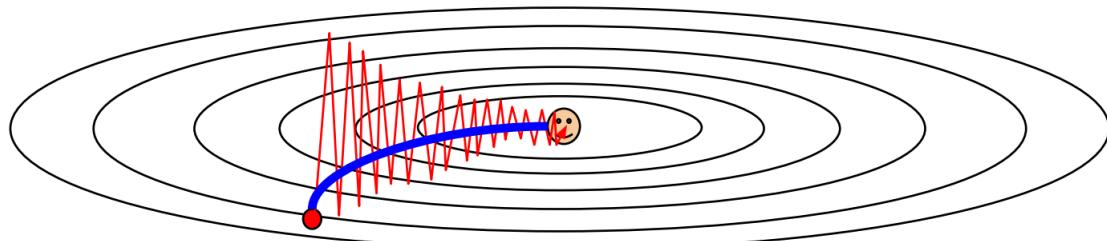
- 容易陷入局部最优点或鞍点 (Saddle points, 在高维空间中更普遍)

Local Minima Saddle points



- 因为梯度计算来源于mini-batch, 所以可能会包含很多噪声

SGD + Momentum



SGD+Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$

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

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
```

You may see SGD+Momentum formulated different ways,
but they are equivalent - give same sequence of x

直观理解就是，下一次优化的方向不仅与当前时刻的梯度有关，还与之前的优化方向和速度有关。

ρ 类似于提供了摩擦力，使得之前的速度得到衰减。一般 $\rho = 0$ 取 0.9 或 0.99。

$\rho = 0$, 与 SGD 相同； $\rho = 1$, 过去的速度完全不衰减。

但是 Momentum 方法如果动量过大容易冲过头。

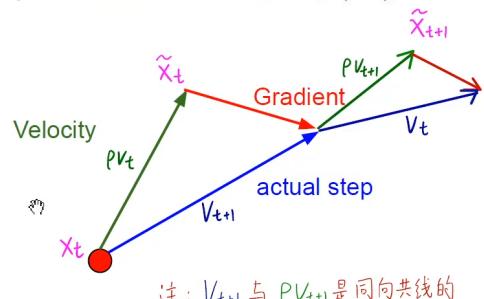
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

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

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

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



"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

(对于上图, \tilde{x}_t 是之后更新的基准点, 从公式也可以看出。之后的迭代与 x_t 本身无关)

(先根据前一时刻 t 的速度计算梯度, 然后退回到 x_t , 从 x_t 开始更新)

Nesterov Momentum 相当于不是计算的当前速度的梯度再与前一时刻的速度进行矢量和, 而是直接根据前一时刻的速度计算梯度进行矢量和, 得到最终的优化方向。相当于提前看一步, 提早感知坡底。

AdaGrad and RMSProp

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)
```

AdaGrad 增加惩罚项, 沿“陡峭”方向前进减弱, 沿“平坦”方向前进加速; 随着训练的进行分母越来越大, 后续更新几乎停止。

RMSProp 在AdaGrad的基础上增加了一个衰减因子, 考虑保留之前多久的累积项, 防止更新停止。

Adam

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx 第一动量 Momentum
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx 第二动量
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)) AdaGrad / RMSProp
```

Adam同时结合了两种动量方式的优势, 但是刚开始的几轮中一次和二次动量值都比较小。进行修正

```

first_moment = 0      } 初始化为0。在训练起始阶段难以预热进入工作状态。
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))

```

Momentum

使得刚开始训练时两个动量有较高
Bias correction 初始值

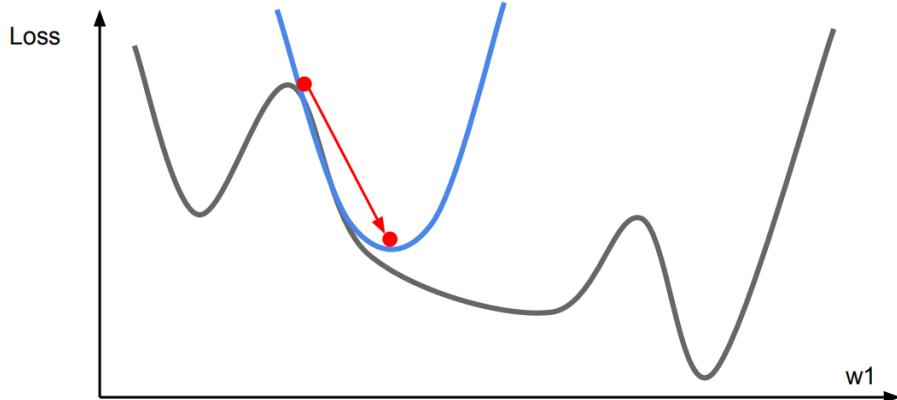
AdaGrad / RMSProp

First-Order and Second-Order Optimization

以上提到的所有用梯度下降优化的方法都是一阶方法。

对于二阶方法：（二阶方法不需要设置学习率）

- (1) Use gradient and Hessian to form quadratic approximation
- (2) Step to the minima of the approximation



second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

梯度下降是一阶收敛，只考虑了当前坡度最大的方向；

牛顿法考虑走了一步之后坡度是否会变得更大。

但是Hessian矩阵要计算二阶导，还要计算逆矩阵，计算量太大。

另外有高斯牛顿法，用 Jacobian 矩阵 $J^T J$ 代替 Hessian 矩阵。

可以得到如下方程组：

$$\boxed{\mathbf{J}(\mathbf{x})^T \mathbf{J}(\mathbf{x})} \Delta \mathbf{x} = \boxed{-\mathbf{J}(\mathbf{x})^T f(\mathbf{x})}. \quad (6.21)$$

注意，我们要求解的变量是 $\Delta \mathbf{x}$ ，因此这是一个线性方程组，我们称它为增量方程，也可以称为高斯牛顿方程 (Gauss Newton equations) 或者正规方程 (Normal equations)。我们把左边的系数定义为 \mathbf{H} ，右边定义为 \mathbf{g} ，那么上式变为：

$$\mathbf{H} \Delta \mathbf{x} = \mathbf{g}. \quad (6.22)$$

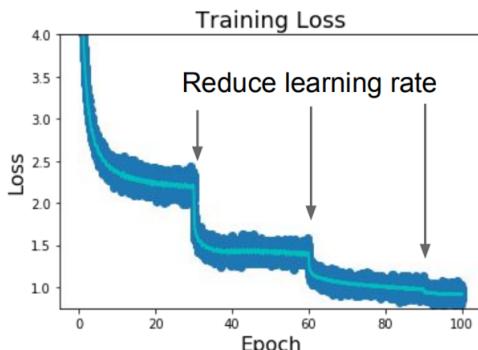
这里把左侧记作 \mathbf{H} 是有意义的。对比牛顿法可见，Gauss-Newton 用 $\mathbf{J}^T \mathbf{J}$ 作为牛顿法中二阶 Hessian 矩阵的近似，从而省略了计算 \mathbf{H} 的过程。求解增量方程是整个优化问题的核心所在。如果我们能够顺利解出该方程，那么 Gauss-Newton 的算法步骤可以写成：

1. 给定初始值 \mathbf{x}_0 。
2. 对于第 k 次迭代，求出当前的雅可比矩阵 $\mathbf{J}(\mathbf{x}_k)$ 和误差 $f(\mathbf{x}_k)$ 。
3. 求解增量方程： $\mathbf{H} \Delta \mathbf{x}_k = \mathbf{g}$.
→ 得到 \mathbf{H} 和 \mathbf{g}
4. 若 $\Delta \mathbf{x}_k$ 足够小，则停止。否则，令 $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$ ，返回 2.

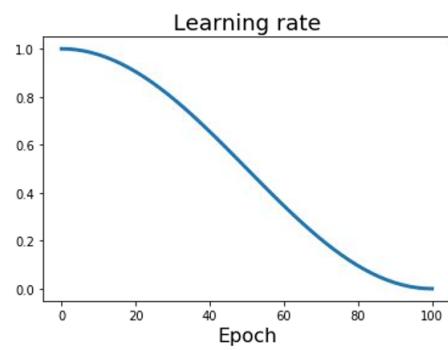
Learning rate schedules

训练初期学习率应比较大，训练后期学习率应比较小。学习率衰减相当于一种二阶参数的方式。

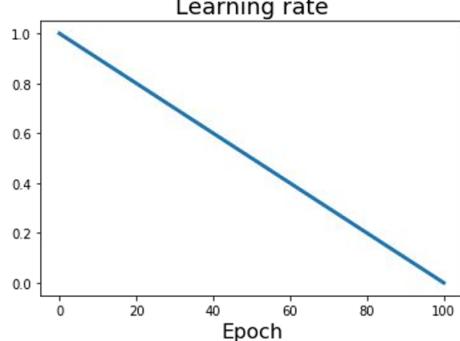
一般的方式是先不用学习率衰减，看看loss曲线的变化情况，判断在哪里开始衰减比较合适。



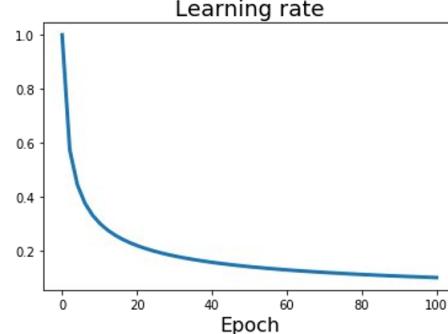
Reduce learning rate at a few fixed points.



Cosine: $\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$



Linear: $\alpha_t = \alpha_0(1 - t/T)$

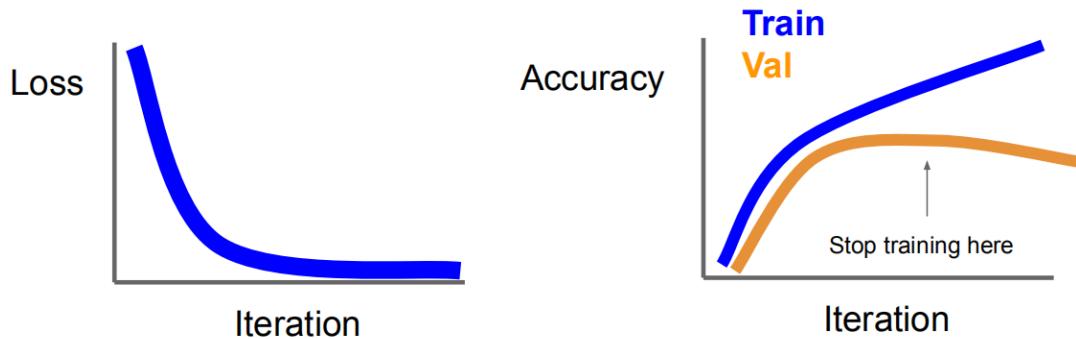


Inverse sqrt: $\alpha_t = \alpha_0 / \sqrt{t}$

Overfitting

在训练集上有比较好的效果（即loss曲线下降比较合适），但验证集准确率很低。

Early Stopping



Model Ensembles

- Train multiple independent models
- At test time average their results
- 其实就是Adaboost的思想

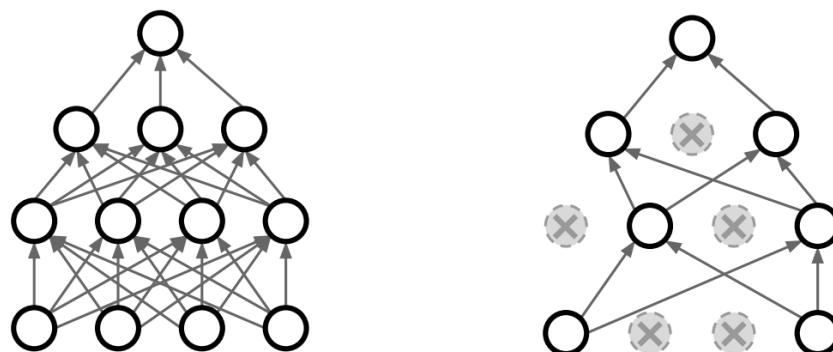
Regularization

Add term to loss: (之前讲过，这里不展开)

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

Drop out:

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



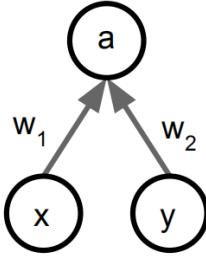
- 打破了特征之间的联合适应性，使得每个特征都能够独当一面
- Dropout每次删去不同神经元都相当于构建一个新的模型，起到了模型集成的作用

通过mask的方式实现神经元的删除（权重置0）

而在test阶段，因为在训练时删去神经元，导致神经元的输出的期望值下降，需要在test时补偿。

a.g. 两个神经元，在dropout中一共4种情况。

Consider a single neuron.



At test time we have: $E[a] = w_1x + w_2y$

$$\begin{aligned} \text{During training we have: } E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

At test time, multiply by dropout probability

但是更common的方法时，我们在训练时提前补偿，训练阶段就不需要进行改变。

More common: “Inverted dropout”

```

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
  
```

test time is unchanged!

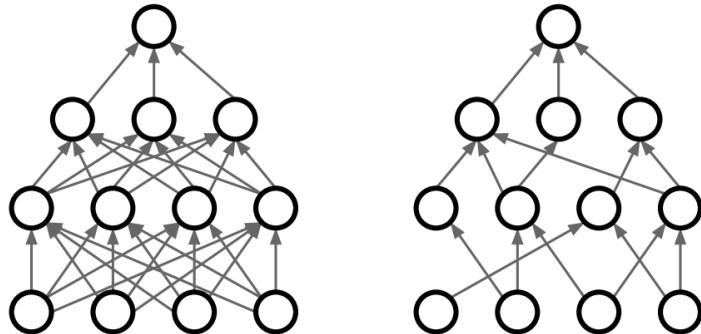
Drop Connect

Training: Drop connections between neurons (set weights to 0)

Testing: Use all the connections

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect



Data Augmentation

通过对原始数据融入先验知识，加工出更多数据的表示，有助于模型判别数据中统计噪声，加强本体特征的学习，减少模型过拟合，提升泛化能力。

单(图像)样本增强主要有翻转、旋转、缩放、裁剪、平移、添加噪声等。

Regularization - In practice

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout
Batch Normalization
Data Augmentation
~~DropConnect~~
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup

- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try cutout and mixup especially for small classification datasets

Choosing Hyperparameters

- **Step 1:** Check initial loss
 - Turn off weight decay, sanity check loss at initialization
- **Step 2:** Overfit a small sample
 - Try to train to 100% training accuracy on a small sample of training data;
- **Step 3:** Find LR that makes loss go down
 - Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations
- **Step 4:** Coarse grid, train for ~1-5 epochs
 - Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.
- **Step 5:** Refine grid, train longer
 - Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay
- **Step 6:** Look at loss and accuracy curves
- **Step 7:** GOTO step 5

Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

Transfer learning



根据自己数据集的体量决定需要对哪一部分网络结构进行重新训练 (一般最多重新训练FC层)

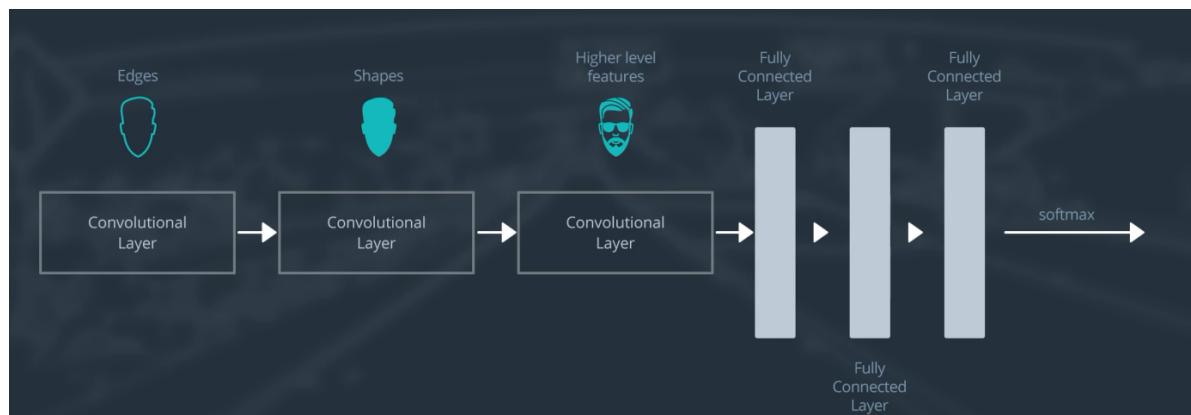
其余部分的结构和权重均保持不变。

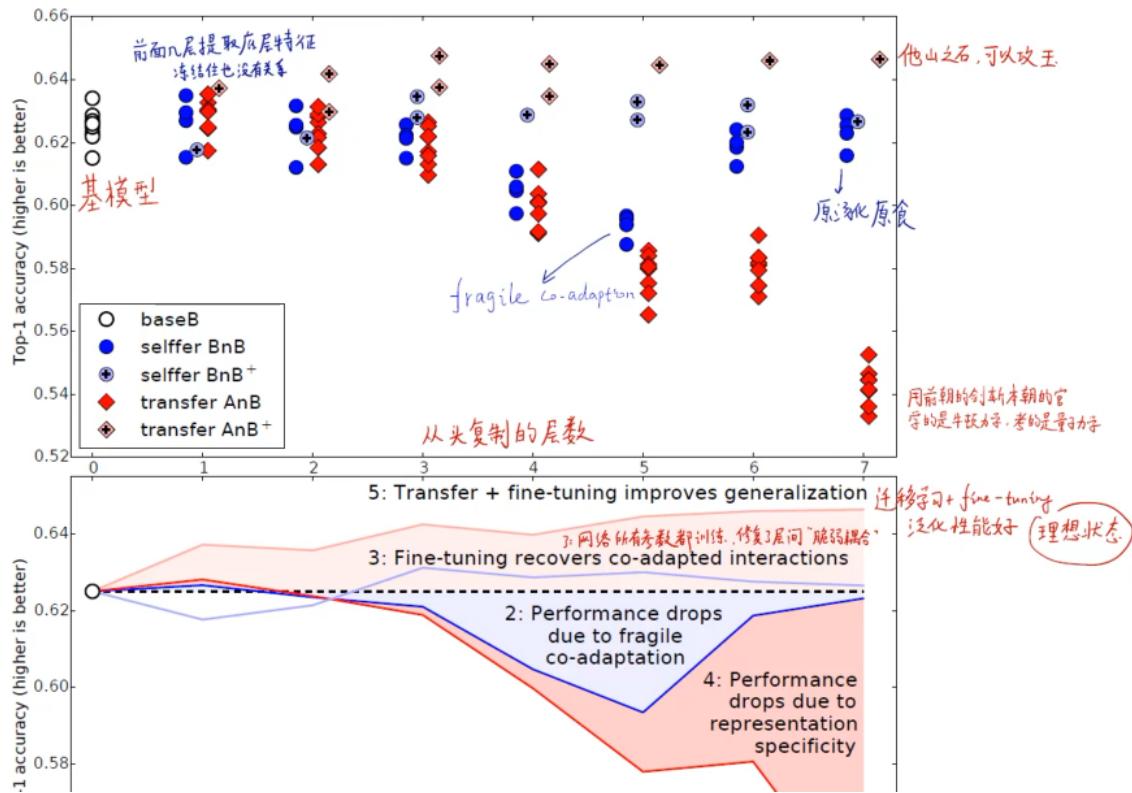
Takeaway for your projects and beyond:

Have some dataset of interest but it has $< \sim 1M$ images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

迁移学习的思想认为，对于FC线性层之前的conv、pool等层，均起到了提取特征等作用，能提取到普适信息。对于各种计算机问题应该是普适的，因此大多数问题不需要重新训练前面的内容，只需要重新训练FC层。



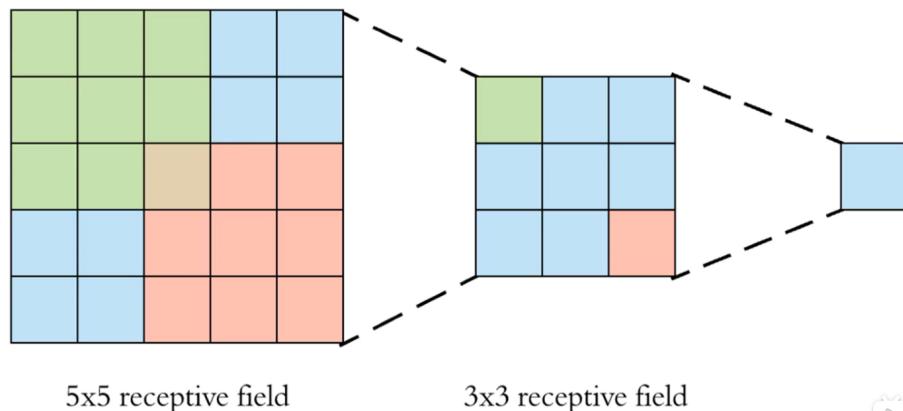


不能随便打破conv之间的联系（保留前面的conv而重新训练后面的conv），不然可能和上图蓝线一样发生性能的下降。

Lecture 6: CNNs in Practice

How to stack them (The power of small filters)

- 可以用两个 3×3 的卷积核替代 5×5 的卷积核，减少计算量，且计算结果不变。



根据计算公式 $(n + 2 * p - f) / stride + 1$:

一个 5×5 卷积: $(32-5)/1+1=28$

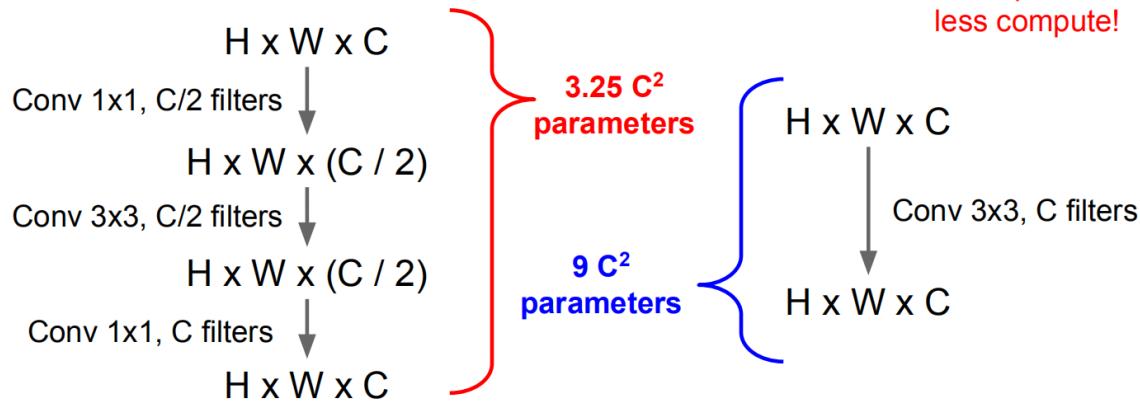
两个 3×3 卷积核: $(32-3)/1+1=30$; $(30-3)/1+1=28$

堆叠含有小尺寸卷积核的卷积层来代替具有大尺寸的卷积核的卷积层，有更少的参数量，并且能够使得感受野大小不变，而且多个 3×3 的卷积核比一个大尺寸卷积核有更多的非线性（每个堆叠的卷积层中都包含激活函数），使得decision function更加具有判别性。

- 同理，可以用三个 3×3 的卷积核替代一个 7×7 的卷积核

Why stop at 3×3 filters? Why not try 1×1 ?

More nonlinearity,
fewer params,
less compute!



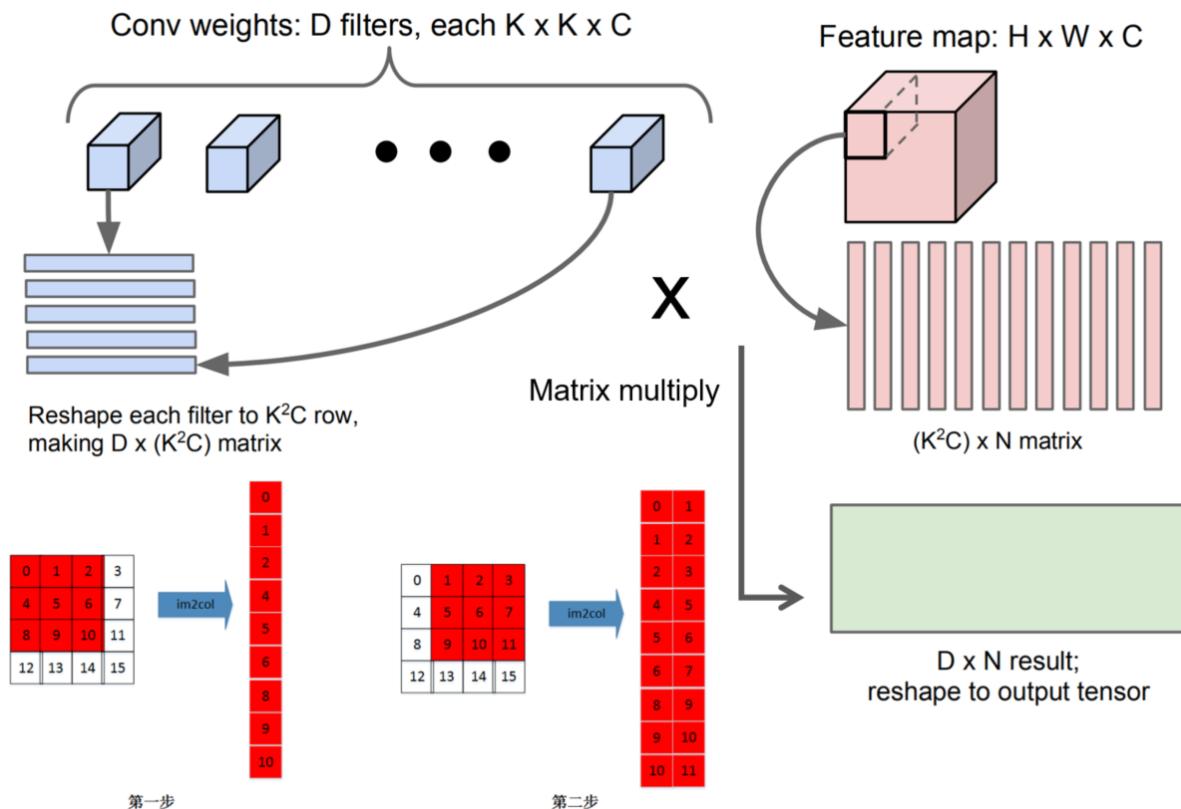
$$\text{kernal size} \times \text{kernal size} \times \text{input channels} \times \text{filter nums}$$

How to compute them

将卷积运算变为矩阵运算，利用现成的矩阵加速运算工具包进行高速计算。

Can we turn convolution into matrix multiplication?

im2col



- K 表示卷积核的大小
- N 表示一个feature map中元素的个数
- D 表示卷积核的个数
- 最后矩阵乘法得到一个 $D \times N$ 的结果，相当于每一行表示一个卷积核生成的feature map拉平得到的 N 个元素。

a.g. $32 \times 32 \times 3$, $\text{kernel_size}=5$, $\text{slide}=1$, $\text{padding}=0$, feature map的通道数为6

则 $f = K = 5$, $D = 6$, $(n+2p-f)/slide+1 = 32-5+1 = 28$, $N = 28 \times 28$

最后得到 $D \times N = 6 \times 28 \times 28$ 的结果, 将 N 重新reshape成 28×28 即可完成这一次的卷积操作。

FFT

Convolution Theorem: The convolution of f and g is equal to the elementwise product of their Fourier Transforms:

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$$

FFT steps:

- Compute FFT of weights: $F(W)$
- Compute FFT of image: $F(X)$
- Compute elementwise product: $F(W) \circ F(X)$
- Compute inverse FFT: $Y = F^{-1}(F(W) \circ F(X))$

对于大卷积核有比较好的效果, 对于小卷积核没有明显优势。

Lecture 7: CNN Architectures and Recurrent Neural Networks

CNN Architectures

AlexNet : 开山鼻祖

重要说明 :

- 用于提取图像特征的卷积层以及用于分类的全连接层是同时学习的;
- 卷积层与全连接层在学习过程中会相互影响、相互促进

重要技巧 :

- Dropout策略防止过拟合;
- 使用加入动量的随机梯度下降算法, 加速收敛;
- 验证集损失不下降时, 手动降低10倍的学习率;
- 采用样本增强策略增加训练样本数量, 防止过拟合;
- 集成多个模型, 进一步提高精度。

ZFNet : 轻微改动

与AlexNet网络结构基本一致！

主要改进：

- 将第一个卷积层的卷积核大小改为了 7×7 ；
- 将第二、第三个卷积层的卷积步长都设置为2；
- 增加了第三、第四个卷积层的卷积核个数。

VGGNet：提出用多个small 卷积核替代big 卷积核

问题2：为什么VGG网络前四段里，每经过一次池化操作，卷积核个数就增加一倍？

回答：

1. 池化操作可以减小特征图尺寸，降低显存占用
2. 增加卷积核个数有助于学习更多的结构特征，但会增加网络参数数量以及内存消耗
3. 一减一增的设计平衡了识别精度与存储、计算开销

问题3：为什么卷积核个数增加到512后就不再增加了？

回答：

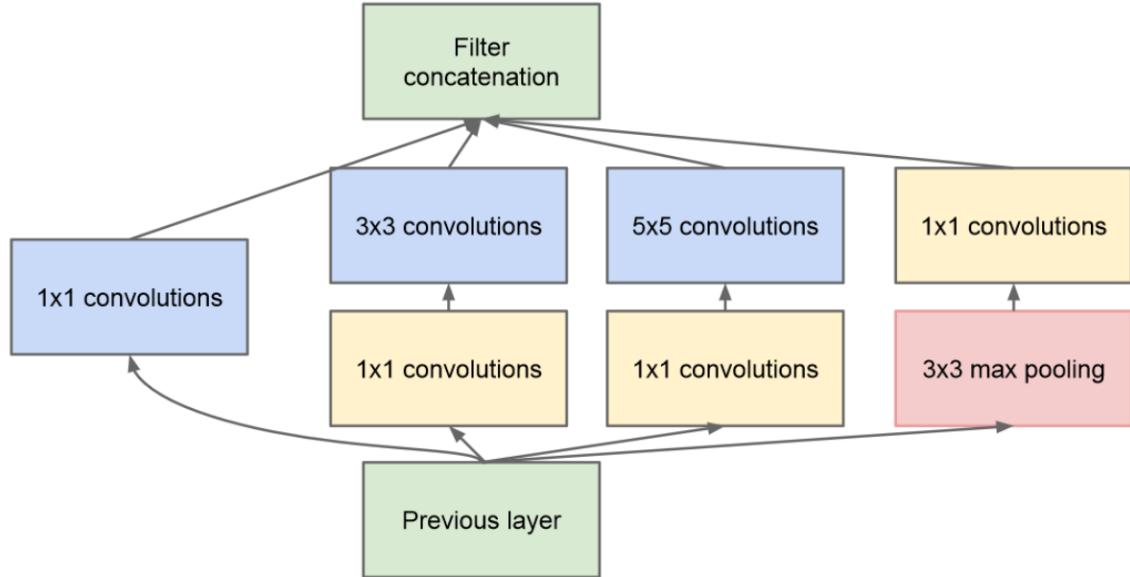
1. 第一个全连接层含102M参数，占总参数个数的74%；
2. 这一层的参数个数是特征图的尺寸与个数的乘积；
3. 参数过多容易过拟合，且不易被训练

GoogLeNet：提出Inception module，网络由多个Inception module组成，取消了FC层。

- 提出了一种Inception结构，它能保留输入信号中的更多特征信息；
- 去掉了AlexNet的前两个全连接层，并采用了平均池化，这一设计使得

GoogLeNet只有500万参数，比AlexNet少了12倍；

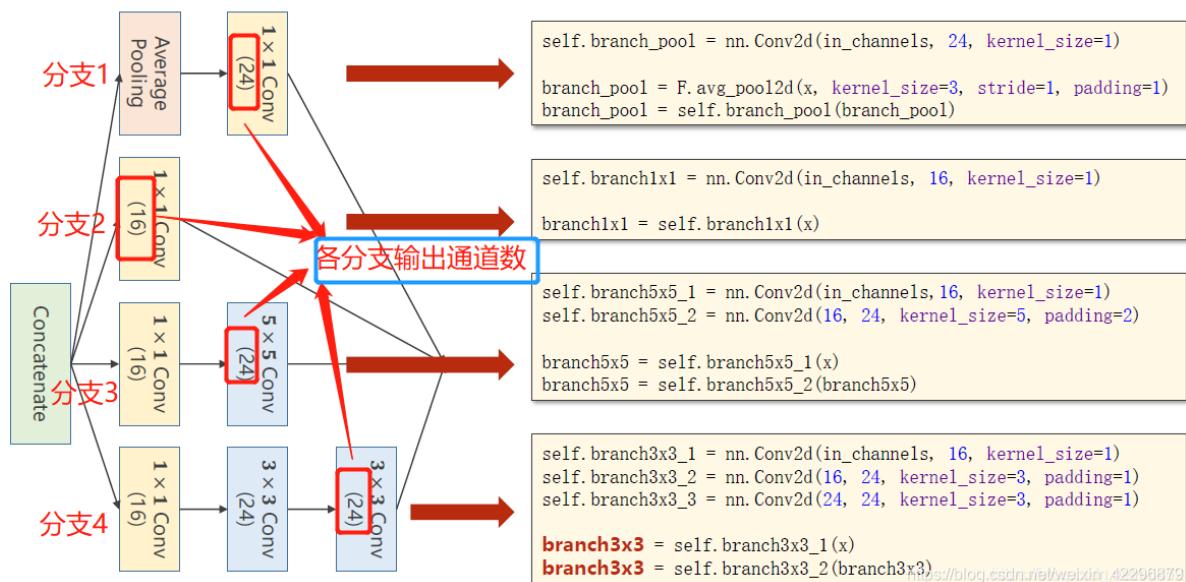
- 在网络的中部引入了辅助分类器，克服了训练过程中的梯度消失问题。



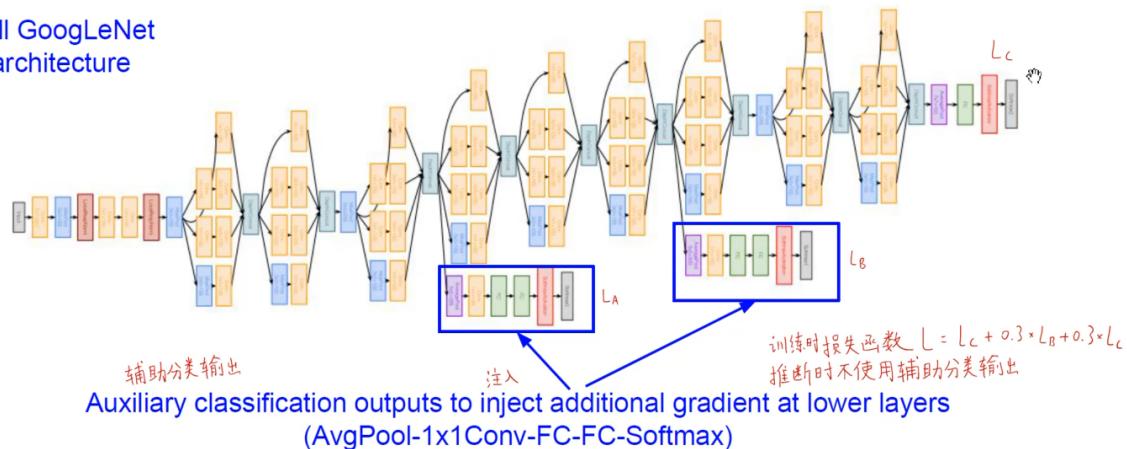
针对不同图像中同类物体尺度不同的问题，采用不同尺度的卷积核来解决这个问题。

1x1起到了数据压缩的作用，3x3和5x5起到了提取不同尺度特征的作用，3x3 max pooling起到了将响应大的像素进行扩张（即3x3邻域的像素值都变为与最大的中心点相同）。在最后concatenation时要保证每条分支拼接时的H和W相同，channel数直接叠加。如：96+96+96+48个通道

Implementation of Inception Module

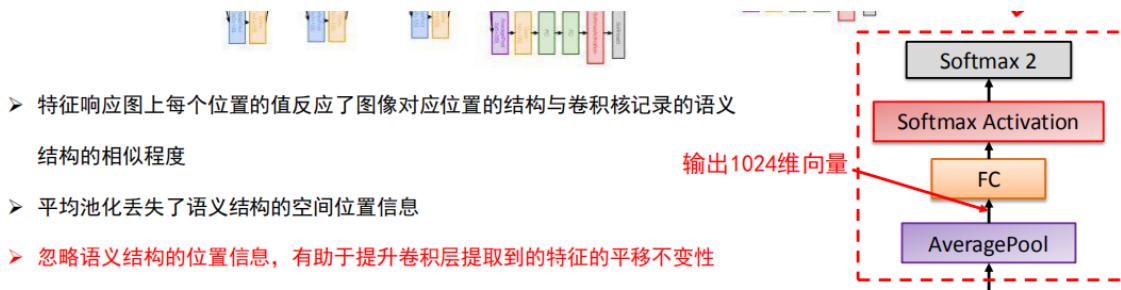


Full GoogLeNet architecture



增加了两个提前分类项，可以提前进行反向传播，相当于增加了正则项。

对输出feature map平均池化，每个feature map取一个平均值，取消前两层FC，参数大大下降。

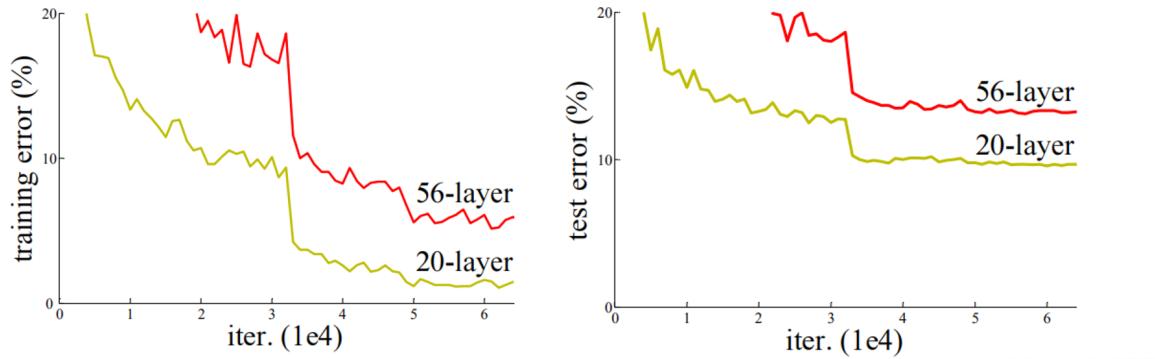


ResNet:

- 提出了一种残差模块，通过堆叠残差模块可以构建任意深度的神经网络，而不会出现“退化”现象。
- 提出了批归一化方法来对抗梯度消失，该方法降低了网络训练过程对于权重初始化的依赖；
- 提出了一种针对ReLU激活函数的初始化方法；

“越深的网络准确率越高”这一观点是错误的，该现象被称为“退化（Degradation）”。同时，太深网络容易出现梯度消失或梯度爆炸。

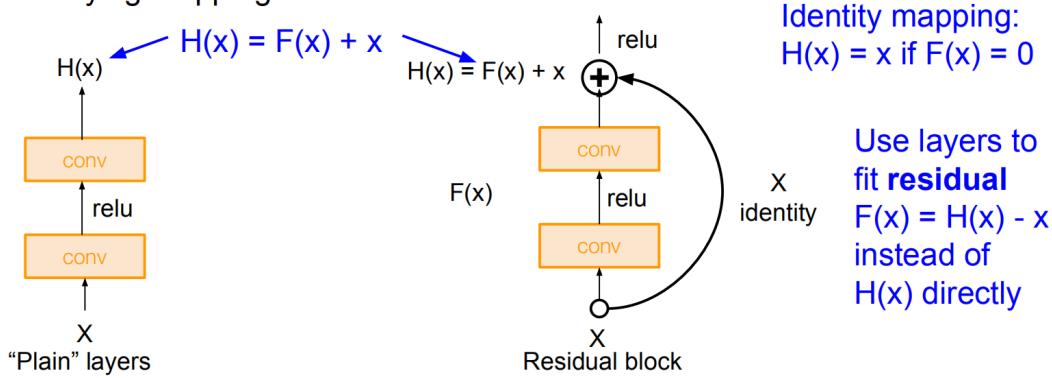
在ResNet论文中说通过数据的预处理以及在网络中使用BN（Batch Normalization）层能够解决梯度消失或者梯度爆炸问题。



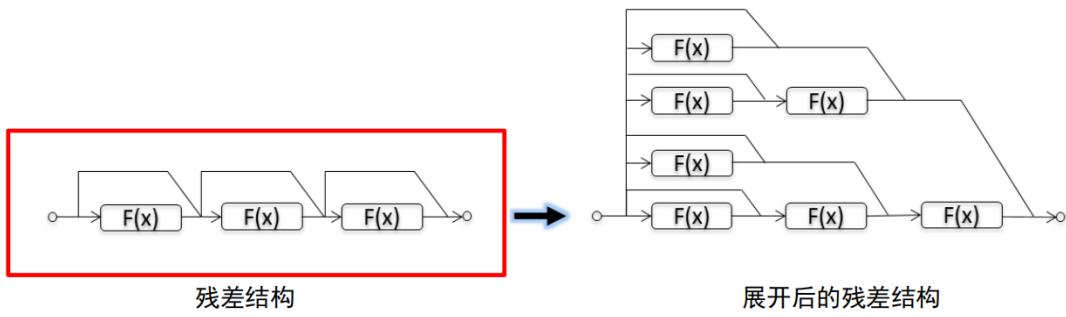
退化现象让我们对非线性转换进行反思，非线性转换极大的提高了数据分类能力，但是，随着网络的深度不断的加大，我们在非线性转换方面已经走的太远，竟然无法实现线性转换。

在ResNet提出的**residual**残差结构中增加了Shortcut Connection分支，在线性转换和非线性转换之间寻求一个平衡。

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



将堆叠的几层layer称之为一个block，对于某个block，其可以拟合的函数为 $F(x)$ ，如果期望的潜在映射为 $H(x)$ ， $F(x)$ 直接学习潜在的映射，不如去学习残差 $H(x) - x$ ，即 $F(x) = H(x) - x$ ，这样原本的前向路径上就变成了 $H(x) = F(x) + x$ ，来拟合 $H(x)$ 。因为**相比于让 $F(x)$ 学习成恒等映射，让 $F(x)$ 学习成0要更加容易**。这样，对于冗余的block，只需 $F(x) \rightarrow 0$ 就可以得到恒等映射，性能不减。

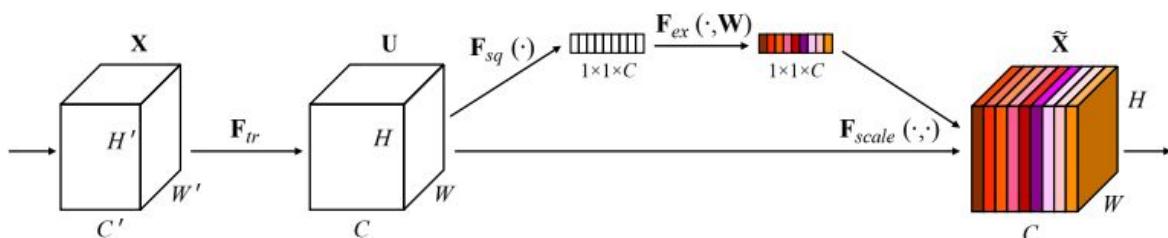


结论：残差网络是一种集成模型，这是重要特点也是它高效的一个原因！



残差的效果其实就类似这种图像增强。

SENet:

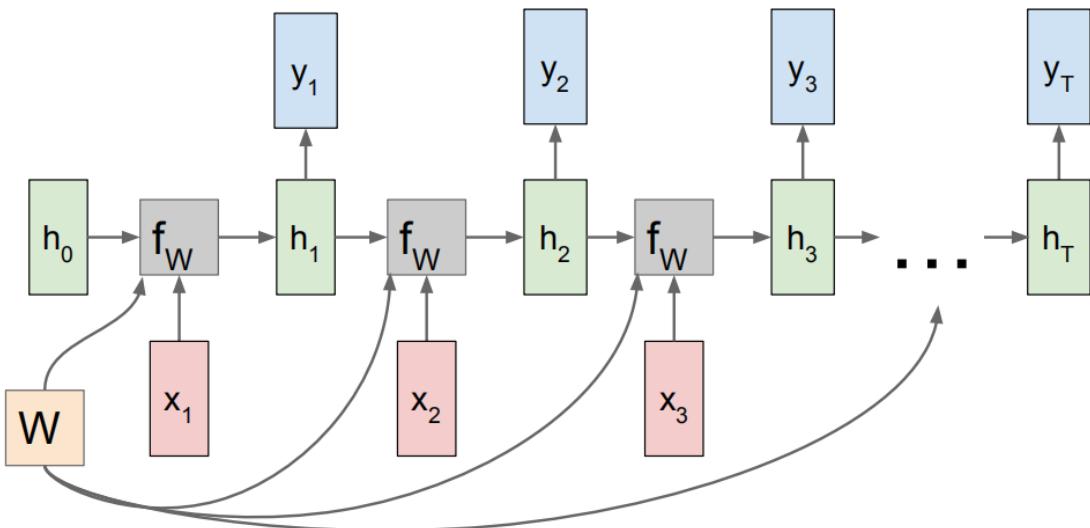


可以自适应学习得到每个channel的权重。

Recurrent Neural Networks

RNN适用于序列数据，广泛应用于NLP领域。

用于图像分类领域，就要将处理序列数据的方法用于非序列数据。



$$h_t = f_W(h_{t-1}, x_t)$$



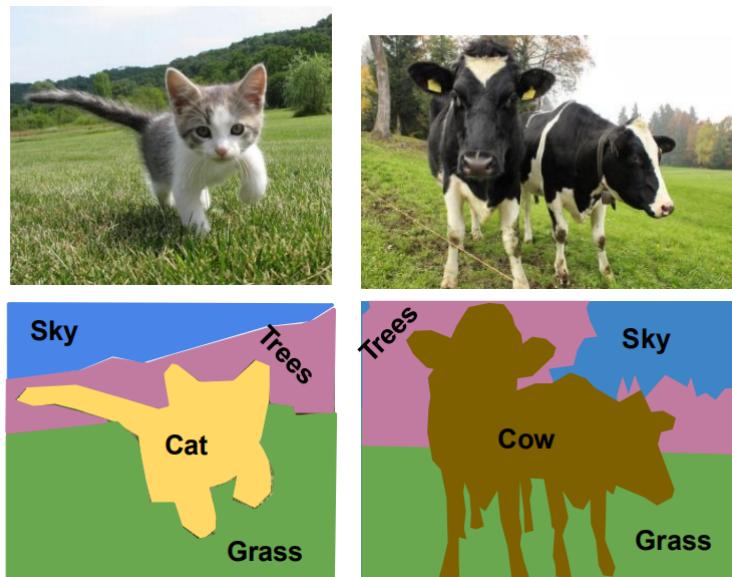
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

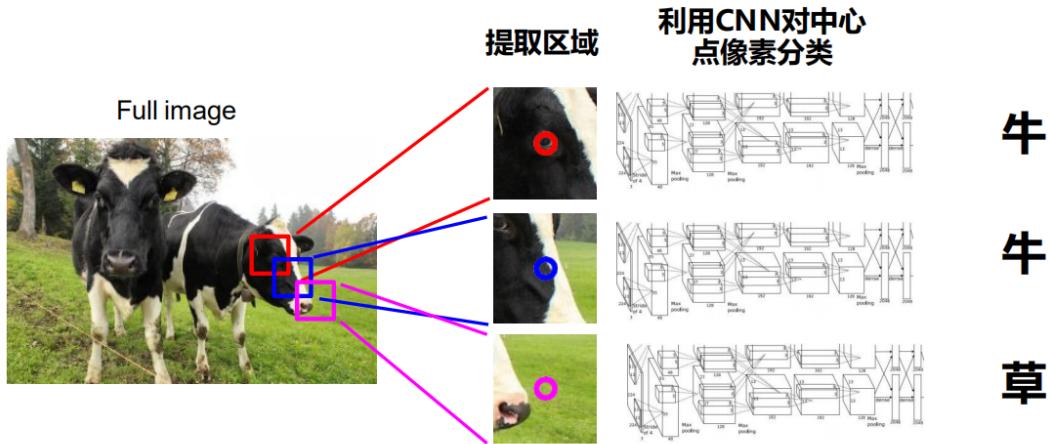
W 权重是共享的 (W_{hh} W_{xh} W_{hy} 均相同)。

Lecture 8: Semantic Segmentation

语义分割给每个像素分配类别标签，但是不区分实例，只区分类别。



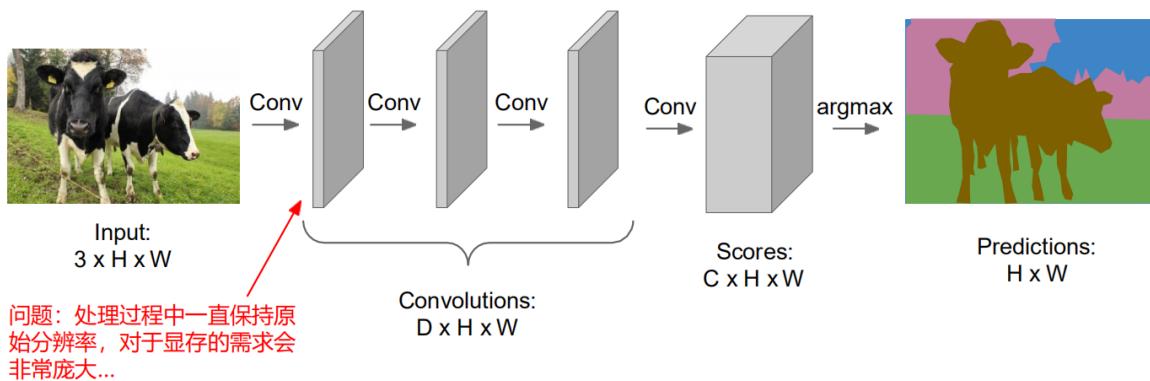
Sliding Window



- 取中心点周围的某个区域，对该区域进行分类任务。
- 但是效率太低！重叠区域的特征反复被计算

Fully Convolution Network (FCN)

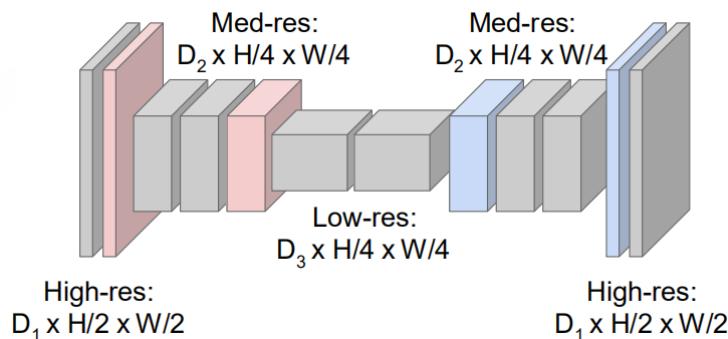
解决方案：让整个网络只包含卷积层，
一次性输出所有像素的类别预测。



让最后的输出为C个通道，与分类的类别数目相同。对于每一个特定的位置，对应一个C X 1的向量。将标签设为 $[1 \ 0 \ 0 \ 0 \dots \ 0]_{1 \times c}$ ，表示该像素本应该属于第一类，然后用 softmax 的思路训练网络。

问题：处理过程中一直保持原始分辨率，对于显存的需求会非常庞大

解决方案：让整个网络只包含卷积层，并在网络中嵌入下采样与上采样过程。



下采样

可以用: pooling、sliding > 1等方法

上采样

Unpooling (反池化操作)

Nearest Neighbor

1	2
1	1
3	4

Input: 2 x 2

“Bed of Nails”

1	2
0	0
3	4

Input: 2 x 2

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Output: 4 x 4

1	0	2	0
0	0	0	0
3	0	4	0
0	0	0	0

Output: 4 x 4

这两种都不太好。更常用的是下面这种

Max Unpooling

Max Pooling

Remember which element was max!

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

Input: 4 x 4

Output: 2 x 2

Rest of the network

Max Unpooling

Use positions from
pooling layer

1	2
3	4

Input: 2 x 2

0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4

Output: 4 x 4

即需要在max pooling时记下保留的是哪个位置的值，然后在上采样的时候放到对应位置。

Transpose Convolution

转置卷积是一种可学习的上采样

3 x 3 转置卷积 (transpose convolution) , stride 2 pad 1

Input: 2 x 2

Input gives weight for filter

Output: 4 x 4

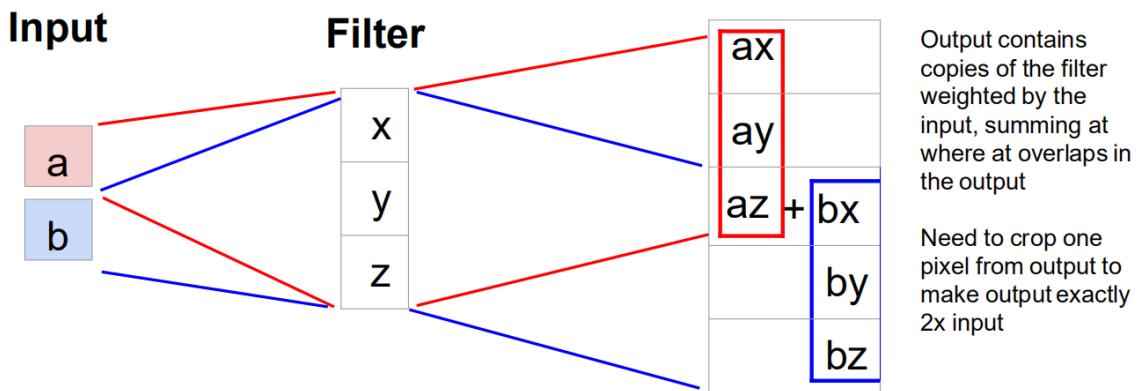
重叠区域的值需
要求和

Filter moves 2 pixels in
the output for every one
pixel in the input

Stride gives ratio between
movement in output and
input

即重叠区域由两个input的像素共同决定，需要学习这两个分配的权重。举一个一维的例子：

卷积与矩阵相乘 (一维例子)



卷积与矩阵相乘 (一维例子)

将卷积写为矩阵乘法

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & x & y & x & 0 & 0 \\ 0 & 0 & x & y & x & 0 \\ 0 & 0 & 0 & x & y & x \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

例子: 1D 卷积, 卷积核尺寸

步长=1, 零填充=1

$$\vec{x} = [x, y, z]$$

$$\vec{a} = [a, b, c, d]$$

$$Y = \vec{x} * \vec{a}$$

$$[0, a, b, c, d, 0]$$

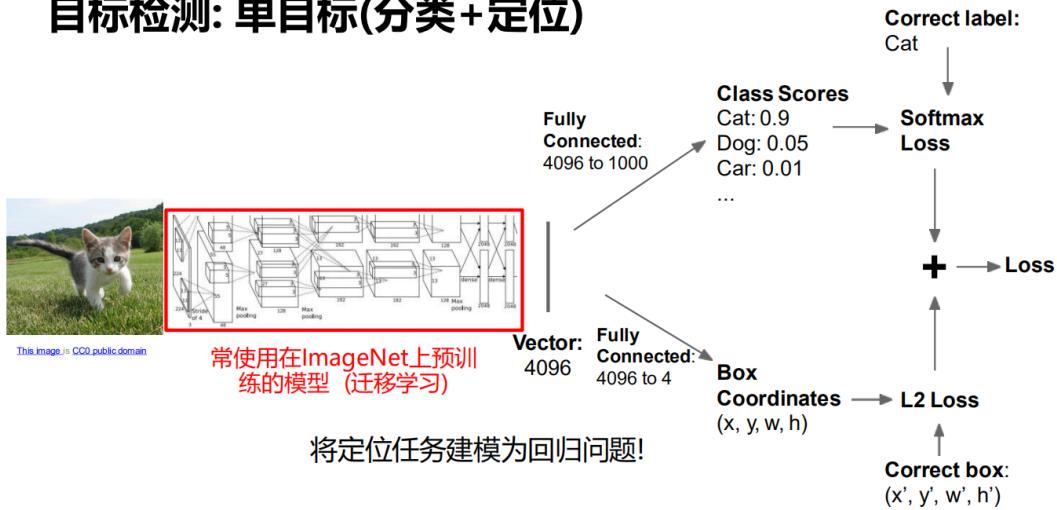
将向量 x 写成 X 矩阵的形式, 相当于得到了移动卷积核在不同位置与增广(即增加padding)后的待卷积向量进行卷积的结果。可以用一次矩阵运算得到。由此, 可得逆运算的上采样结果:

$$\vec{x} *^T \vec{a} = X^T \vec{a}$$

$$\begin{bmatrix} x & 0 & 0 & 0 \\ y & x & 0 & 0 \\ z & y & x & 0 \\ 0 & z & y & x \\ 0 & 0 & z & y \\ 0 & 0 & 0 & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} ax \\ ay + bx \\ az + by + cx \\ bz + cy + dx \\ cz + dy \\ dz \end{bmatrix}$$

Lecture 9: Object Detection

目标检测: 单目标(分类+定位)

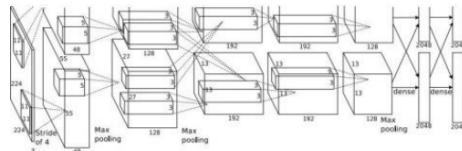
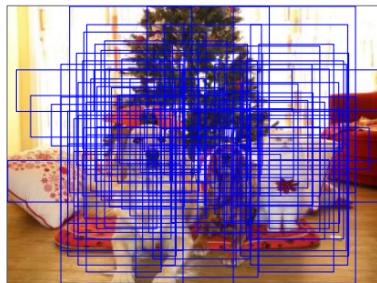


最后输出一个1004维的向量。前1000维对应softmax输出的概率值，用来判断锚框中的物体属于哪个类别；最后四维表示锚框的位置，一般是锚框左上角的坐标（opencv中好像是锚框中心的坐标）以及锚框的长宽。分类问题得到一个交叉熵loss，定位问题得到一个L2loss，用两个loss加个权值配比得到最终的loss，即任务转化为一个多目标损失问题。

但是这种处理方法必须提前确定有多少个目标，才能确定输出向量的维度。因此该方法不适用于多目标问题。

多目标检测总体方法

利用CNN对图像中的区域进行多分类，
以确定当前区域是背景还是哪个类别的
目标。



狗? 不是
猫? 是
背景? 不是

困境：CNN需要对图像中所有可能的区域（不同位置、尺寸、长宽比）进行分类，计算量巨大！

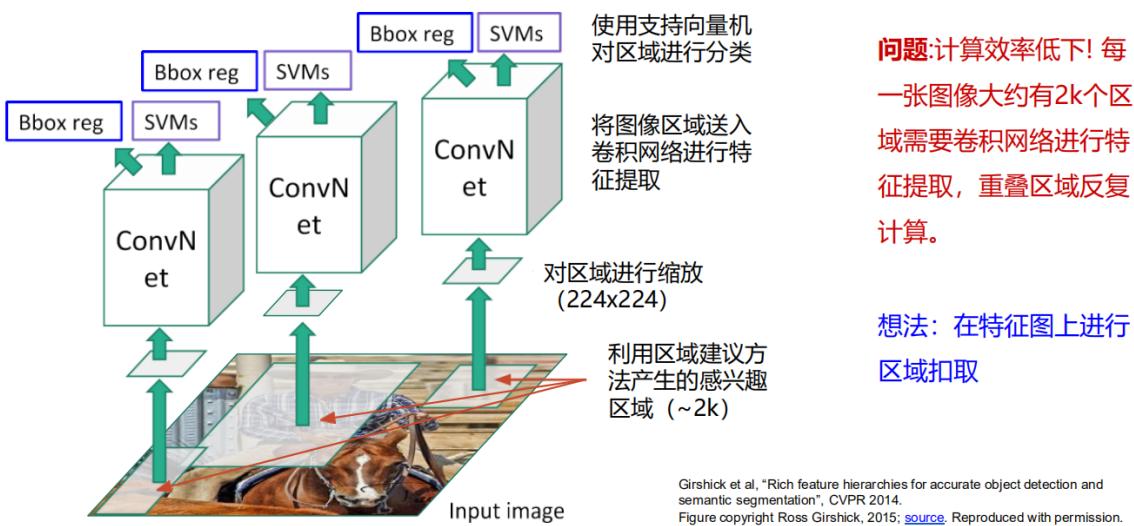
即利用sliding windows的方法，选择不同尺度的区域，对该区域进行分类。（增加了背景类）

但是计算量过大：需要Selective Search，事先找出所有潜在可能包含目标的区域，仅对这些区域进行操作。

Two-stage object detector

RCNN

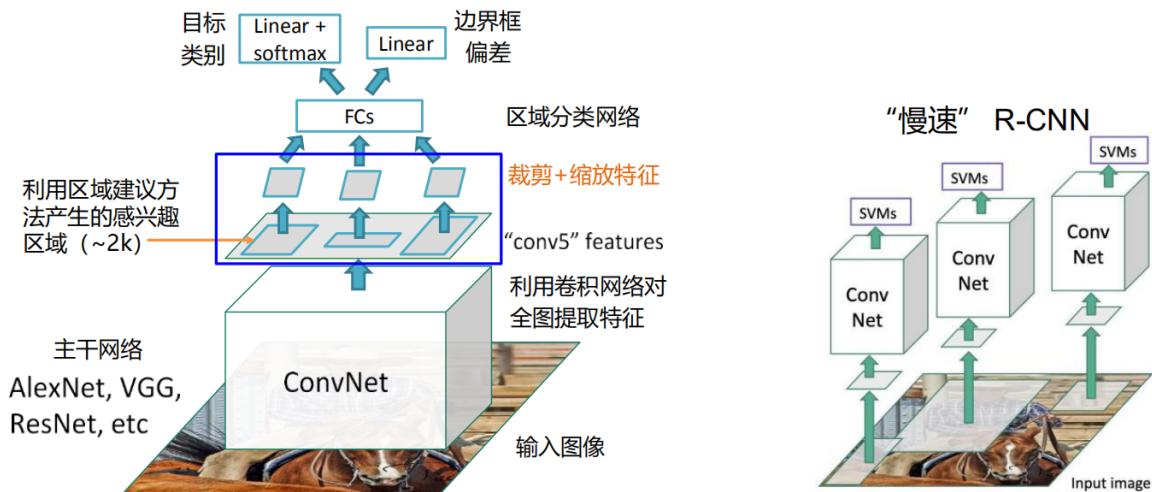
R-CNN



仅用CNN进行特征提取，取消了最后的两层用于分类的FC层，改用SVM

Fast R-CNN

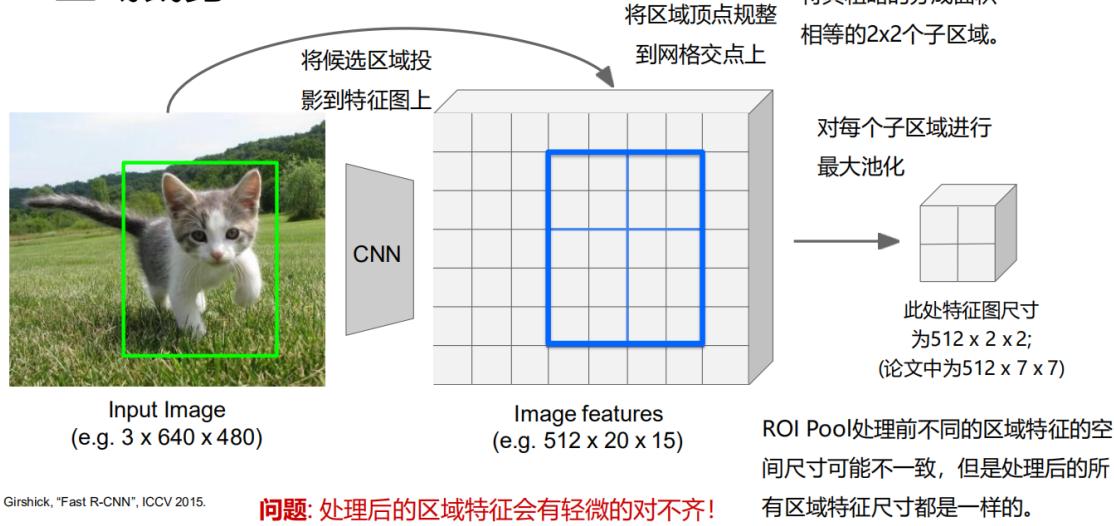
- **Feature Extraction**: 使用ConvNet只对图像进行一次计算提取特征图；
- **Sample Rols**: 在原图上选择性搜索生产 region proposals；
- **ROI Projection**: 将选择性搜索生成的 region proposals 投影到CNN提取的特征图上；
- **ROI Pooling**: ROI Pooling 层将 ROI Projection后的特征变成固定大小的特征图输出；
- **分类和回归**: 最后，通过后续的FC层分别生成bbox和class。



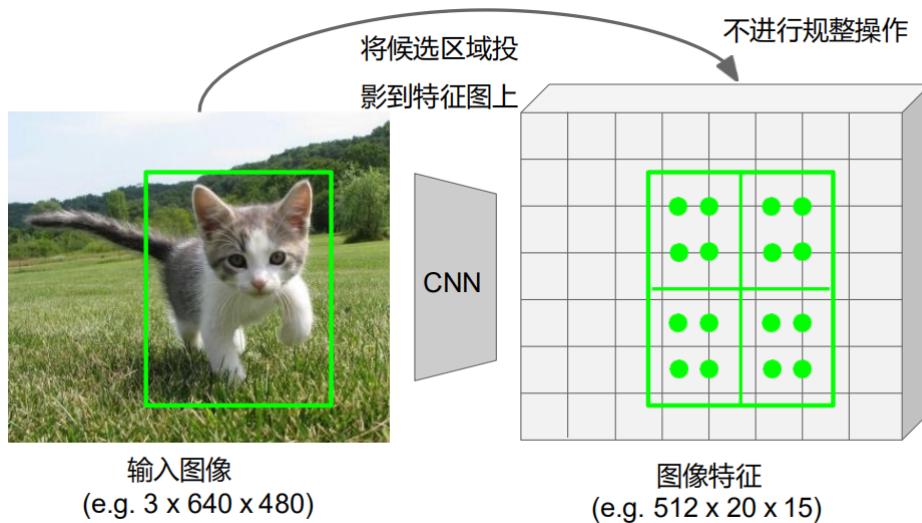
- Selective Search的结果直接投影到经过CNN的特征图上，这样做的好处是，原来建议框重合部分非常多，卷积重复计算严重，而这里每个位置都只计算了一次卷积，大大减少了计算量。
- 由于建议框大小不一，得到的特征框需要转化为相同大小，需要对建议框区域进行裁剪和缩放。这一步是通过ROI池化层来实现的（ROI表示region of interest即目标）

ROI Pooling

区域裁剪: ROI Pool

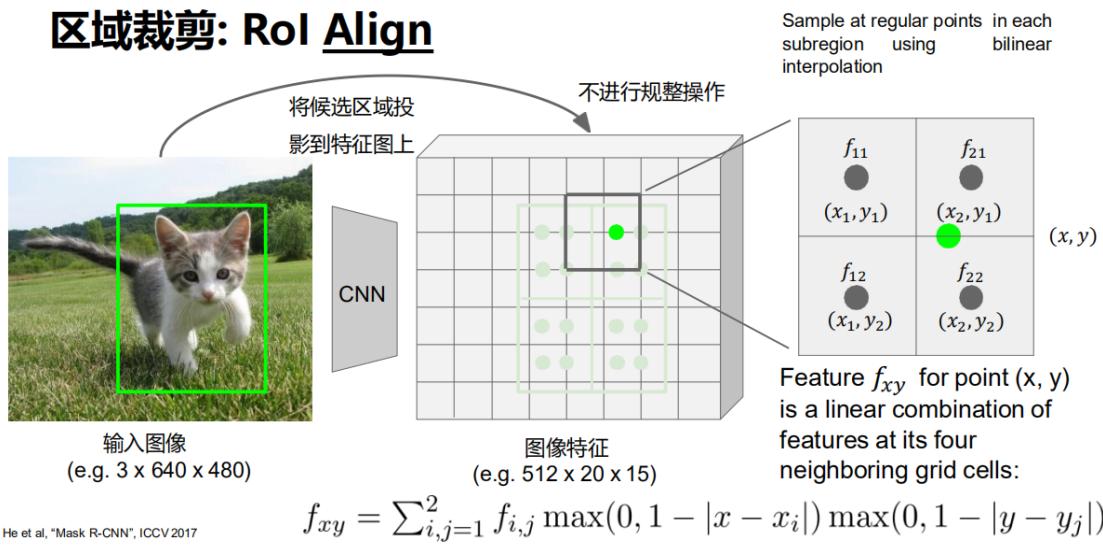


ROI Align



直接进行投影, 不对齐到整点, 并且在区域内选择几个等间隔的点。

区域裁剪: ROI Align



利用标准网格上的4个点通过双线性插值得到自己取的绿色点的坐标。

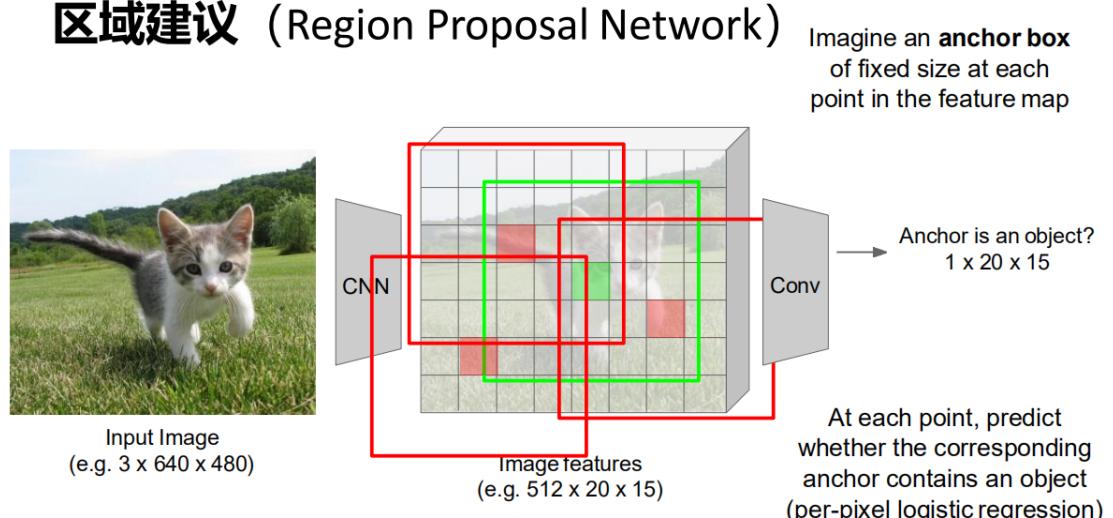
然后利用4个绿色点，对每个区域进行max pooling，相较 Roi pooling更加准确。

Faster R-CNN

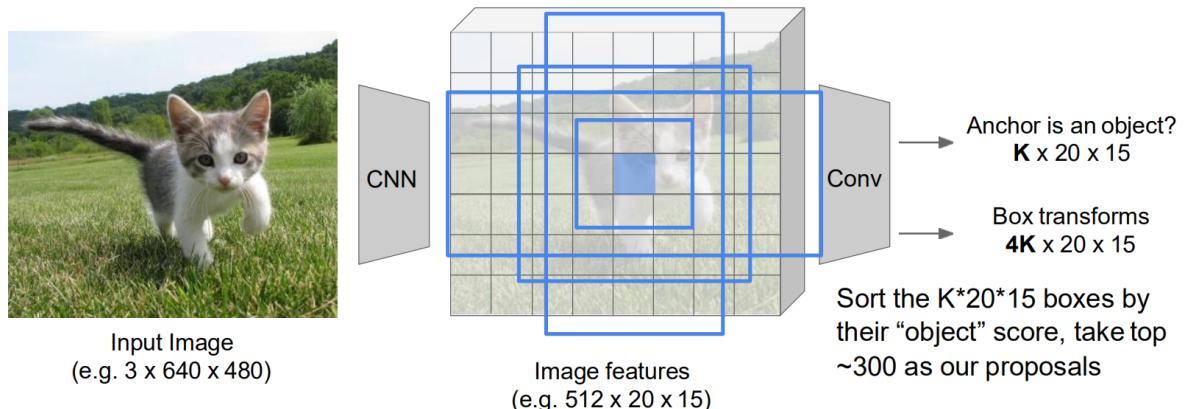
1. Conv layers。作为一种CNN网络目标检测方法，Faster RCNN首先使用一组基础的 conv+relu+pooling层提取image的feature maps。该feature maps被共享用于后续RPN层和全连接层。
2. Region Proposal Networks。RPN网络用于生成region proposals。该层通过softmax判断 anchors属于positive或者negative，再利用bounding box regression修正anchors获得精确的proposals。
3. Roi Pooling。该层收集输入的feature maps和proposals，综合这些信息后提取proposal feature maps，送入后续全连接层判定目标类别。
4. Classification。利用proposal feature maps计算proposal的类别，同时再次bounding box regression获得检测框最终的精确位置。

相较 Fast R-CNN，改成在特征图上直接找感兴趣区域。在特征图上用RPN网络选择合适的锚点 (Anchor) (??)，以锚点为中心，用固定大小的框来框图，对该区域直接进行分类。

区域建议 (Region Proposal Network)



实际使用中，对于每个特征图上的每个位置，我们通常会采用 k 个不同尺寸和分辨率的锚点区域 (anchor boxes)



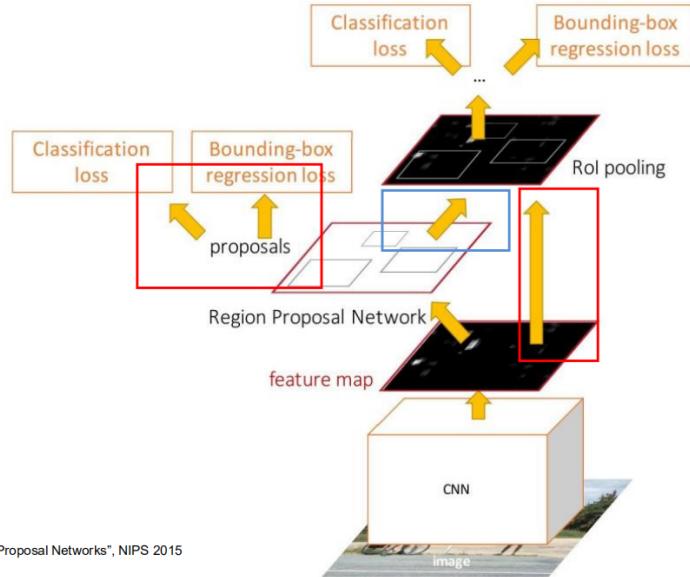
每个anchor处都框出4个不同尺度的anchor box，然后分别分类，看对应 K 个类别中的哪一类。即每一个锚框对应一个 K 维向量 (0-1编码表示属于哪个类别)，所以每个anchor是 $4K$ (4个框)，整张 feature map对应 $4K \times H \times W$ 。而另外一个输出是一个值0或者1来判断该区域是不是我们的目标，所以这部分对应 $K \times H \times W$ 。

Faster R-CNN

利用卷积网络产生候选区域！

四种损失联合训练：

- RPN分类损失(目标/非目标)
- RPN边界框坐标回归损失
- 候选区域分类损失
- 最终边界框坐标回归损失



Ren et al, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", NIPS 2015
Figure copyright 2015, Ross Girshick; reproduced with permission

注意在红色框的地方有梯度传回来，但是在蓝色框的地方没有，这是RoI pooling的缺陷。

以上都属于两阶段的方法：第一阶段特征提取，得到候选区域；第二阶段进行类别和边界框的预测

Single-Stage Object Detectors

YOLO

SSD