CS231n Convolutional Neural Networks for Visual Recognition

Lecture 1 Introduction

Computer vision → study of visual data.

2015 CISCO study estimated that by 2017, roughly 80% of all traffic on the Internet would be video. (from a pure number of bits perspective)

Hubel & Wiesel, 1959.

Visual processing starts with simple structures of the visual world, oriented edges and as information moves along the visual processing pathway, the brain builds up the complexity of the visual information.

David Marr 1970s.

Input image → Primal sketch (Edge image) → 2.5D sketch → 3-D model representation.

Brook & Binford 1979, Generalized Cylinder

Fischler and Elschlager, 1973. Pictorial Structure

basic idea → every object is composed of simple geometric primitives.

Shi & Malik, 1997. Normalized Cut. → group pixels into meaningful areas.

Object Segmentation, image segmentation

Viola & Jones, 2001 Face Detection

AdaBoost algorithm → real-time face detection

first digital camera with face detection → 2006, Fuji Film

David Lowe, 1999. "SIFT" & Object Recognition.

feature based object recognition. (to match an entire object)

invariant ↓ to changes. (light, viewpoint, etc)

Schmid & Ponce 2006, Spatial Pyramid Matching

SVM on feature descriptor.

Felzenszwalb, McAllester, Ramanan 2009, Deformable Part Model

Dalal & Triggs 2005, Histogram of Gradients.

PASCAL visual object challenge.

IMAGENET. Large Scale Visual Recognition Challenge.

所以此时数据复杂维数高，容易 overfit. 当训练数据量不够时.

Convolutional Neural Network 最早在90年代发明，但取决于计算能力和数据集，直到2012年才取得突破.
↓
需要大量带标签的图像和像素

Lecture 2 Image Classification Pipline

An image is just a big grid of numbers between [0, 255].
eg. 800×600×3 (3 channels RGB)

Challenges: → algorithm must be robust.
- viewpoint variation
- illumination.
- occlusion 遮挡.
- Deformation 形变.
- background clutter 背景杂乱
- Intraclass variation 类内差异.

- hard-core algorithm (e.g. find edges, find corners...)
  ↳ super brittle 不准确.
  not scalable, 对不同的类别需要重新写对应的算法

- Data-Driven Approach.
  1. collect a dataset of images and labels
  2. Use machine learning to train a classifier
  3. Evaluate the classifier on new images.

  train → predict

First classifier: Nearest Neighbor.

```
def train(images, labels): # Memorize all data and labels
    return model
def predict(model, test_images): # Predict the label of the most
    return test_labels                similar training image.
```

with N examples, Train O(1), Predict O(N). ← Bad.

we want classifiers that are fast at prediction, slow for training is ok.

Tips:

在思考 computer vison 问题时，在不同的角度反复思考验证很有用.

- idea of high dimensional points in the plane
  如将每张图片看作是高维度空间中的一点.
- concrete images (high dimensional vectors) of pixels.
  具体到图片像素

K-Nearest Neighbors

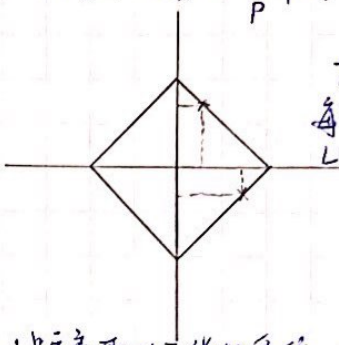
Instead of copying label from nearest neighbor, take majority vote from K closest points.

K越大使决策边界更平滑.

Distance Metric.

L1 (Manhattan) distance.

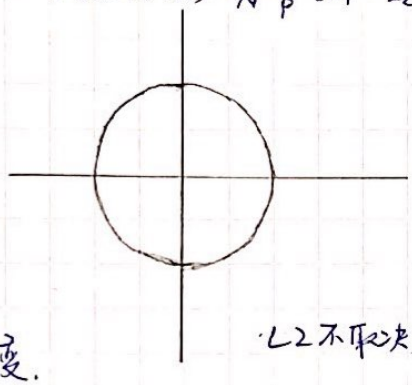
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



正方形边界上每点到原点在L1都是等距的

L2 (Euclidean) distance

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



L1距离取决于坐标系统，如果坐标轴转动，L1也会改变.

在输入变量的值具有重要意义(如用户年龄，收入等)，L1距离更合适.
(features)

L2不取决于坐标系统.

在KNN结果对比中，L1的结果显示决策边界倾向于沿着坐标轴.
而L2结果显示不受坐标轴限制，边界比较自然.

Hyperparameters, choices about the algorithm that we set rather than learn.

↓

如K-NN中的K, distance metric.

very problem-dependant, must try them all out and find the optimal.

找到最佳 hyperparameters 或训练模型的方式：

将数据集分为 train, validation, test.

在 training set 上，用不同的 hyperparameter 训练算法，在 validation set 上评估

找出表现最好的一组参数. 最后在 test set 上跑一次得到最终数据.

(test set 独立于训练数据，检验模型泛化能力) 相当于"新数据"

(cross-validation 主, split data into folds + test set, try each fold as validation and average the results)

实际应用中，k-NN on images never used.

- very slow at test time
- Distance metrics on pixels are not informative.
- curse of dimensionality.

随着维数增加，计算量呈指数级增加.

Linear Classification

Image (Array of $x \times y \times 3$ numbers) ⟶ $f(x, W)$ (W, parameters or weights) ⟶ predictions on class scores.

$Wx + b$

通过训练得到 W 包含模型信息，可以有效地在应用端使用，不需要原本的 data set x.

X 表示图片中的 pixels.

解读：— 有在线性不可分的情况

— one single-templete per category (W 中每一行对应一个类，每个值对应一个 pixel)

换一个角度：在高维空间中，每个图片为一个点，Linear classifier 在决策边界上画一个线性分类面来划分一个类别.

Further Reading

< A Few Useful Things to Know about Machine Learning >

"Recognizing and Learning Object Categories" people.csail.mit.edu/torralba/shortCourseRLOC/index.html

Lecture 3 Loss Functions and Optimization

1. Loss function quantifies our unhappiness with the scores across training data.
2. Optimization, a way of efficiently finding the parameters that minimize the loss function.

Multiclass Support Vector Machine Loss

The SVM loss is set up so that the SVM "wants" the correct class for each image to have a score higher than the incorrect classes by some fixed margin Δ.

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

data (x_i, y_i) (y_i: label)

s_j: score for the jth class $s_j = f(x_i, W)_j$

e.g. linear score function $f(x_i, W) = Wx_i$

$$L_i = \sum_{j \neq y_i} \max(0, w_j x_i - w_{y_i} x_i + \Delta)$$

$\max(0, -) \rightarrow$ hinge loss (max-margin loss)

Regularization.

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}} \quad \text{Full loss}$$

flash-back,

$$\min \ \frac{1}{2}\|W\|^2 + C \sum_{i=1}^{m} \xi_i \longrightarrow \text{SVM with L1 regularization}$$

$$\text{s.t.} \ \ y^{(i)}(w^T x^{(i)} + b) \geq 1, \ i = 1, \dots, m$$

$$\xi_i \geq 0, \ i = 1, \dots, m$$

Occam's Razor: Among competing hypotheses, the simplest is the best.

Regularization methods:

- L1 $R(W) = \sum \|W\|_1$
- L2 $R(W) = \sum \|W\|_2^2$
- Elastic net (L1 + L2) $R(W) = \sum (\beta \|W\|_2^2 + \|W\|_1)$
- Max norm regularization
- Dropout
- Batch normalization, stochastic depth ...

Intepretation for L^1 and L^2 regularization:

$L1$ regularization → encouraging sparsity

$L2$ regularization → for W, spread the influence across all the values in x to generate y_i.
robust to the change of x_i.
sensitive to outlier.

Practical Considerations:

- The exact value of the margin between the scores (e.g. $\Delta = 1$ or, $\Delta = 100$) is meaningless because the weights can shink or stretch the differences arbitrarily. (it's safe to set $\Delta = 1.0$ in all cases)
- In the case with Neural Networks in general, we will always work with the optimization objectives in their unconstrained primal form. (many of these objectives are technically not differentiable, but in practice, this is not a problem and it's common to use a subgradient).

Softmax classifier

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \Longrightarrow L_i = -f_{y_i} + \log\sum_j e^{f_j} \rightarrow \text{cross-entropy loss.}$$

$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$: softmax function

The cross-entropy between a "true" distribution p and an estimated distribution q is defined as:

$$H(p, q) = -\sum_x p(x) \log q(x)$$

The softmax classifier is hence minimizing the cross-entropy between the estimated class probabilities ($q = \frac{e^{f_{y_i}}}{\sum e^{f_j}}$) and the "true" distribution (p = [0...1...0] contains a single 1 at the y_ith position).

$$H(p, q) = H(p) + D_{KL}(p \| q)$$

In this case $H(p) = 0$. Hence, softmax or this cross-entropy objective wants to minimizing the KL divergence between the two distributions (a measure of distance). $\left(D_{KL}(P\|Q) = \sum_x P(x) \cdot \log\frac{P(x)}{Q(x)}\right)$

Maximum likelihood and Maximum a posteriori estimation.
(ML) (MAP)

Estimate an unobserved population parameter θ on the basis of observation x.

f is the sampling distribution of x, $\rightarrow f(x|\theta)$

ML: maximum likelihood estimate of $\theta \rightarrow \hat{\theta}_{MLE}(x) = \arg\max_{\theta} f(x|\theta)$

MAP: assume a prior distribution g over θ exsits.

$$f(\theta|x) = \frac{f(x|\theta)g(\theta)}{\int_{\theta} f(x|\nu)g(\nu)d\nu}$$

$P(A,B) = P(A|B)P(B) = P(B|A)P(A)$

$\int_{\theta} f(x|\nu)g(\nu)d\nu \rightarrow$ doesn't depend on θ and always >0

$$\hat{\theta}_{MAP}(x) = \arg\max_{\theta} f(\theta|x) = \arg\max_{\theta} f(x|\theta)g(\theta)$$

For softmax classifier, we can interpreted the $R(W)$ in full lose function as coming from a gaussian prior over the weight matrix W, where instead of MLE, we perform MAP-estimation.

Practical issue:

Numeric stability, $\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{C \cdot e^{f_{y_i}}}{C \sum_j e^{f_j}} = \frac{e^{f_{y_i}+\log C}}{\sum_j e^{f_j+\log C}}$

Diving large $\sum_j e^{f_j}$ can be numerically unstable.

(common choice: $\log C = -\max_j f_j$)

Softmax v.s. SVM

SVM gets correct score to be greater than a margin above the incorrect scores.

Softmax interprets the score as (unnormalized) log probabilities for each class and then encourages the (normalized) log probabilities of correct class to be high.

SVM不在乎具体in score，只要正确in score高出错误in一定margin就满足，但 softmax classifier is never fully happy with the scores，因为正确in score总会有更高in probability（趋近于1）.

Optimization

Visualizing the loss function.

1. generate random W_1 (direction), W_2, W.
2. evaluating $L(W + aW_1 + bW_2)$ for different values of a and b
3. plot the L with a, b as axis x and y.

for 1 dimension, just a.

Once we extend the score functions f to Neural Networks, the objective functions will become non-convex, and the visualizations will not feature bowls but complex bumpy terrains. (subgradient exits and commonly used.)

Strategy #1. Random search (bad idea)

Strategy #2. Random Local Search (wasteful & and computationally expensive)
start out with random W. generate random perturbation δW
to it find if $(W + \delta W)$'s loss is lower.

Strategy #3. Following the Gradient.

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

$$\nabla_W L_i \longrightarrow \text{analytic gradient}$$

Practical: Always use analytic gradient, but check implementation with numerical gradient. (gradient check)
Compute the analytic gradient and compare it to the numerical to check the correctness.

Gradient Descent loop {1. compute gradient
2. update the parameters with learning rate × gradient}

Mini-batch gradient descent, compute the gradient over batches of training data.
size: 32/64/128/256...

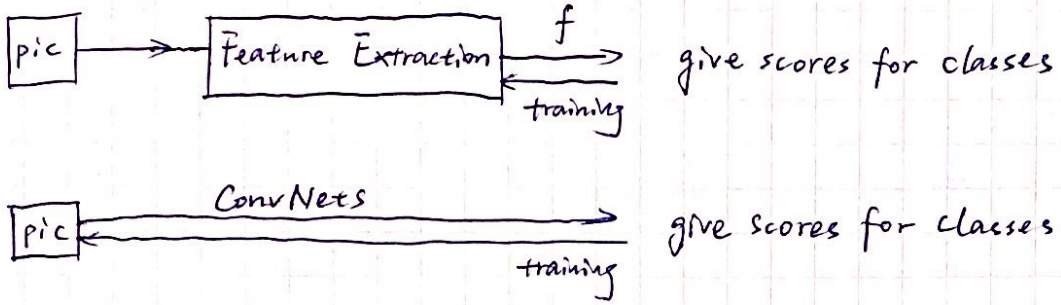
Stochastic gradient descent. One example each time to compute the gradient.

other gradient descent methods: - gd with momentum.
- Adam optimizer

Image Features → apply feature transform to make "points" separable by classifiers

- color Histogram
- ~~Motivation~~
- Histogram of Oriented Gradients (HoG)
- Bag of Words

Image features vs. ConvNets



Rather than writing down the features ahead of time, ConvNets learn the features directly from the data.

Further Reading

< Deep Learning using Linear Support Vector Machines >

Some intuitions

classification → $L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1)$, $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$

Large number of classes → Hierarchical Softmax. 分解 labels 成一棵 tree. 每个 label 都是树的一个 path. 在节点上训练 softmax 作为分类.

Attribute classification → y_i 是一个 binary vector, 每个 example 可能有多个 attributes

$L_i = \sum_j \max(0, 1 - y_{ij} f_j)$, $L_i = \sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))$

$\frac{\partial L_i}{\partial f_j} = y_{ij} - \sigma(f_j)$

Regression → $L_i = \|f - y_i\|_2^2$ (L2), $L_i = \|f - y_i\|_1$ (L1)

L2 Loss 比更稳定的 loss (如 softmax) 更难被优化. ① L2 Loss 需要非常服从和特定的网络属性来输出精确的正确值. for each input. 而 softmax 不同每个 score 的准确值没有那么重要. score 的量合适比较重要. ② L2 Loss 对 outliers 敏感.

所以当面对 regression 任务时, 先考虑是否绝对必须. 取而代之的是更倾向于离散化输出然后 perform classification. (如 AlexNet 的 loss function 为 softmax 多分类)

Structured prediction → Structured SVM Loss (Labels are arbitrary structures, graphs, trees...)

Lecture 4 Backpropagation and Neural Networks

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

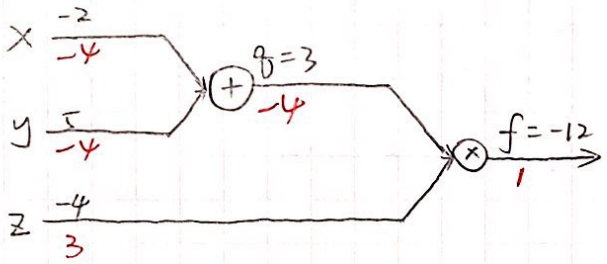
The derivative on each variable tells the sensitivity of the whole expression on its value.

Chain rule

$$z = f(y) \quad y = g(x)$$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

exp:



Backpropagation

Every gate in a circuit diagram gets some inputs and can right away compute two things: 1. its output value

2. the local gradient of its output with respect to its input value.

During backpropagation, we can get the gradient of output on the inputs of entire circuit by the chain rule.

The gates are relatively arbitary. Any kind of differentiable function can act as a gate.

Patterns in backward flow

add gate: gradient distributor.
takes the gradient on its output and distributes it equally to all of its inputs.

max gate: gradient router
distributes the gradient to exactly one of its inputs (the input that had the highest value during forward pass)

multiply gate: gradient switcher.

Gradients for vectorized operations

Tip: ① explicitly write out a minimal vectorized example, derive the gradient on paper and then generalize the pattern to its efficient, vectorized form.

② The gradient with respect to a variable should have the same shape as the variable.
Because each element of the gradient is quantifying how much that element is contributing to the final output.

Lecture 5 Convolutional Neural Networks

Differences with ordinary Neural Networks

- ConvNet architectures make the explicit assumption that the inputs are images, which allow us to encode certain properties into the architecture. (preserve spatial structure).
- forward function more efficient to implement.
  reduce the amount of parameters in the network.

A ConvNets is made up of layers. Every layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function with or without parameters.

A simple ConvNet for CIFAR-10 classification:

INPUT — CONV — RELU — POOL — FC

Convolutional Layer

Overview and Intuition

- The CONV layer's parameters consis of a set of learnable filters.
- Every filter is small spatially (along width and height), but extends through the full depth of the input volume. (e.g. 5x5x3)
- During forward pass, we slide (convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position.
- Several 2-dimensional activation maps is generated to give the responses of filter at every spatial position.
- The network will learn filters that activate when they see some type of visual features. (edge, blotch, honeycomb or wheel-like patterns)

□ Local Connectivity.

— Each neuron will only connected to a local region of the input volume.

— The spacial extent of local connectivity is a hyperparameter called receptive field of the neuron (= filter size)

— The extend of the connectivity along the depth axis is always equal to the depth of the input volume.

□ Spatial Arrangment.

Three hyperparameters control the size of the output volume: depth, stride and zero-padding.

— depth, the number of filters, each learning to look for sth different in the input.
depth column (fibre), a set of neurons that are looking at the same region of the input.

— stride, how many pixels will be skipped when we move the filters.
1 or 2 is commonly used. Larger stride will produce smaller output volumes spatially.

— zero-padding, pad the input volume with zeros around the border.
exactly preserve the spatial size of the input volume.

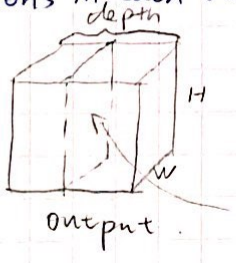
Input volume size ($W \times W$), the receptive field size (F), the stride (S), the amount of zero-padding (P), number of folters (K)

the size of output = $(W - F + 2P)/S + 1$

$P = (F-1)/2$ $S = 1$ ↓ input volume and output have the same size

The Conv Layer:
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters, K, F, S, P
- produces a volume of size $W_2 \times H_2 \times D_2$, where
  $W_2 = (W_1 - F + 2P)/S + 1$, $H_2 = (H_1 - F + 2P)/S + 1$, $D_2 = K$.
- with parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1)K$ weights and K biases.

□ Parameter Sharing: the neurons in each depth slice use the same weights and biases



depth = K

depth slice

The parameter sharing assumption may not make sense when the input images to a ConvNet have some specific centered structure. (different features should be learned on one side of the image than another.)

one practical example → faces centered in the image.

Instead of parameter sharing → Locally-Connected Layer

☐ 1×1 convolution < Network In Network>

☐ Dilated convolutions < Multi-Scale Context Aggregation by Dilated Convolutions>

dilation 0 $W[0]\times X[0] + W[1]\times X[1] + W[2]\times X[2]$

dilation 1 $W[0]\times X[0] + W[1]\times X[2] + W[2]\times X[4]$

merge spatial information across the inputs much more agressively with fewer layers.

Pooling Layer

reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting.

Typical form: 2×2 filter max pooling, stride = 2

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters: F, S
- Produces a volume of size $W_2 \times H_2 \times D_2$

$W_2 = (W_1 - F)/S + 1$ $\quad D_2 = D_1$

$H_2 = (H_1 - F)/S + 1$

- Introduce zero parameters since it computes a fix function
- commonly no zero padding

☐ Getting rid of pooling. < Striving for Simplicity: All Convolutional Net>

To reduce the size of representation using larger stride in Conv layer instead of pooling.

Discarding pooling layers has been found to be important in training good generative models (variational autoencoders. VAEs, generative adversarial networks. GANs)

Converting FC layers to CONV layers

Fully-connected

The only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. 连接稀疏性 (2×2 ↔ 1×1×1×1) 参数共享

Both layers compute dot products.

- For any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing)
- Any FC layer can be converted to a CONV layer. For example, an FC layer with K=4096 that is looking at 7×7×512 input can be equivalently expressed as a CONV layer with F=7, P=0, S=1, K=4096. The output will be 1×1×4096. (filter size = input volume)

实际应用中将 FC 转变为 CONV层更加有用.

假设一个卷积神经网络的输入是 224×224×3. 在一系列卷积层和池化层之后 activation 的大小为 7×7×512. (在 AlexNet 中是这样. 有5个池化层, 每次下降一半. 224/32=7). 一个 AlexNet 用了两个尺寸为 4096 的 FC, 最后一个有 1000 个神经元的 FC 用于计算分类评分.

可将这3个 FC layer 中任意一个转化为 CONV layer:

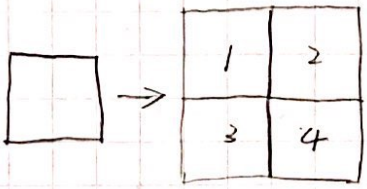
FC1. 7×7×512 → 1×1×4096. filter 尺寸为 7 K=4096

FC2. F=1 K=4096

FC3. F=1 K=1000.

每次这样的变换需要把 FC 的 weight 重塑成 CONV layer 的滤波器.

将 FC 转化为 CONV 层的作用: 当让卷积网络(带FC)在一层更大的输入图片上滑动(即把一张更大的图的不同区域分别带入到有FC的卷积网络, 得到每个区域的得分), 得到多个输出. 这样的转化可以在单个转化后的卷积网络向前传播中完成.



例：上 224×224 的窗口以步长 32 在 384×384 的图片滑动，把每个经停的位置都带入有 FC 的 ConvNet，得到 6×6 个位置的类别得分。

⇕ 等价

在 384×384 大图上直接经过"把 FC 转换为 Conv layer"的卷积网络向前传播。

224×224 → 7×7×512 （5层池化） —FC→ 1×1×1000

384×384 → 12×12×512 （5层池化）

↓ 3个卷积层

6×6×1000

Summary

→ 相比"滑动"的方法，使用转化过的卷积网络计算效率高得多。（只有1次向前传播，滑动需要计算36次，转化后相当于36次计算共享计算资源）。

→ 实践中，通常将一张图尺寸变大，然后使用变换后 ConvNet 对空间不同位置进行评估得到分类得分，再求平均值。

当在大图上以 2^{Pool} 进行滑动时，等同于~~在大图上以步长为1直接卷积~~由在大图直接卷积得到的activation再次以步长为1卷积。

将原FC转变为 Conv layer.

→ 若使用小于 2^{Pool} 步长滑窗，需要多次向前传播。

先在原图用原步长向前传播一次，再分别沿宽，沿高，最后沿宽高，把原图平移一个步长个像素，把平移后六图分别带入向前传播。（转化后的 ConvNet）

以原步长

S=32，16，8…

ConvNet Architectures

最常见的卷积神经网络：

Input → [[Conv → RELU]×N → Pool?]×M → [FC → RELU]×K → FC

其中×号表示重复次数，Pool? 表示一个可选的汇聚层，其中 N≥0，通常 N≤3，M≥0，K≥0 通常 K<3

常见结构模式：

- Input → FC，实现一个线性分类器，N=M=K=0
- Input → CONV → RELU → FC
- Input → [CONV → RELU → POOL]×2 → FC → RELU → FC，每个汇聚层间有一个卷积层
- Input → [CONV → RELU → CONV → RELU → POOL]×3 → [FC → RELU]×2 → FC

每个汇聚层前有2个卷积层，这种思路适用于更大更深网络，因为在执行具有破坏性的 Pool 前，多重的卷积层可以从输入数据中发现更为复杂特征。

Intuition:

一些小滤波器卷积层的组合比一个大滤波器卷积层要好.

例如. 有3个3×3的卷积层. 第一层 Conv layer 的神经元对 Input 都有3×3的 view, 第二层卷积层每个神经元对第一层有3×3的 view, 即对输入有5×5的 view, 第三层的每个神经元对输入有7×7的 view. 这样与单独7×7感受野的卷积层的感受野一样. 但有

大 filter 缺点: ① 多个卷积层含非线性的激活层交替的结构, 相比单一卷积层的结构更能提取出深层的更好特征,

② 假设所有数据有C个通道(深度为C), 单独的7×7卷积层有 $C\times(7\times7\times C)=49C^2$ 个参数, 而3个3×3卷积层的组合有 $C\times(3\times3\times C)\times3=27C^2$ 个参数. 小滤波器卷积层的组合参数更少.

大 filter 优点: 进行反向传播时, 需要用多个小滤波器的卷积层, 可能会占用更多内存去存放中间层的结果.

Recent Departures: 传统的将层按线性排列的方法受到了谷歌 Inception 和微软亚洲的残差网络(Residual Net)的挑战.

Layer Sizing Patterns

- Input layer. 应该可以被2整除多次, 如 CIFAR 的 32, 64, 96 (STL-10), 224, 384, 512.
- 卷积层. 应使用小滤波器(3×3, 最多5×5), 步长 S=1. 重要的是对输入使 padding. 当 F=3 时 P=1 能保持输入尺寸, 当 F=5, P=2. 对于一般的 F, 当 $P=(F-1)/2$ 时能保持输入尺寸. 若必须使用更大的 F, 如 7×7, 通常只用在第一个面对原始图像的卷积层上.
- Pooling layer: downsampling the spatial dimentions of the input. 通常用 2×2 的感受野(F=2)最大值汇聚, 步长为2. 会丢掉输入的75%.

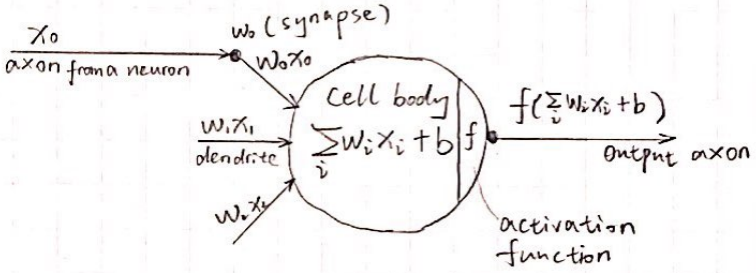
如果使用步长大于1的 filter 和卷积层且不使 padding, 也能达到 downsampling 的效果, 但是需要仔细监督数据通过卷积神经网络的过程, 确保所有步长和 filter 尺寸都吻合. 有时需要对因为内存限制做妥协. 实际践中倾向于在第一个卷积层使用. 如 AlexNet 第一个卷积层使用步长为4, 11×11的滤波器.

Case studies :

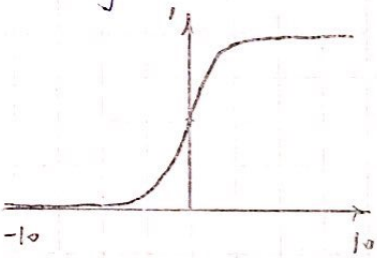
- LeNet
- AlexNet
- ZF Net
- GoogleNet
- VGG Net
- ResNet

Lecture 6. Training Neural Networks I

Activation Functions



— Sigmoid



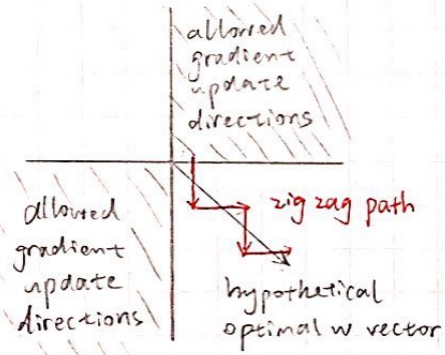
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

缺点：1. Saturated neurons "kill" the gradients

当 x 很大或很负时，梯度趋近于 0，backprop 中无法得到梯度流的反馈.

2. Sigmoid outputs are not zero-centered.

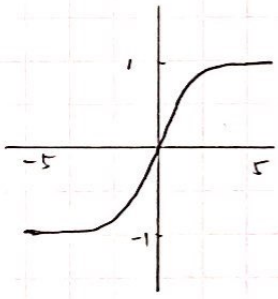
Sigmoid 输入到下一个 neuron 的输入恒为正，则此 neuron weights 的 gradients 恒为正或负. 在更新 weights 时会有 zigzagging 收敛低效.



当一个 batch 中 data 的 gradients 加起来时最终 w 的更新可以有不同的正负. 缓解了不是 zero-centered 的问题. 但不太方便.

3. exp() is a bit compute expensive.

— tanh(x)



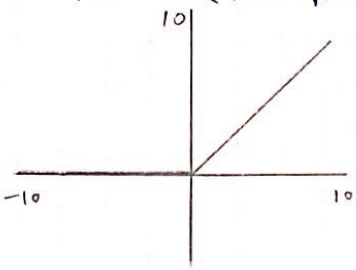
$$\tanh(x) = 2\sigma(2x) - 1$$

优点：tanh 是 zero-centered.

缺点：gradients saturation.

实际应用中 tanh 优先于 sigmoid.

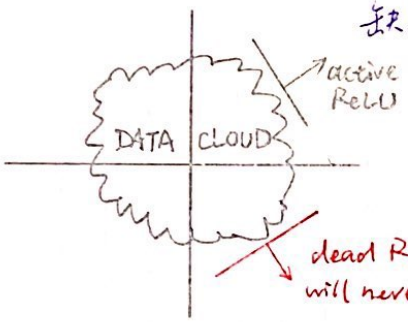
— ReLU (Rectified Linear Unit)



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

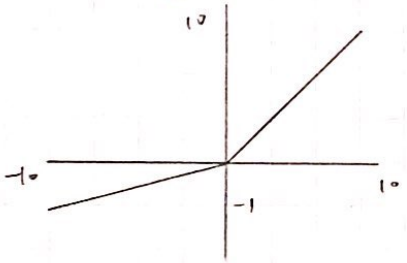
优点：1. Does not saturate in + region
2. very computationally efficient
3. Converges much faster than sigmoid/tanh (e.g. 6x) AlexNet

缺点：1. Not zero-centered output
2. 被置0的 data 将不会再更新. 可能的原因是 learning rate 过大 (40% of the network can be dead)



缓解方法：initialize ReLU neurons with slightly positive biases (e.g. 0.01) 增加在初始化时被激活的可能性并得到更新; proper setting of the Lr.

— Leaky ReLU

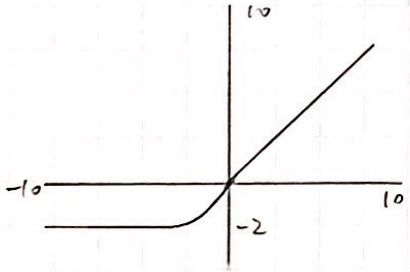


$f(x) = \max(\alpha x, x)$ α is a small positive constant.
α 也可以由学习得到.

Leaky ReLU 拥有 ReLU 的优点. 并解决了被置0之后不会再更新的问题.

The consistency of the benefit across tasks is presently unclear.

— ELU (Exponential Linear Units)



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

优点：1. all benefits of ReLU
2. closer to zero mean outputs
3. Negative saturation regime compared with Leaky ReLU, adds some robustness to noise.

缺点：computation requires exp()

— Maxout

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

优点：1. generalizes ReLU and Leaky ReLU
有ReLU的优点，也不会有dying ReLU的问题
2. Linear Regime. Does not saturate.

缺点：doubles the number of parameters/neuron.

Summary

- 用ReLU，注意 Learning rates 和网络中"dead" units所占的比例.
- 尝试 Leaky ReLU, Maxout 和 ELU，如果 dead units 过多.
- 尝试 tanh，但一般不会比 ReLU/Maxout 有效.
- 不要使用 sigmoid.

Representational Power

Universality theorem: Neural networks with a single hidden layer can be used to approximate any continuous function to any desired precision.

实践中，通常3层的神经网络比2层表现好，4,5,6层等更深的神经网络提升不大.
相反，深度对于CNN非常重要，更深的CNN表现更好.可能的原因是图象包含多种分层的结构信息(如脸由眼、嘴等构成，它们都有edge).

神经网络大小的选择：不要因为害怕overfitting而选用小的神经网络.(更小的NN有更少的local minima，但是更容易收敛且loss更大.训练一个小的NN，最终loss含有很大的variance.即有时loss很小有时loss很大)

应该在计算资源允许的情况下选用更大的神经网络，然后使用如L2 regularization, dropout, input noise等正则化方式来控制overfitting.
(更大的NN，会找到更多不同的local minima，这些结果loss的variance很小，rely less on the luck of random initialization)

Data Preprocessing

— Mean subtraction

对每个数据减其对应的特征的mean. 从几何上可解释为将 data cloud 居中到原点. `X -= np.mean(X, axis=0)`

特别地对于图象. 通常减去所有 pixel 的 mean. 或对 RGB channel 分别 mean subtraction. `X-= np.mean(X)`

— Normalization.

归一化数据使其在大约相同的度量里.(范围)

一种方式是将经过 zero-centered 的数据除以标准差. 在每个 dimension 上.

`X/= np.std(X, axis=0)`

另一种方式是分别将最大值和最小值对应 1 和 -1.

归一化仅在不同输入的特征有不同 in scales 才有意义. 对于图象而言, pixels 的相对 scales 大致相同 (0~255), 通常不必要做归一化.

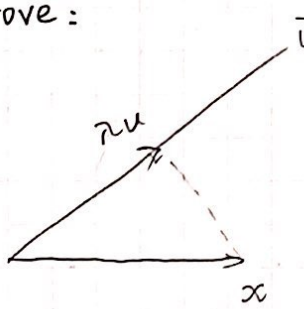
— PCA and Whitening

$$\frac{X^TX}{n} = VS^2V^T$$

```
X -= np.mean(X, axis=0)                  # zero-center X (N×D)
cov = np.dot(X.T, X)/X.shape[0]          # get the data covariance matrix
U,S,V = np.linalg.svd(cov)               # SVD. U为eigenvectors(因为XᵀX对称半正定)
                                         #      S为eigenvalues,从大到小排列
Xrot = np.dot(X, U)                      # project the original zero-centered
Xrot_reduced = np.dot(X, U[:,:100])      #   data into the eigenbasis.
```

PCA 将原始数据在其 eigenbasis 上投影而达到降低维度和最大化数据间 variance (尽多地保留信息) 的目的

Prove:



PCA maximizes the projection to receive information as much as possible, when $\vec{u}$ is the eigenvector to guarantee the new x has the largest variance.

vector:

$$(x-\lambda u)^T u = 0$$
$$\lambda = x^T u$$

matrix:

$$N\begin{bmatrix} \overset{D}{(x^{(1)})^T} \\ X \end{bmatrix}$$

$$(X^T - u\lambda)^T u = 0$$
$$\lambda = Xu$$

当 u 是 unit vt. λ 为投影大小

最大化投影大小

↓

maximize $(Xu)^T Xu$

$= u^T X^T X u$.

maximize the variance of the projections

$$\Updownarrow$$

$$\text{maximize } \frac{1}{m}\sum_{i=1}^{m}(x^{(i)T}u)^2 = \frac{1}{m}\sum_{i=1}^{m} u^T x^{(i)} x^{(i)T} u \qquad \begin{pmatrix} m: \text{sample number} \\ x, u: \text{vector} \end{pmatrix}$$

$$= u^T\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)T}\right)u$$

$$= u^T \Sigma u$$

$$\underset{D}{\begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(N)} \\ | & & | \end{bmatrix}}_{D\times N} \cdot \underset{N}{\begin{bmatrix} -x^{(1)T}- \\ \vdots \\ -x^{(N)T}- \end{bmatrix}}_{N\times D} = \sum_{i=0}^{N} x^{(i)}x^{(i)T} = X^T X$$

X^T $\qquad$ X

Lagrange: $\max u^T\Sigma u$ s.t. $u^Tu=1$ $\Rightarrow$

$$\mathcal{L}(u,\lambda) = u^T\Sigma u - \lambda(u^Tu-1)$$

$$d\mathcal{L} = (du)^T\Sigma u + u^T\Sigma du - \lambda du^T\cdot u - \lambda u^T du$$

$$tr(d\mathcal{L}) = 2\,tr(u^T\Sigma - \lambda u^T)du$$

(Σ is symmetric)

$$\frac{\partial \mathcal{L}}{\partial u} = 2(\Sigma u - \lambda u) \overset{\text{set}}{=\!=} 0$$

$$\Sigma u = \lambda u$$

So, if we wish to find a 1-dimensional subspace to approximate the data, u should be the principal eigenvector of Σ.

Generally, if we wish to project our data into a k-dimensional subspace, we should choose $u_1, \dots, u_k$ to be the top k eigenvectors of Σ.

$$\underset{N\times D}{\begin{bmatrix} -x^{(1)T}- \\ \vdots \\ -x^{(N)T}- \end{bmatrix}} \cdot \underset{D\times k}{\begin{bmatrix} | & & | \\ u_1 & \cdots & u_k \\ | & & | \end{bmatrix}} = \underset{N\times k}{\begin{bmatrix} x^{(1)T}u_1 & x^{(1)T}u_2 & \cdots \\ \vdots & & \end{bmatrix}}$$

X $\qquad$ U

Whitening operation takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale.

$$X_{white} = X_{rot} \,/\, np.sqrt(S + 1e\text{-}5)$$

↓

prevent dividing by zero, but will exaggerate the noise in the data

mitigated by stronger smoothing (e.g. increasing 1e-5 to be a larger number)

— Data preprocessing in practice

1. 对于图象数据(CNN)的 zero centering 比较重要，通常不做 PCA 和 whitening 或 normalization.

2. 任何预处理的统计数字必须只在 training data 里计算，然后应用于 vali/test data.

Weight Initialization

— all zero initialization.

错误的做法. 若所有 W 都从 0 为初始，每个神经元在输入数据的基础上有相同的操作，因为它们将输出相同的值，得到相同的梯度，从而以相同的方式更新. (no source of asymmetry between neurons)

— small random numbers 不一定会 work strictly better

W = 0.01 * np.random.randn(D, H) → 从 N(0,1) 中生成随机数

使用小的随机数(zero mean)初始化会使 activations 变得很小趋近于 0，从而在 backprop 时梯度也很小，基本无更新. 对于更深的网络影响更大.

用更大的随机数可能有饱和的风险.

— Calibrating the variances with 1/sqrt(n)

使用小随机数(np.random.randn)的问题在于：从随机初始化的 neuron 的输出的 variance 随输出个数的增大而增大 ↓

内积 $s = \sum_{i=0}^{n} w_i x_i$

$Var(s) = Var(\sum_{i=0}^{n} w_i x_i)$

$= \sum_{i=0}^{n} Var(w_i x_i)$ ← 充分不必要条件：$w_1x_1, w_2x_2, \dots, w_nx_n$ are independent

w_i, x_i are independent →

$= \sum_{i=0}^{n} [E(w_i)]^2 Var(x_i) + [E(x_i)]^2 Var(w_i) + Var(x_i) \cdot Var(w_i)$

$= \sum_{i=0}^{n} Var(x_i) Var(w_i)$ ← inputs and weights are zero-mean

$= (n\, Var(w)) Var(x)$ ← assume w_i and x_i are identically distributed.

Review:

$Var(X) = E[(X-\mu)^2]$

$Var(X) = cov(X, X)$

$Var(X) = E(X^2) - E(X)^2$

$\forall X, Y$ are independant

$Var(XY) = [E(X)]^2 Var(Y) + [E(Y)]^2 Var(X) + Var(X)Var(Y)$

$= E(X^2)E(Y^2) - [E(X)]^2[E(Y)]^2$

$Var(aX) = a^2 Var(X)$

初始化时
如果想保证S和输入X有相同的variance，因为 $Var(aX) = a^2 Var(X)$，初始化的 weights 需要除以 $\sqrt{\frac{1}{n}}$

`w = np.random.randn(n) / sqrt(n)`

similar initialization → $Var(w) = 2/(n_{in} + n_{out})$ ← 同时考虑 backprop 的情况，此时输入为 forward 时的输出.

number of units in the previous layer and the next

initialization for ReLU → `w = np.random.randn(n) * sqrt(2.0/n)`

— Sparse initialization.

Another way to address the uncalibrated variances problem is to set all weight matrices to zero, but to break symmetry every neuron is randomly connected to a fixed number of neurons below it. (A typical number may be as small as 10)

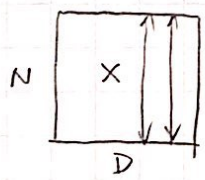
— Initializing the baises

It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights.

— Batch Normalization

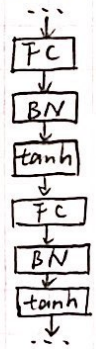
基本思想："you want zero-mean unit-variance activations? just make them so"

位于 Fully connected 或 Convolutional Layer 之后且在 nonlinearity 之前.



分别对每个 dimension 计算 empirical mean 和 variance.

再做归一化. $\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$



Input: Values of x over a mini-batch: $B = \{x_{1...m}\}$

Parameters to be Learned: γ, β.

Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean

$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ // mini-batch variance

$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$ // normalize

$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$ // scale and shift

Intuition:

— 提升了网络中的 gradient flow

— 允许更高的 Learning rate

— 降低了对于初始化的依赖

— 某种意义上的 regularization，并降低了对 dropout 的需求
(因为每层输出都来源于输入 x 以及 batch 中被采样的其它样本的 μ 和 σ^2)

— 引入了 γ 和 β 的作用在于增加了关于全归一化和饱合之间的 flexibility.

— Training 的 μ_B 和 σ_B 由当前 batch 算出. 而 Testing 时的 μ_B 和 σ_B 应使用 Training 时保存或类似经过处理的值

— BN 中所有操作都平滑可导，使得 Backprop 可以有效运行并学到相应的 γ 和 β

Lecture 7. Training Neural Networks II

Regularization

— L2 regularization $(\frac{1}{2}\lambda w^2)$

diffuse weight ↓

L2正则化趋向于鼓励网络使用所有input, a little. 而不是使用某些input a lot.
在参数更新时. 最终L2正则会将weights趋近于0. (W += -lambda * W).
对x的改变不敏感. 但对于outlier敏感. (对噪声的鲁棒性)
(L2正则对于NN的意义可能不那么明确. 有时会在NN中选择其它方案)

— L1 regularization $(\lambda|w|)$

L1正则鼓励稀疏(sparse). 使用L1的网络最终只会使用稀疏的部分输入. 对噪声输入基本不变.
实际中. 如果不关心明确的特征选择. L2正则有望比L1正则更好.

— Elastic net regularization

combine the L1 and L2 regularization. $(\lambda_1|w| + \lambda_2 w^2)$
通常比Lasso(L1)表现好. 同时保留了相似的稀疏性. 鼓励grouping effect (非常相关的特征趋向于一起在或不在模型中).

— Max norm constraints

对权重的大小设置上限. ($\|w\|_2 < c$) 需要时对w进行: $w \leftarrow w \cdot \frac{c}{\|w\|_2}$
当learning rate太高时. 使用max-norm不会使网络"explode". 因为上限是被限制的.

— Dropout

Dropout指的是在forward pass时, 按一定概率p. 随机保留或丢弃neurons.

Interpretation:
1. Forces the network to have a redundant representation.
   Prevents co-adaptation of features.
2. Dropout is training a large ensemble of models (that share parameters)
   Each binary mask is one model

在test或prediction时不做dropout. 但要对hidden layers的activations做概率为p的缩放. 以保证testing时neurons的输出和training时的期望值一致.

- Inverted Dropout

为了保证test或predict更有效率. 不在test时做和p的乘法, 而在training时. 在每一次dropout时除以p. 还有个好处在于可以选择部分做dropout而不是所有.
使用dropout通常训练会花更长时间, 因为每个step只更新网络的子部分

- Dropout 对比 Batch Normalization.

  BN中一个数据点可能出现在不同的 mini batches中. 对单个数据点来说在训练中如何被正则化是有随机性的. 所以 BN 在训练时引入了某种随机性或噪声. 又在 test 时中和 (average) 掉. 也是一种正则化的方式.

  当用 BN 在训练网络时, 正则化效果有时足够好而不需要 dropout, 但 dropout 的优点在于可调整正则化的强度. 通过调整 P 来达到. 而 BN 没有.

- DropConnect.

  DropConnect sets a randomly selected of weights with the network to zero. Each unit thus receives input from a random subset of units in the previous layer.

- Data Augmentation

  通过对有限的数据进行一定的变换生成新数据, 防止过拟合.

  变换方式包括 (Horizontal Flips, Random crops and scales, Color Jitter, PCA Jittering translation, rotation, stretching, shearing, lens distortions ...)

- Fractional Max Pooling

  随机部分 max pooling.

- Seochastic Depth.

  在训练过程中, 随机丢弃部分 Layers.

以上包括 Dropout 在内的正则化方法都得益于(training时)引入了一定的随机性, 而在test时. 随机性带来的 noise 会被解析地(dropout 采以 p) 或数字地(随机采样完成多次 forward passes 最后取平均) 忽略, 中和.

Summary

通常不必要对 Bias 作正则化 以及不必要对每层 layer 使不同强度的正则化.

实践中常见的是单一的全局 L2 正则并使用 cross-validation. 常用和 dropout 一起使用.

$p=0.5$ 是默认的best, 可以在 validation 时 tune.

- The recommended preprocessing is to center the data to have mean of zero, and normalize its scale to [-1 1] along each feature.
- Initialize the weights by drawing them from guassian distribution with standard deviation of $\sqrt{\frac{2}{n}}$, where n is the number of inputs to the neuron.
- use L2 regularization and dropout (the inverted version)
- use batch normalization.

Learning

— Gradient Checks

In theory, performing a gradient check is as simple as comparing the analytic gradient to the numerical gradient.

- Use the centered formula. (numerical gradient)

$$\frac{df(x)}{dx} = \frac{f(x+h)-f(x)}{h} \quad \text{(bad, don't use)}$$

实践中 $h \sim 10^{-5}$，更多使用 centered difference formula.

$$\frac{df(x)}{dx} = \frac{f(x+h)-f(x-h)}{2h} \quad \text{(use instead)}$$

直观来讲，这个式子评估了 loss function 2次来检查每一个维度的梯度，这个近似更为准确. 使用 Taylor expansion 展开 $f(x+h)$ 和 $f(x-h)$ 会发现第一个式子的误差为 $O(h)$，第二个式子误差为 $O(h^2)$.

prove: 令 $f(x)$ 在 $x=a$ 处展开: $f(x) = f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2} + f'''(a)\frac{(x-a)^3}{6} + \underbrace{O(x-a)^3}$

把拉格朗日余项 $R(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1}$ $(a<\xi<x)$

将 $x=a+h$ 代入

$$f(a+h) = f(a) + f'(a)h + f''(a)\frac{h^2}{2} + f'''(a)\frac{h^3}{6} + O(h^3)$$

将 $x=a-h$ 代入

$$f(a-h) = f(a) - f'(a)h + f''(a)\frac{h^2}{2} - f'''(a)\frac{h^3}{6} - O(h^3)$$

$$\therefore \frac{f(a+h)-f(a)}{h} = f'(a) + \underline{f''(a)\frac{h}{2} + f'''(a)\frac{h^2}{6} + O(h^2)}.$$

$$= f'(a) + O(h)$$

$$\frac{f(a+h)-f(a-h)}{2h} = f'(a) + f'''(a)\frac{h^2}{6} + O(h^2)$$

$$= f'(a) + O(h^2)$$

- Use relative error for the comparison.

比较 numerical gradient f_n' 和 analytic gradient f_a' 的差来追踪 gradient check 的正确性.

$$\text{relative error} = \frac{|f_a' - f_n'|}{\max(|f_a'|, |f_n'|)}$$

注意当 f_a' 和 f_n' 都为 0 时，判断 pass gradient check.

实践中的判断：

相对误差 $> 10^{-2}$ → gradient probably wrong

10^{-2} 和相对误差 $> 10^{-4}$ → should make you feel uncomfortable

$10^{-4} >$ 相对误差 → okey for objectives with kinks.

too high for no kinks (e.g. tanh and softmax)

$10^{-7} >$ 相对误差 → happy.

越深的网络，相对误差会越高 (errors build up on the way)

如果 10 层的网络误差为 10^{-2}，可能会 ok. 但是一层的误差 10^{-2} 肯定是不正确的.

aside:

L2 error check

$$\frac{\|f_a' - f_n'\|_2}{\|f_a\|_2 + \|f_n\|_2}$$

- Use double precision

  计算 Gradient check 时最好使用双精度. 因为经常会在梯度正确时因为仅用了单精度 float 而使 error 大到 1e-2.

- Stick around active range of floating point

  如果梯度大小 (1e-10) 可能需要临时 scale loss function up by a constant 来达到 nicer range where floats are more dense. (ideally on the order of 1.0)

- kinks in the objective.

  kinks 指的是 non-differentiable parts of an objective function. 如 ReLU, SVM loss, Maxout neurons 等等.

  部分的不可导可能带来误差. 如 ReLU 中, $x = 1e-6$, 此时梯度应该为 0. 但 numerical gradient 可能不为 0. 因为 $f(x+h)$ 当 $h > 1e-6$ 时, 提供了一个不为 0 的 contribution.

  检测 kinks 被跨跃的方法: 评估 $f(x+h)$ 和 $f(x-h)$ 时, 若 $\max(x, y)$ 结果的选择发生了变化, 则发生了 kinks cross. 此时的 numerical gradient 不准确.

- Use only few datapoints.

  只用很少的数据来做 gradient check 也可以减轻 kinks 的影响.

  2, 3 个 data points 对于一整个 batch 的 gradient check 已经足够 (faster and efficient)

- Be careful with the step size h.

  h 不宜过小, 过小的 h 会导致精度问题 (1e-6 ~ 1e-4 可能比较合适)

- Gradcheck during a "characteristic" mode of operation.

  梯度检验是在参数空间上特定一点 (randomly) 做 perform 的. 不能立即确定这点的正确性就是全局的. 如在 SVM 中以很小的值初始化 weight. 所有的 score 会近 0. 则所有点的 gradient 有相同的特征, 这种情况不能 generalize 到更一般的梯度有大有小的情况.

  所以安全的做法是给定一个 short burn-in time. 即等 loss 开始下降时开始做梯度检验. 不要在第一次迭代时就做 gradient check.

- Don't let the regularization overwhelm the data.

  通常 loss function 是 data loss 和 regularization loss 的和. 可能的错误是正则化的 loss 会覆盖 data loss. 推荐的做法是先关掉正则化, 检查 data loss, 再单独检查正则项.

  单独检查正则项可以去掉 data loss 项, 或加大 regularization strength. 来保证确定它对 gradient check 有大的影响. 那么 an incorrect implementation would be spotted.

- Remember to turn off dropout / augmentations.

  当执行梯度检测时, 保证关闭所有网络中的不确定影响 (deterministic effect).

  不然这些因素会带来很多 error. 在做 numerical 的时候.

  关闭这些影响的糟糕动作是如无法强迫 dropout 的梯度 (dropout 可能无法正确 backprop)

  更好的方法是确定一个特定的 random seed. 再计算 $f(x+h)$ 和 $f(x-h)$, 再计算 analytic gradient.

- Check only few dimensions

实践中，梯度可能有数百万个参数，Le的实际的做法是检验某些 dimensions 并确保梯度检验到 ~~the dimensions for~~ 每一个单独的参数的一些 dimensions.

Before Learning : sanity checks Tips/Tricks

- Look for correct loss at chance performance

确保在初始化后，得到正确的 loss. 最好先检验没有正则项，只有 data loss.

如 CIFAR-10 的初始 loss 应为 $-\ln(0.1) = 2.302$

Weston Watkins SVM 初始 loss 为 9（每个错误的 class 的 margin 为 1）

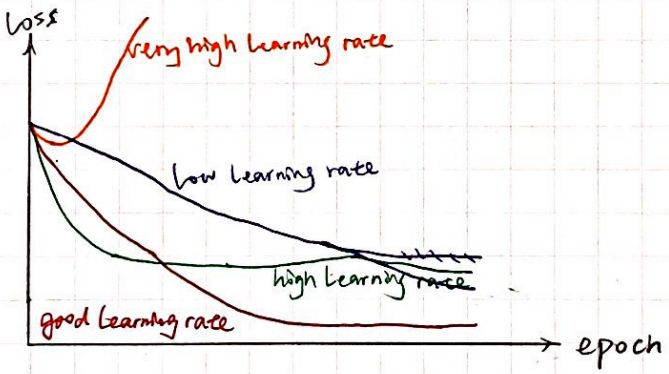
- Second sanity check, 增加正则的 strength 应该增加 loss.
- Overfit a tiny subset of data.

在训练所有数据之前，先拿出一小部分 (e.g. 20 examples) 训练，确保可以达到 0 cost. (需要先关闭正则项)

Babysitting the Learning process

one epoch means that every examples has been seen once. 更倾向于追踪 epochs 而不是 iterations. 因为迭代次数取决于 batch 的大小.

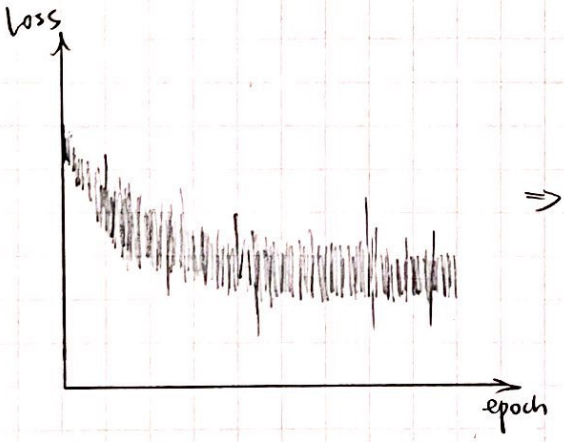
— Loss function.



Low learning rates → improvements will be linear

high learning rates → improvements look more exponential

~~da~~ decay fast, but get stuck at worse value (due to parameters bouncing around chaotically)

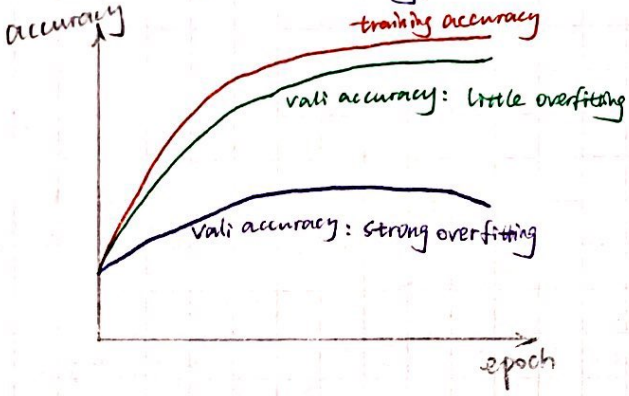


⇒ { Loss function looks reasonable
maybe small learning rate
batch size small (noisy cost)

loss 上的 "wiggle" 表示 batch size. ~~wiggle~~

batch size 越大，wiggle 越小.

— Train/Val accuracy



- training and validation 的准确度中的 gap 表明了 overfitting 的程度
- → 当 validation 的准确度和 training 的 gap 较大时为严重的过拟合，需要增加正则化或得到更多的数据。
- → 当 validation accuracy 跟随 training accuracy 相当好时(绿色)，此时的 model 能力可能不够，需要增加模型的复杂度。

— Ratio of weights: updates.

track weights 更新量和原值量的比值。(比值在在 1e-3 左右) 如果过小则 learning rate 小，如果过大则 learning rate 过大。(可比较 max, min 或 norm 等)

```
param_scale = np.linalg.norm(W.ravel())
update = -lr*dW      # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update
print(update_scale / param_scale).
```

W.ravel() 矩阵降维
≈ W.flatten()
flatten 返回 copy
ravel 返回 view.

— Activation / Gradient distributions per layer

诊断 activation 和 Gradient. plot histograms for all layers. 可以直观地看到 activation 和 gradient 的分布，判断是否异常。

— First-layer Visualization

plot 出 first layer features 可以直观地观察出训练的状态。

- o plot 出的 weights 看上去模糊：有 noise，网络不收敛，learning rate 设置不好，依正则化。
- o plot 的 weights 清晰，干净：多种 features 在 plot 中是好的迹象，表明运行正常。

Parameter updates.

— SGD and bells and whistles

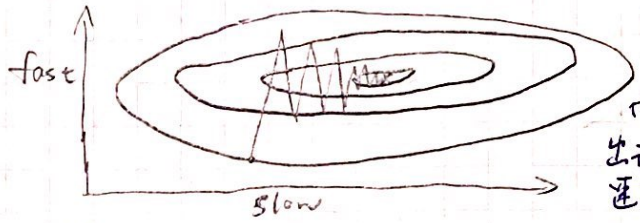
- o Vanilla update

最简单的更新方式是沿着负梯度的方向更新(因为梯度表示着增长的方向，我们往往希望最小化 loss function)

$$x \mathrel{+}= -\text{learning\_rate} * dx.$$

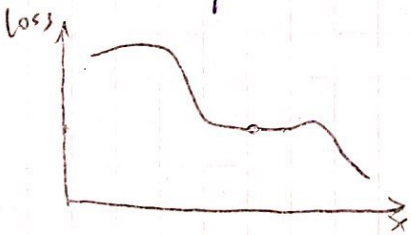
- Momentum update

单纯SGD的一些缺陷：1. taco shell problem

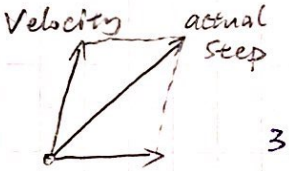


Hessian matrix 中最大值和最小值(singular value)之间差距较大.
梯度在某个方向上下降很快，在某方向上下降很慢，于是梯度下降时会出现这样 zig-zag 的情况，降低收敛速度.

2. saddle point.



当遇到局部最小或鞍点时，梯度会变为0，此时SGD的梯度下降会停止.
在高维中，saddle point 的现象更为普遍，因为一个点上的许多维度的下降方向很可能被彼此抵消.



3. noise.

SGD的梯度来源于 mini-batch的 estimation. 梯度更新时会显得 noisy. 导致需要花更多的时间来达到 optimum.

解决缺陷的动量法：

类比下山的过程，用随机初始化相当于从0速度在某处放置一个质点.

方法一
```
V = rho * V - Learning_rate * dx
x += V
```

V初始化为0. rho = [0.5, 0.9, 0.95, 0.99] 可以理解为摩擦力，如果没有 rho，则质点永远不会停止运动.

类似于 Learning rate 的"退火"机制，典型的设置是 rho 开始为 0.5，在更新过程中逐渐增大到 0.99.

方法二
```
V = rho*V + dx
x -= learning_rate * V
```

with Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient.

方法三 $V = \rho V + (1-\rho)dx$ → 和上面方法等效，只不过同时缩小了 1-rho（注意V以0为起点，不影响结论）

$x -= learning\_rate * V$

✓ - Nesterov Momentum

在更新计算梯度时，可以把 (x + rho*V) 当作即将要达到点附近的预估. 取 (x+rho*V) 点处的梯度作为更新.


```
x_ahead = X + roho * V
V = rho * V - learning_rate * dx_ahead
X += V
```

$V_{t+1} = \rho V_t - \alpha \nabla f(\tilde{x}_t + \rho V_t)$

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

实践中我们不想更新 x_ahead 处的梯度，我们要一直存"ahead version"，想让表达式类似于 SGD 和 momentum，用 x 替代 x_ahead.

```
V_prev = V
V = rho * V - learning_rate * dx
X += - rho * V_prev + (1+rho) * V
```

$\tilde{x}_t = x_t + \rho V_t$

$\tilde{x}_{t+1} - \rho V_{t+1} = \tilde{x}_t - \rho V_t + V_{t+1}$

$\tilde{x}_{t+1} = \tilde{x}_t - \rho V_t + (1+\rho) V_{t+1}$

$V_{t+1} = \rho V_t - \alpha \nabla f(\tilde{x}_t)$

—— Annealing the learning rate

- Step decay

  减小 learning rate by some factor every few epochs. (by half every 5 epochs, or 0.1 every 20 epochs).

  One heuristic → 观察 validation error, 使用 fixed learning rate 训练, 当 validation error 不再变化时再减小 learning rate (如 0.1倍)

- Exponential decay.

  $\alpha = \alpha_0 e^{-kt}$, α_0 和 k 是 hyperparameters. t 可以是 iteration number 或 units of epochs

- $1/t$ decay

  $\alpha = \alpha_0/(1+kt)$, α_0, k, t 同上.

实践中 slightly perfer Step decay. 其超参数可解释性比 α_0, k 更强; 如果计算资源允许, 尝试使用更慢的 decay 并训练更长时间.

—— Second order methods.

Hessian matrix (a square matrix of second-order partial derivatives of the function)

基于牛顿法来更新参数: $x \leftarrow x - [Hf(x)]^{-1}\nabla f(x)$

牛顿法是求 $f(x)=0$ 时的根. 利用牛顿法来找到 $f'(x)=0$ 时的根即找到了最值或鞍点. 这样的方法比梯度下降更加有效, 来以 Hessian matrix 的逆矩阵可以使得在低曲率的方向上得到更激进的 steps. 在陡曲率的方向上得到小的步长.

另外这种方法没有 Learning rate 这样的超参 (相比 first-order methods).

缺陷:

对于深度学习来说, 牛顿法有些不切实际.

1. 计算 Hessian matrix 的逆十分 costly. 若有 10^6 个参数, Hessian 的大小为 $[10^6 \times 10^6]$.
   → 有许多 quasi-Newton 方法来计算 Hessian matrix 逆的近似
   如最流行的 L-BFGS, 使用梯度变化的趋势来做近似
2. L-BFGS 必须在整个训练集上计算. 在 mini-batch 上计算 L-BFGS 目前仍是研究领域

实践中:

在大规模的 DL 中 Second-order methods 不太常见. 更多的使用的是 SGD 和一些 variants 基于 momentum.

—— Per-parameter adaptive learing rate methods

- AdaGrad

```
dx = compute_gradient(x)
grad_squared += dx * dx      # dx is a vector
x -= learning_rate * dx / (np.sqrt(grad_squared) + eps)
```

grad_squared 和 dx 的 size 相同并且 keep track of 每个参数的梯度的平方和.
然后用 grad_squared 来 normalize 参数更新项 (element-wise), 对高梯度的参数更新有在 learning rate 的削减作用, 对低梯度的参数更新有加快作用.
eps 项 (1e-4 ~ 1e-8) 用来防止除以 0.
(时间足够长, 更新的 stepsize -> 0)
Adagrad 的缺点在于, 对于 DL, 单调的学习速率通常被认为是过于激进和过早停止学习.

- RMSprop.

Exponentially weighted average.

```
dx = compute_gradient(x)
grad_squared = decay * grad_squared + (1 - decay) * dx * dx
x -= learning_rate * dx / (np.sqrt(grad_squared) + eps)
```

RMSprop 调整了 Adagrad, 减小了侵略性~~单调地降低~~的学习速率, 但其不会单调降低.
并且仍然保留了基于梯度大小调整学习速率.
decay 是一个超参 [0.9, 0.99, 0.999]

- Adam (Adaptive Moment Estimation)

```
m = beta1 * m + (1 - beta1) * dx
V = beta2 * V + (1 - beta2) * (dx**2)
x += - learning_rate * m / (np.sqrt(V) + eps)
```

recommendation:
esp = 1e-8
beta1 = 0.9
beta2 = 0.999
learning rate = 1e-3 or 5e-4

Adam 和 RMSprop 很像, 只是多了对 m 的更新方式.
Adam 是现在推荐的算法, 经常比 RMSprop 的表现好一些. 额外的, 尝试一下 SGD+ Nesterov Momentum.

完整的 Adam 还包含一个 bias correction 的机制. 在刚开始的几个 step 中, m, v 刚被初始化, 都为 0. 需加入一个 bias correction 来使 m 和 v 都可以初始化为 0, 使 m 和 v 更快 "warm up"

```
m = beta1 * m + (1 - beta1) * dx      -> Momentum
V = beta2 * V + (1 - beta2) * (dx**2) -> AdaGrad/RMSprop
mt = m / (1 - beta1**t)               } Bias correction
Vt = V / (1 - beta2**t)               } Bias correction
x += - learning_rate * mt / (np.sqrt(Vt) + eps)
```

t += 1 (每个 iteration +1)

Hyperparameter optimization

The most common hyperparameter in context of Neural Networks:

- the initial learning rate
- Learning rate decay schedule (such as the decay constant)
- regularization strength (L2 penalty, dropout strength).

— Implementation worker和master两个程序来做training

worker, continuously samples random hyperparameters and performs the optimization
在训练过程中，worker跟踪每个epoch后的validation结果，并把model checkpoint（多样的统计数据如loss over time）写进文件.

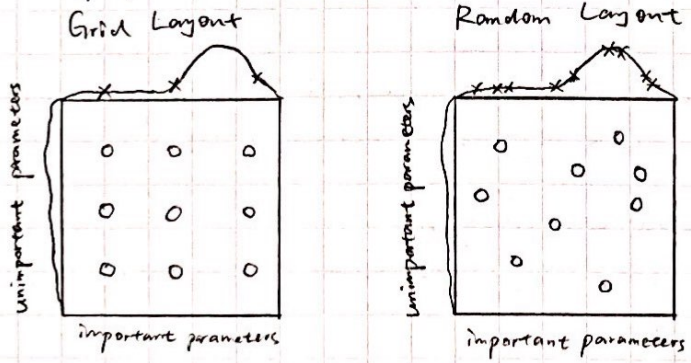
master, launches or kills workers across a computing cluster, and may additionally inspect the checkpoint written by workers and plot the statistics.

— Prefer one validation fold to cross-validation.
大多数情况一个validation set本质上简化了codebase，不需要cross-validation.

— Hyperparameter ranges

Search for hyperparameters on log scale. e.g. learning_rate = 10 ** uniform(-6, 1)
因为对于training dynamics，learning rate起到的是multiplicative effects.（regularization strength也相同）
一些其它参数则在uniform original scale上搜索，如 dropout = uniform(0, 1)

— Prefer random search to grid search.
随机搜索hyperparameter比网格搜索更有效，也更容易执行



— Careful with best values on border.
当得到超参的结果后，需要确认结果是否在区间的边界上，若在边界则很有可能错过了更优的超参数.

— Stage your search from coarse to fine

阶梯式地搜索超参数。一步一步缩小搜索范围（先以初始范围在1个epoch或更少的迭代次数里，因为很多超参数会导致根本不学习或很快explode；再缩小范围在5个epochs中搜索；最终在更长的时间里做更细节的搜索）。

— Bayesian Hyperparameter Optimization.

是在更有效地探索参数空间的一系列算法。核心思想是合理地平衡exploration和exploitation trade-off. when querying the performance a different hyperpara.

一些有名的 libraries：Spearmint, SMAC, Hyperopt.

但在实践中，对于 ConvNets 相对困难去在carefully-chosen的空间里击败random search.

Evaluation

Model Ensembles

实践中，Ensemble是一种可靠的可以提高a few percent performance的方式（如：训练多个独立模型，在test时average它们的结果）

A few approaches:

- Same model, different initializations.
  用 cross-validation 来确定最佳的超参数，再用这组超参数在随机初始化下训练多个模型。（The danger: the only variety is due to initialization）.
- Top models discovered during cross validation.
  使用交叉验证来确定最佳超参数，选取top few (如top 10) models来form ensemble.
  这种方式提高了variety但引入了suboptimal models. 更容易来实施，因为不需要额外的部署。
- Different checkpoints of a single model.
  如果训练非常expensive, some people have had limited success in taking different checkpoints of a single network overtime and using those to form an ensemble.
- Running average of parameters during training. (exponentially decaying sum)
  在内存中保存另一份参数，并在最后N个iterations叠加平均这些参数。最终使用"smoothed"参数。（almost always achieves better performance）

  Intuition: the objective is bowl-shaped and network is jumping around, so the average has a higher chance of being somewhere nearer the mode.

Ensemble的缺陷在于太多model在做prediction的时候会变得非常costly.

Geoff Hinton's idea: distill a good ensemble back to a single model by incorporating the ensemble log likelihoods into a modified objective.

Summary

- 对 implementation 使用小 batch 做 Gradient check 避免bug陷阱.
- Sanity check. 确保初始的 loss 是合理的. 并且可以用很少的数据达到 100% 正确率 (过拟合)
- 在训练时, 监视 loss, training/validation accuracy, fancier → updates量与 parameter value的比值 (should be ~ 1e-3), 在处理 Conv Nets 时的 first-layer weights.
- 2个推荐的 updates: SGD+Nesterov 或 Adam.
- 在训练时逐渐衰减 learning rate.
- 使用随机搜索来寻找超参数. 并 Stage the search from coarse to fine. (coarse: wide hyperpara ranges, train only for 1-5 epochs; fine: narrower rangers, many more epochs)
- Form model ensemble for extra performance.

Lecture 8 Hardware and Software

Deep Learning Hardware

CPU. fewer cores, but each core is much faster and much more capable.
great at sequential tasks.

GPU. More cores, but each core is much slower and "dumber"
great for parallel tasks.
典型的最适合GPU的算法 → 矩阵乘法.

Programming GPUs:
- CUDA (NVIDIA only)
  C-like code that runs directly on the GPU
  Optimized APIs: cuBLAS, cuFFT, cuDNN. etc.
- OpenCL
  similar to CUDA, but runs on anything.
  usually slower on NVIDIA hardware
- HIP.
  convert CUDA code to sth can run on AMD GPUs.

在处理 CPU/GPU communication 时需要注意不要被从存储器中读取数据的速度限制.

Solutions:
- Read all data into RAM.
- Use SSD instead of HDD.
- Use multiple CPU threads to prefetch data.

Deep Learning Software

— PyTorch.

- Tensor, like a numpy array, but can run on GPU
- Autograd, package for building computational graphs out of Tensors, and automatically computing gradients
- Module, a neural network layer; may store state or learnable weights.

Dynamic computation Graphs.

Building the graph and computing the graph happen at the same time.

— TensorFlow.

- First define computational graph
- Then run the graph many times.

Static Computation Graphs.

— Static v.s. Dynamic.

Static 适用于 Optimization 和 Serialization.

- static graphs framework can optimize the graph before it runs.
- Once the graph is built, can serialize it and run it without code that built the graph.

Dynamic 更适用于 Conditional 和 Loops.

- Pytorch 可以轻松实现条件判断语句和循环.
- TensorFlow 中所有计算都通过计算图实现. 条件判断和循环略 odd.
- Dynamic Graph Applications: Recurrent networks. Recursive networks. Modular networks.

PyTorch	TensorFlow
Dynamic Graphs	Static Graphs
Static: ONNX, Caffe2	Dynamic: Eager

PyTorch, - clean API.
- dynamic graphs make it very easy to develop and debug.
- Can build model in PyTorch then export to caffe 2 with ONNX for production/mobile.

TensorFlow, - safe (for most projects).
- Not perfect but has huge community, wide usage
- Same framework for research and production.
- Only choice to run on TPUs.

Lecture 9 CNN Architectures

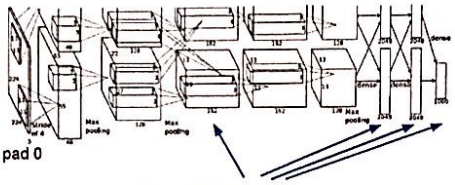
AlexNet.

- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation.
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus.
- L2 weight decay 5e-4
- 7 CNN ensemble : 18.2% → 15.4%

Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)



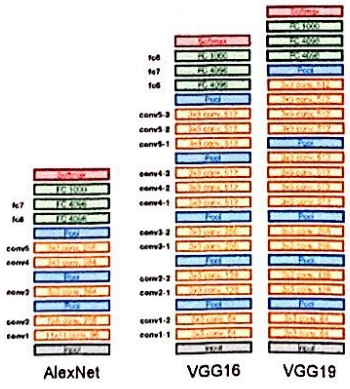
CONV3, FC6, FC7, FC8:
Connections with all feature maps in preceding layer, communication across GPUs

VGGNet

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Details:
- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



AlexNet VGG16 VGG19

- Smaller filters, only 3x3 conv stride 1, pad 1. 2x2 MaxPool, stride 2.
- Deeper networks.

使用小的filter的原因在于：
3个3x3 conv (stride 1)层的叠加和一个7x7 conv层的 effective receptive field是一样的.

① deeper, more non-linearities

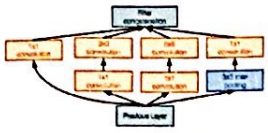
② fewer parameters.
$3\times(3\times3\times C)\times C < (7\times7\times C)\times C$
C for channels per layer.

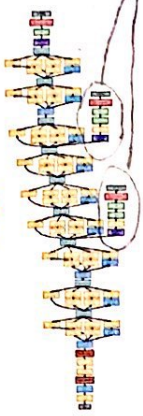
GoogLeNet

Case Study: GoogLeNet
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient "Inception" module
- No FC layers
- Only 5 million parameters! 12x less than AlexNet
- ILSVRC'14 classification winner (6.7% top 5 error)



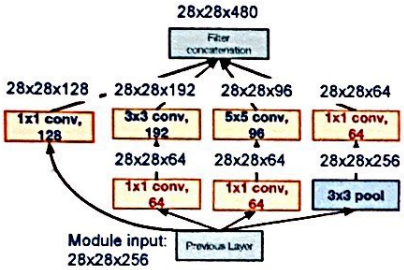


Inception module 采用了"network within a network"的思想.

将不同感受野的filter并行处理,包含1×1, 3×3, 5×5的convolution和3×3的Pooling.

最后再把输出连接(concatenate) depth-wise.

Case Study: GoogLeNet
[Szegedy et al., 2014]



Inception module with dimension reduction

Using same parallel layers as naive example, and adding "1x1 conv, 64 filter" bottlenecks:

spatial locations, filter size

Conv Ops:
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
Total: 358M ops

Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

单纯的多个conv layer并行会带来计算复杂度过大的问题.

解决方式引入"bottlenecks".

通过1×1的conv减少depth.

GoogLeNet还有2个额外的辅助分类输出.

因为当网络很深时, backprop时的一些梯度信号会被变小,并在靠近输入的时候丢失.

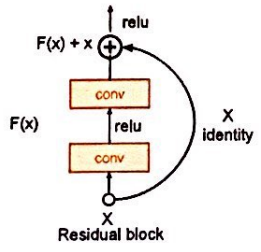
辅助分类输出提供了额外的梯度信号.

ResNet

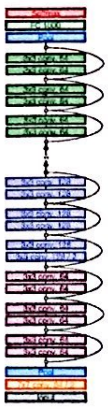
Case Study: ResNet
[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



$F(x) = H(x) - X$



问题点: 如果一直在"plain" CNN上堆更深的层, 更深的网络表现会更差. 并且不是由overfitting引起. training and test error都更差.

原因: hypothesis → 更深的网络更难被optimized.

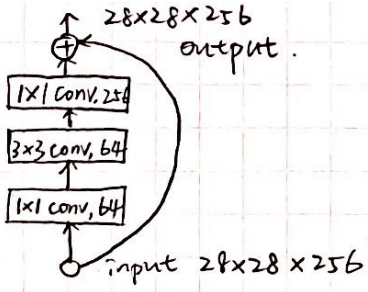
现论上,更深的网络应该至少和浅的网络表现一样好(深网络=浅网络+identity mapping layers)

Solution: 使网络来fit residual mapping而不是直接fit一个desired underlying mapping.

fit residual mapping, 我们只想学习到需要在输入x上添加或减去的部分.(H(x)-x)

完整的结构:
- stack residual blocks and each block has two 3×3 conv layers.
- Periodically, double # of filters and downsample spatially using stride 2. (/2 in each dimension)
- Additional conv layer at beginning
- Only one "FC 1000" layer at the end.

对于更深的网络（ResNet 50+），使用"bottleneck"来提高效率（similar to GooLeNet）

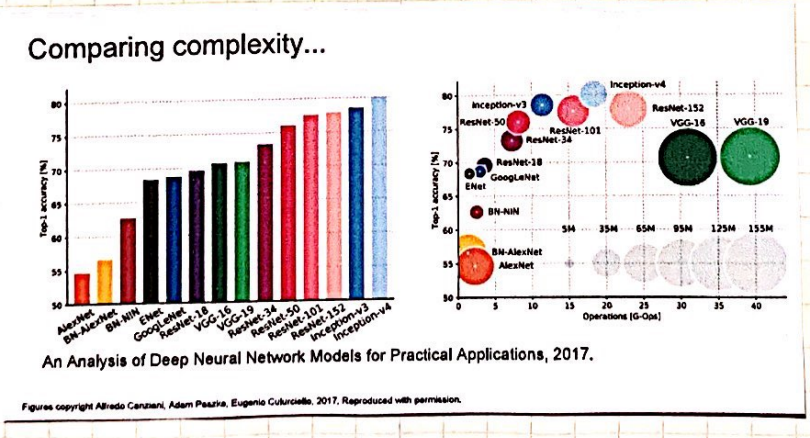


training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier 2/ initialization from He et al.
- SGD + Momentum 0.9.
- Learning rate 0.1, divided by 10 when validation error plateaus.
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used.

Summary

- VGG, GooLeNet, ResNet all in wide use, available in model zoos.
- ResNet current best default, also consider SENet when available.
- Trend towards extremely deep networks.
- Significant research centers around design of layer/skip connections and improving gradient flow.
- Efforts to investigate necessity of depth vs. width and residual connections.
- Even more recent trend towards meta-learning.



Inception-V4 : ResNet + Inception
VGG : Highest memory, most operations
GooLeNet : most efficient
AlexNet : Smaller compute, memory heavy, lower accuracy
ResNet : Moderate efficiency depending on model, highest accuracy.

Lecture 10 Recurrent Neural Networks

Network input and onput with application.

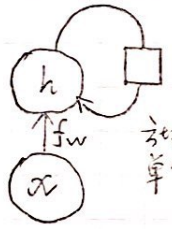
one to one : (input) (output) vanilla Neural Networks

one to many : Image Captioning. (image -> sequence of words).

many to one : Sentiment classification (sequence of words -> sentiment)

many to many: Machine Translation ; Video classification on frame level.

Recurrent Neural Networks.



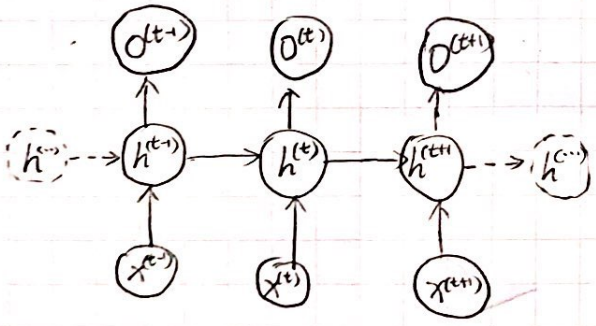
这个表示单个时间步的运算.

new state old state input at some time step.

$$h_t = f(h_{t-1}, x_t; W)$$

parameter (参数矩阵).

隐藏状态 h_t 汇总过去序列(直到t)与任务相关的有损摘要(lossy summary).
有损是因为网络映射任意长度的序列 $(x_1, x_2, x_3 \cdots x_t)$ 到一固定长度的向量 h_t.
根据不同的训练准则,摘要可能选择性地精确保留过去序列的某些信息.



$$\begin{cases} h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t) \\ O_t = W_{hy} h_t \end{cases}$$

循环神经网络的设计模式总结:

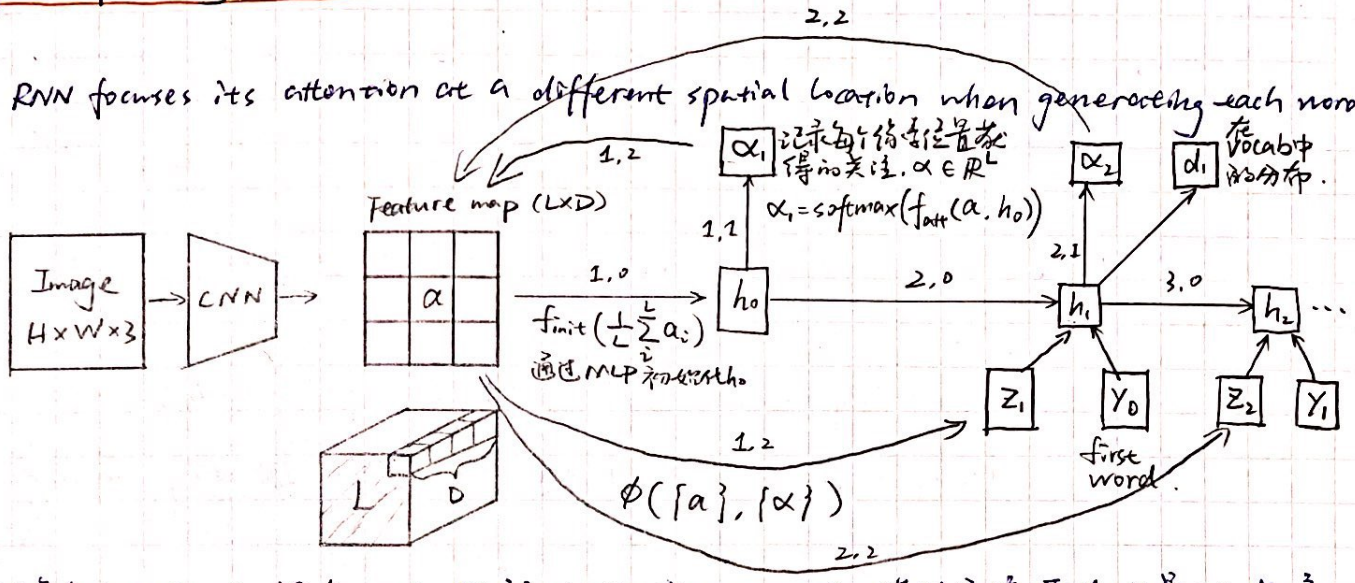
① 每个时间步都有输出,并且在隐藏单元之间有循环连接.(如左图)

② 每个时间步都有输出,只有当前时刻输出到下个时刻的隐藏单元之间有循环连接.

③ 读取整个序列后产生单个输出,且隐藏单元之间存在循环连接.

Image Captioning with Attention

RNN focuses its attention at a different spatial location when generating each word.



对于每个location i, $i \in L$, α_i 解释为用于产生下一个词, 模型应该聚焦位置 i 的概率.

$$e_{ti} = f_{att}(a_i, h_{t-1})$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{L} \exp(e_{tk})}$$

f_{att} 为 "attention model" (一个多层感知机 MLP).

$\hat{z}_t$ 为 context vector, $\hat{z}_t = \phi(\{a_i\}, \{\alpha_i\})$.

Intuition,

由 CNN 产生的 feature map 和隐藏单元 h_{t-1} (前一状态的) 产生下一个单词所应该关注的区域信息 α. 然后 α 再和 feature map 产生语境向量同词一起作为下一个隐藏单元的输入.

输出下一个词的 α 和 d (在 vocab 中的分布).

Long Short Term Memory (LSTM)

考虑vanilla RNN的梯度流，

$$\nabla_{h^{(t)}} L = W^T \operatorname{diag}(1-(h^{(t+1)})^2)\cdot \nabla_{h^{(t+1)}} L + V^T(\nabla_{o^{(t)}} L)$$

在计算 $\nabla_{h^{(t)}} L$ 时，会包含后一状态 $h^{(t+1)}$ 的梯度，意味着越初始的状态的梯度会乘越多的 W 相乘，导致：

对 W 而言 {
- 如果其最大的奇异值 >1，则会梯度爆炸 (exploding gradients)
- 如果其最小的奇异值 <1，则会梯度消失 (vanishing gradients)

}

奇异值为 W^TW 或 WW^T 的特征值的开方。特征值的求法为矩阵的行列式值，其表示经过该矩阵线性变换的体积缩放比率 (volume scaling factor)

→ 当梯度爆炸时可以通过限制 norm 的大小来防止：

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
    grad *= (threshold/grad_norm)
```

→ 当梯度消失时需要改变 RNN 的架构：LSTM.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

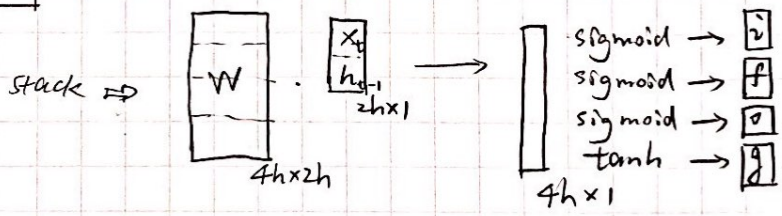
$$h_t = o \odot \tanh(c_t)$$

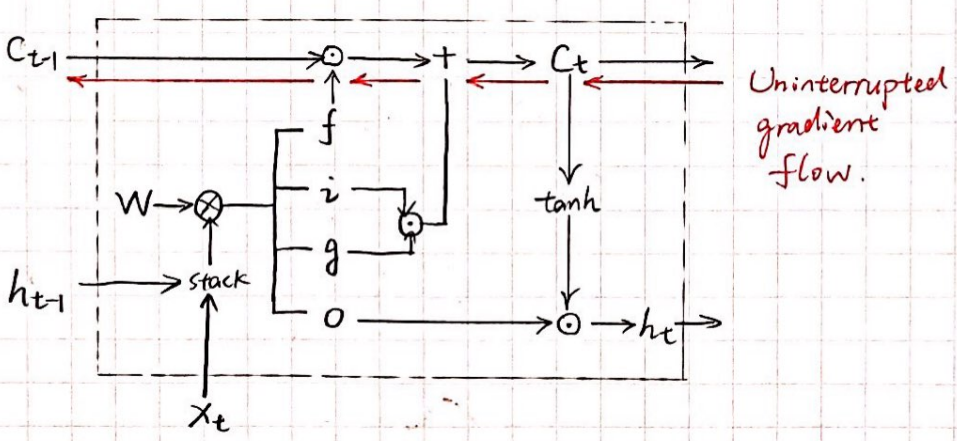
i, Input gate, whether to write to cell

f, Forget gete, whether to erase cell

o, Output gate, how much to reveal cell

g, Gate gate, (how much to write cell





Summary

- Common to use LSTM or GRU, their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish
  Exploding is controlled with gradient clipping.
  Vanishing is controlled with additive interactions (LSTM)

GRU:

$$r_t = \sigma(W_{xr} x_t + W_{hr} h_{t-1} + b_r) \rightarrow \text{复位门}$$

$$z_t = \sigma(W_{xz} x_t + W_{hz} h_{t-1} + b_z) \rightarrow \text{更新门}$$

$$\tilde{h}_t = \tanh(W_{xh} x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

更新门像条件泄漏累积器一样可以线性门控任意维度.

复位门控制当前状态中哪些部分用于计算下一目标状态。在过去状态和未来状态之间引入了附加的非线性效应.

Lecture 11 Detection and Segmentation

几种 CV task 的分类：

1. No objects, just pixels: Sematic Segmentation. 语义分割
2. Single object: Classification + Localization (图标分类并用 bounding box 标出)
3. Multiple objects: Object Detection (和单个物体类似用框标出，并分类)
   Instance Segmentation (在 OD 基础上 segment 出物体).
4. 3D object detection (object categories + 3D bounding boxes).

Semantic Segmentation

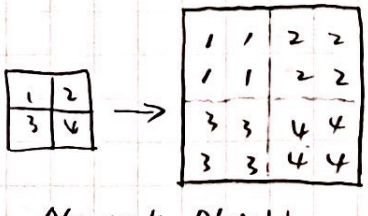
— Label each pixel in the image with category label.
Don't differentiate instances, only care about pixels. (任务描述)

— 一个比较原始的想法是，用一个 sliding window 在原图上滑动得到许多个 extract patch，然后将所有 patch 输入一个 CNN 分别判别所有 patch 的中心像素的 label.
问题是运算效率非常低效，没有利用上存在集 patches 之间共享的 features.

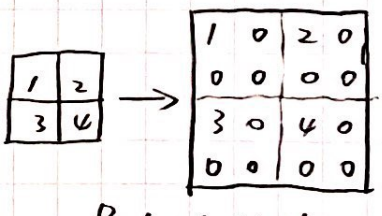
— Solution. Image → Conv NN → scores CxHxW → argmax → predictions. HxW

设计一个具有全像 Conv Layer 的网络预测每个 pixel. at once.
同时对原图做卷积可能会太 expensive. 所以 → 先 down sampling，再在网络后部 up sampling 回原来大小.

- Downsampling: pooling, Strided convolution.
- Upsampling:
  - Unpooling.

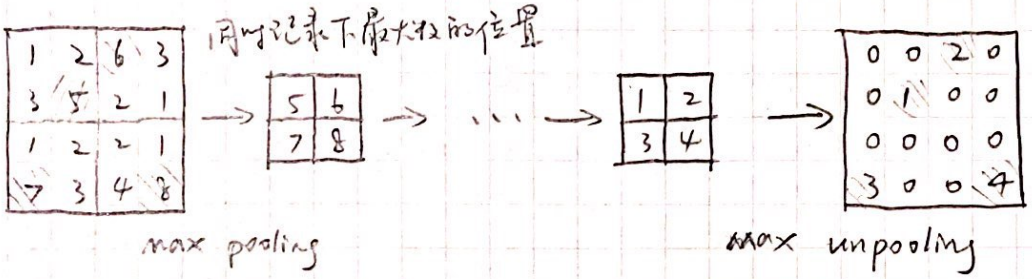


Nearest Neighbor

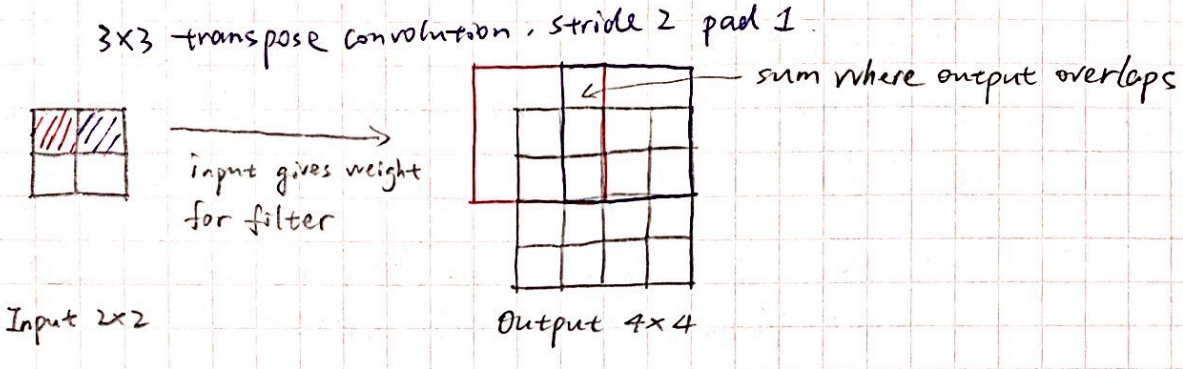


Bed of Nails

△ Max unpooling,



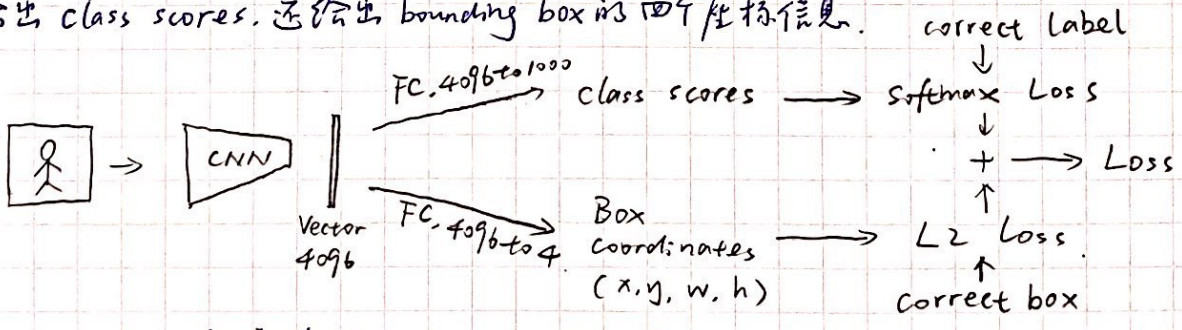
△ Transpose Convolution,



在数学操作上，transpose convolution 的 forward pass 和普通的卷积的 backward pass 是等价的.

2D Object Detection

对单个物体的 classification 和 Localization，可以训练一个 CNN，最终的 FC 层不仅给出 class scores，还给出 bounding box 的四个座标信息.



这时的 localization 就被当成了一个 regression 问题.

但当遇到多个物体时，localization 和 classification 会有多组输出，而且不同的图片物体数量不同，网络的输出个数也不同.（OD as regression）.

换一种思路，在图上使用 sliding window 选取 crops 输入 CNN 而得到这些 crops 的分类结果.

这样做也有问题，因为 crops 实在太多，非常 computationally expensive.（OD as classification）

沿着 object classification 的思路，可以设法减少 crops 的数量：

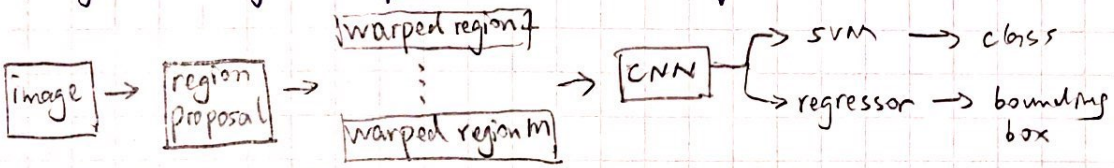
"Selective Search" 可以给出大约 2000 个 region proposals 在 CPU 计算的几秒钟内。

step 1, Generate inital sub-Segmentation. 通过粗略的分割方法得到每个 region 都属于什么物体

step 2, Recursively combine similar regions into larger ones.

step 3, use the generated regions to produce candidate object locations.

R-CNN



step 1, 在原图中选取 Regions of Interest (RoI)，通过一些选取方法如 Selective search

step 2, 将 RoI 变形成统一的形状

step 3, 将每个变形后的 RoI 输入 CNN，输出分类结果以及 bounding box。
对分类结果做 classification 优化，对 bounding box 做 linear regression.

problems: 1. RoI 是相对于训练集来说是固定的，通过某种选取方法，而不是通过学习得到。

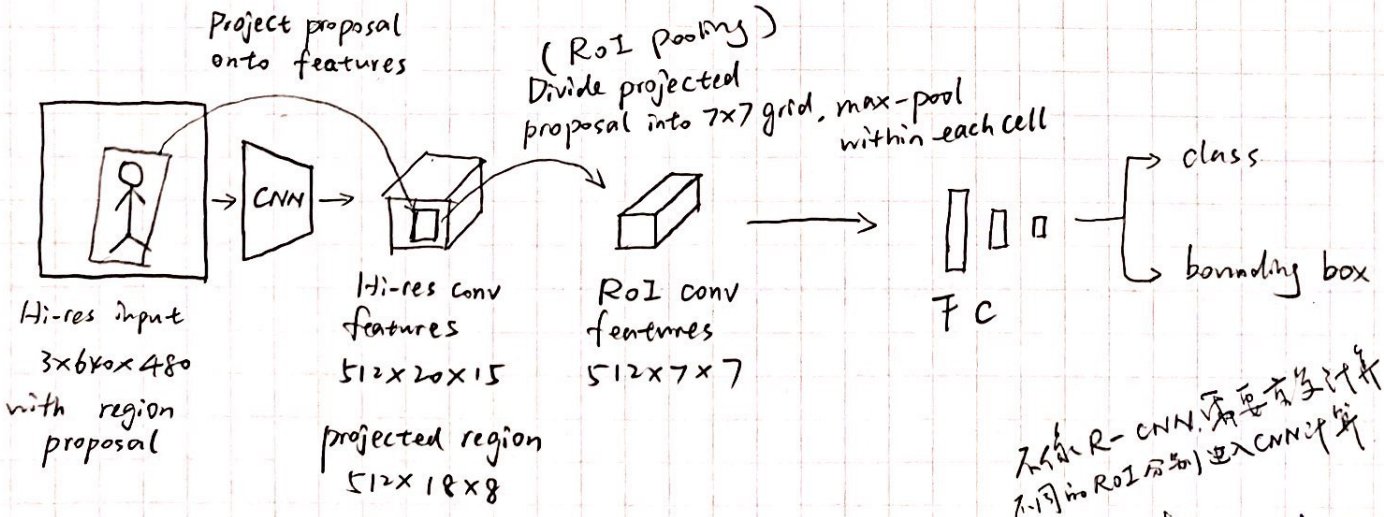
2. 训练的缓慢并且所占的存储空间也很大

3. 预测时间也很慢 (47s per image with VGG16)
(Inference) 需要在 2.3s 完成一次预测，即生成 RoIs 的时间就有 2s

Fast R-CNN

与 R-CNN 不同的是，Fast R-CNN 将整张图作为 CNN 的输入，得到一个 feature map.

再在 feature map 上映射 RoI，将不同的 crops 通过 FC 输出分类结果和 bounding box.



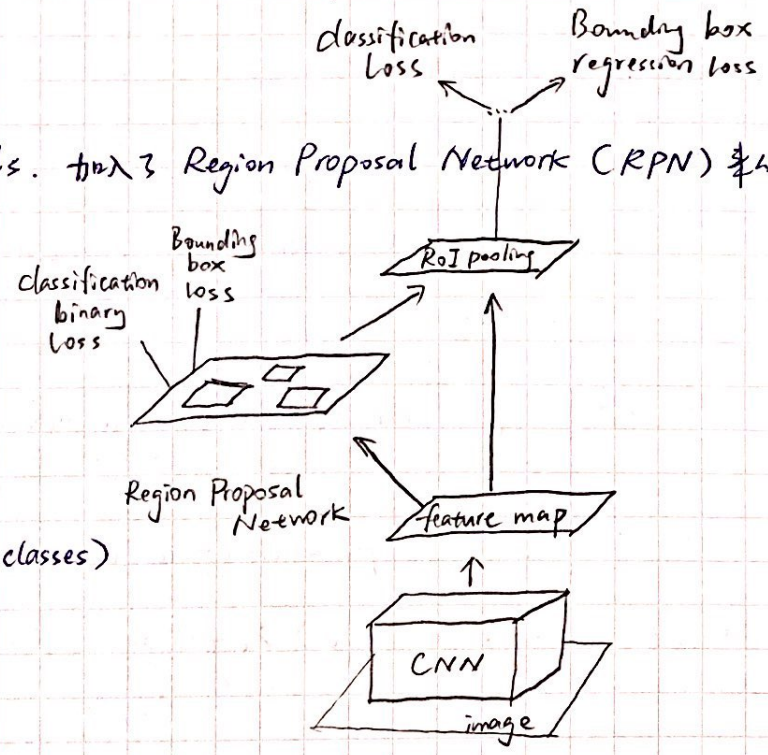
不像 R-CNN 需要分别计算不同 RoI 分别进入 CNN 计算

↓ (training time)

为什么 Fast R-CNN 要快？因为当选好 RoI 后，Fast R-CNN 对于所有 RoI 只需要做一次计算。

但是 Fast R-CNN 的速度还是被 RoI 的选取速度限制了 (run time).

Faster R-CNN



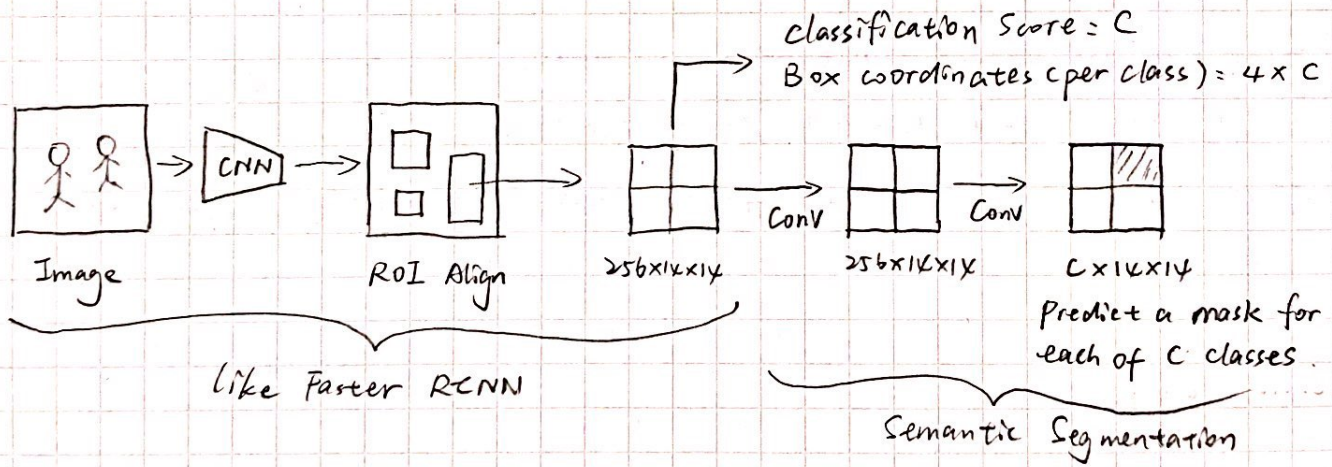
让 CNN 自己来找 region into proposals. 加入了 Region Proposal Network (RPN) 来从 features 中找出 proposals.

Jointly train with 4 losses

1. RPN classify object / not object
2. RPN regress box coordinates
3. Final classification score (object classes)
4. Final box coordinates

Mask R-CNN

for Instance Segmentation. and also does poses. estimation.



Mask R-CNN 的前段和 Faster R-CNN 相同，后段则产生额外的分支在所选区域中对像素进行分类，确定它是不是属于某个物体

Detection without Proposals

YOLO / SSD (You Only Look Once / Single-shot MultiBox Detector)

网络首先将图片分成 7×7 的 grid. 然后将一组 base boxes 分别放在每个 grid 的中间对于每一个 grid cell:

- Regress from each of the B base boxes to a final box with 5 numbers. (dx, dy, dh, dw, confidence)
- Predict scores for each of C classes (including background as a class)

output : 7×7×(5×B+C) 的 tensor.

Summary

Object Detection has a lot of variables. (Base network, Object detection architecture, Image size of region Proposals).

Faster R-CNN is slower but more accurate.
SSD is much faster but not as accurate.

paper: Speed/accuracy trade-offs for modern convolutional object detection.

[Pseudo-code]

- R-CNN.

```
RoIs = region_proposal ( image )
for RoI in RoIs:
    patch = get_patch ( image, RoI )
    results = detector ( patch )
```

run CNN.

— Fast R-CNN.

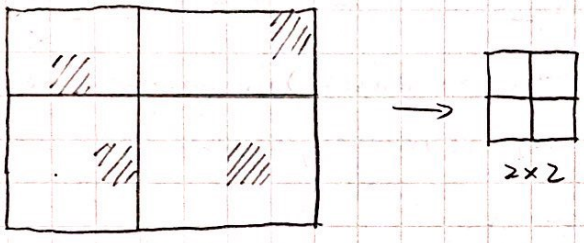
run CNN.(现在卷积网络在for循环外)
从而减少运算

```
feature_maps = process(image)
RoIs = region_proposal(image)  ← Expensive!
for RoI in RoIs:
    patch = roi_pooling(feature_maps, RoI)
    results = detector2(patch)
```

→ fully connected layer.

RoI pooling:

因为 Fast R-CNN使用的是FC层，所有从不同size的RoI需要map成相同大小。
做法是，将RoI划分成等同predefined的大小，在利用已域内取max.

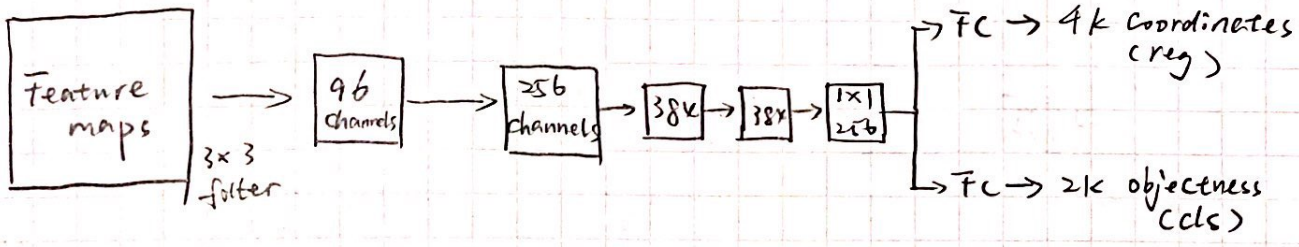


— Faster R-CNN

```
feature_maps = process(image)
RoIs = region_proposal_network(feature_maps)
for RoI in RoIs:
    patch = roi_pooling(feature_maps, RoI)
    class_scores, box = detector(patch)
    class_probabilities = softmax(class_scores)
```

与 Fast R-CNN 类似，但不同点在于引入了 RPN，并且 RoIs 的生成也从原图变为了 feature maps，从而取代了 Fast R-CNN中从原图产生 RoIs 的 region proposal methods.

Region proposal network (RPN)



- RPN用前一阶段的CNN输出的feature maps作为输入.
- 再使用3×3 filter的CNN (ZF network)提取特征, (也可以使用VGG或ResNet).
- ZF network输出的256个数分别进入2个FC. 1个预测 bounding box, 1个产生2个 objectness scores.

  Faster R-CNN中将其作为 classification问题处理. 即有物体类和无物体(背景)类.
  也可以简化为 regression 产生1个 objectness score.
- 对于每个feature maps中的每个位置(3×3 window中心), RPN都会产生K个预测.

  其中K是指K个固定形状(carefully pre-selected)的 anchor boxes.

  Anchor也被称作为 priors 或 default boundary boxes.

  Faster R-CNN中使用了9个anchors.

— Region-based Fully Convolutional Networks (R-FCN)

```
feature_maps = process(image)
    RoIs = region_proposal(feature_maps)
    score_maps = compute_score_map(feature_maps)
    for RoI in RoIs
        V = region_roi_pool(score_maps, RoI)
        class_scores, box = average(V)
        class_probabilities = softmax(class_scores)
```

Faster R-CNN需要对每个RoI通过FC计算结果. 对于很多的RoI, 计算量很巨很大.
R-FCN通过减少每个RoI需要的计算量来加快速度.

Lecture 12 Generative Models.

Supervised Learning v.s. Unsupervised Learning

	Supervised	Unsupervised
Data	(x, y)	x. ← training data is cheap
Goal	Learn a function to map $x \rightarrow y$	Learn some underlying hidden structure of the data.
Examples	Classification, regression, object detection, semantic segmentation, image captioning...	Clustering, dimensionality reduction, feature learning, density estimation...

Generative Models

— Give training data, generate new images (samples) from same distribution.

Generative models address density estimation, which is a core problem in unsupervised learning. (Learn $P_{model}(x)$ similar to $P_{data}(x)$).

- Explicit density estimation
  explicitly define and solve for $P_{model}(x)$
- Implicit density estimation.
  Learn model that can sample from $P_{model}(x)$ without explicitly defining it.

— Why generative models?

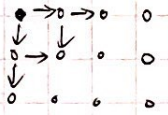
1. Realistic samples for artwork, super-resolution, colorization, etc.
2. Generative models of time-series data can be used for simulation and planning (reinforcement learning applications)
3. Training generative models can also enable inference of latent representations that can be useful as general features.

- Generative Models
  - Explicit density
    - Tractable density:
      - Fully Visible Belief Nets (FVBN)
        - NADE
        - MADE
        - PixelRNN/CNN
      - Change of variables models (nonlinear ICA)
    - Approximate density
      - Variational: Variational Autoencoder
      - Markov chain: Boltzmann Machine.
  - Implicit density →
    - Direct: GAN
    - Markov chain: GSN.

PixelRNN and PixelCNN

Use chain rule to decompose likelihood of an image x into product of 1-d distributions:

$$p(x) = \prod_{i=1}^{n} p(x_i | x_1, x_2 \dots x_{i-1})$$



PixelRNN, generate image pixels starting from corner

dependency on previous pixels modeled using an RNN (LSTM)

Drawback: sequential generation is slow.

PixelCNN, still generate image pixels starting from corner.

dependency on previous pixels now modeled using a CNN over context region.

Training is faster than PixelRNN (can parallelize convolutions since context region values known from training see)

Generation still sequential (slow).

VAE (Variational Autoencoders)

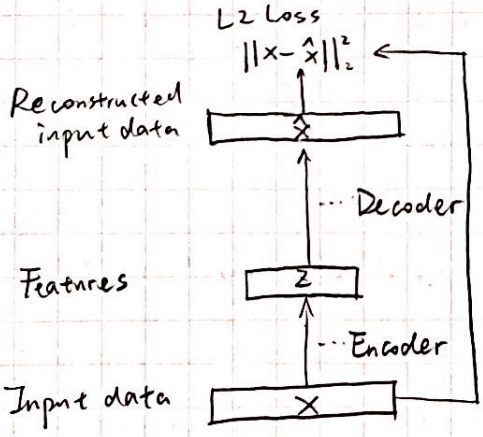
VAEs define intractable density function with latent z:

$$p_\theta(x) = \int p_\theta(z)\, p_\theta(x|z)\, dz$$

Cannot optimize directly, derive and optimize lower bound on likelihood instead.

— Auto-encoders.

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data.



通常z的维数要比x低，因为我们需要的是提取有意义的数据中变化的因素。

Encoder → CNN Decoder → CNN (upconv)

maximize the likelihood of training data.

训练完左侧结构的 autoencoder 之后，可以丢掉 decoder，来使用 encoder 作为 supervised model 的 initialization.

↓

这样做的价值在于我们可用许多无标签的训练数据得到很好的普适特征表达 (general feature representation)

用这些特征表达来初始化一个监督学习问题，适用于数据标注量较少的时候。

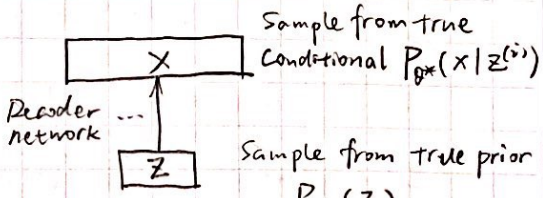
—— Variational Autoencoders

Autoencoders 可以重塑(reconstruct)输入图片并习到 features z.

Probabilistic spin on auto-encoders will let us sample from the model to generate data.

假设训练数据 $\{x^{(i)}\}_{i=1}^N$ 是从隐含的表征 z 中生成的.

我们想要的是来估计出 the true parameters θ^*.



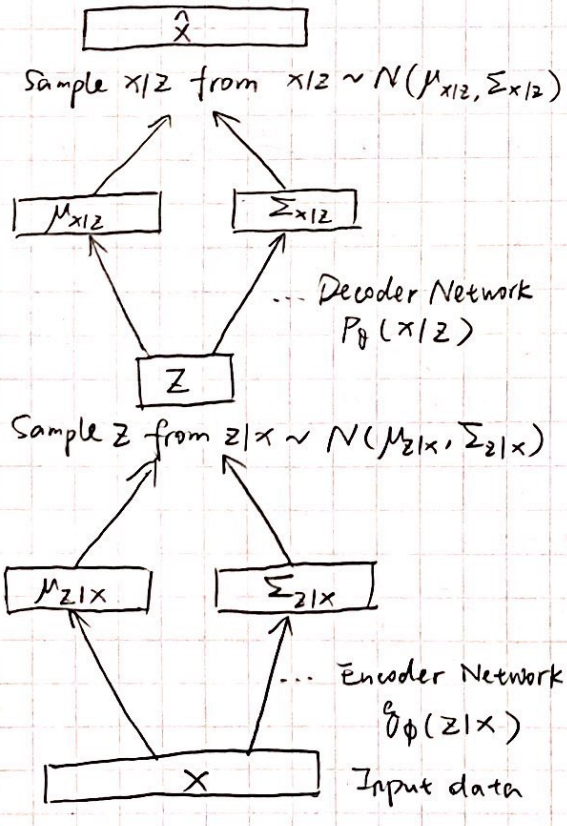
- 如何表示模型?
  1. 选择较简单的先验 $p(z)$, 如 Gaussian.
  2. $p(x|z)$ 较复杂, 用 neural network 表示.

- 如何训练模型? → Fully Visible Belief Networks

在 FVBN 中, 通过最大化 likelihood of training data 来训练模型. 但是这里的 $P_\theta(x)=\int P_\theta(x|z)P_\theta(z)dz$ 是无法直接优化的. 因为这个积分是 intractable 的.

Intractable to compute $p(x|z)$ for every z. 同时后验 $p(z|x)=p(x|z)p(z)/p(x)$ 也是 intractable, 因为不知道 $p(x)$.

Solution: 在 decoder network $P_\theta(x|z)$ 之外, 定义一个 encoder $q_\phi(z|x)$ 来近似 $P_\theta(z|x)$.



$$\log P_\theta(x^{(i)}) = E_{z\sim q_\phi(z|x^{(i)})}\left[\log P_\theta(x^{(i)})\right]$$ ← 可以选择任意的后面在 $P_\theta(x^{(i)})$ 不 depend on z; 这样的选择好处在于后面可以有 KL terms

$$= E_z\left[\log \frac{P_\theta(x^{(i)}|z)P_\theta(z)}{P_\theta(z|x^{(i)})}\right]$$

$$= E_z\left[\log \frac{P_\theta(x^{(i)}|z)P_\theta(z)}{P_\theta(z|x^{(i)})}\cdot\frac{q_\phi(z|x^{(i)})}{q_\phi(z|x^{(i)})}\right]$$

$$= E_z\left[\log P_\theta(x^{(i)}|z)\right] - E_z\left[\log \frac{q_\phi(z|x^{(i)})}{P_\theta(z)}\right] + E_z\left[\log \frac{q_\phi(z|x^{(i)})}{P_\theta(z|x^{(i)})}\right]$$

$$= \underbrace{E_z\left[\log P_\theta(x^{(i)}|z)\right]}_{①} - \underbrace{D_{KL}(q_\phi(z|x^{(i)})\|P_\theta(z))}_{②} + D_{KL}(q_\phi(z|x^{(i)})\|P_\theta(z|x^{(i)}))$$

$\mathcal{L}(x^{(i)},\theta,\phi)$

≥ 0 intractable 无法计算

$$\log P_\theta(x^{(i)}) \geq \mathcal{L}(x^{(i)},\theta,\phi)$$

Training: $\theta^*,\phi^* = \arg\max_{\theta,\phi}\sum_{i=1}^N \mathcal{L}(x^{(i)},\theta,\phi)$

maximize the lower bound.

⇓

① Maximize likelihood of original input being reconstructed

② Make approximate posterior distribution close to prior

① 计算可以由 sampling 得到估计 (sampling differentiable through reparam. "trick")

② 则是 encoder 的高斯分布和 z prior 的 KL 距离, 是 closed-forms.

- 如何生成 data?

只使用 decoder 来做数据的生成. z 的采样不同于训练时候从 $q_\phi(z|x)$ 中采取, 而是仅用我们指定的 prior, 如 $z \sim N(0, I)$.

z 表示的其实仅仅是隐含的特征. 如 z 的 prior 是 diagonal 的, 则表示互相独立的隐含变量. 不同的 z 的维度编码了可解释的变化因素. (维数也是人为规定)

类似于 Autoencoder, z 同样是很好的特征表征 (feature representation), 也可以仅用 encoder 部分输入图片生成 z 来用于 supervised learning.

- 总结:

Probabilistic spin to traditional autoencoders $\Rightarrow$ allows generating data.

Defines an intractable density $\Rightarrow$ derive and optimize a (variational) lower bound.

Pros:

Principled approach to generative models.

Allows inference of $q_\phi(z|x)$, can be useful feature representation.

Cons:

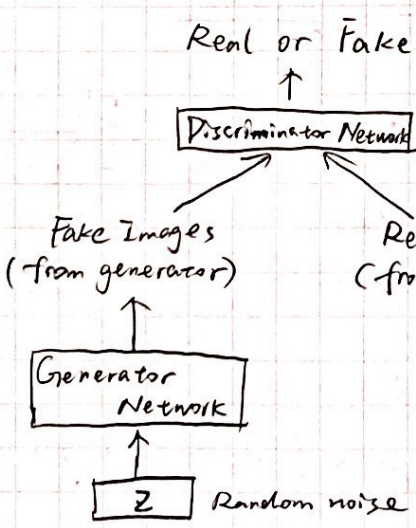
Maximize lower bound of likelihood, not as good evaluation as PixelRNN/CNN

Samples blurrier and lower quality compared to state-of-the-art (GANs)

Generative Adversarial Networks (GAN)

unsupervised learning with a supervised component

GANs don't work with any explicit density function. (我们只想要生成的样本)



Generator network: try to fool the discriminator by generating real-looking images.

Discriminator network: try to distinguish between real and fake images

Outputs likelihood in (0, 1) of real image

Minimax objective function.

$$\min_{\theta_g} \max_{\theta_d} \left[E_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data } x} + E_{z \sim p(z)} \log \left(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated data } G(z)}\right)\right]$$

Alternate between:

最大化 discriminator 做出正确决定的概率

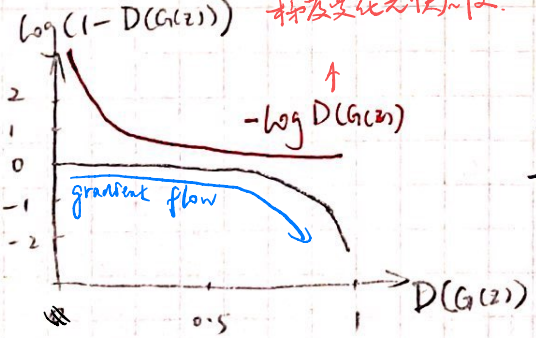
1. Gradient ascent on discriminator

$$\max_{\theta_d} \left[E_{x \sim p_{data}} \log D_{\theta_d}(x) + E_{z \sim p(z)} \log (1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

最小化 discriminator 做出正确决定的概率

2. Gradient descent on generator.

$$\min_{\theta_g} E_{z \sim p(z)} \log (1 - D_{\theta_d}(G_{\theta_g}(z)))$$



$\log(1-D(G(z)))$ 的图像表明当生成的 sample 不够好(fake)时，梯度小，训练慢。相反当生成 sample 够逼真时的梯度大，容易振荡，甚至发散。

Jointly training two networks is challenging, can be unstable. Choosing objectives with better loss landscapes helps training.

优化 → Alternate between:

1. Gradient ascent on discriminator

$$\max_{\theta_d} \left[E_{x\sim P_{data}} \log D_{\theta_d}(x) + E_{z\sim p(z)} \log\left(1 - D_{\theta_d}(G_{\theta_g}(z))\right) \right]$$

2. Gradient ascent on generator.

$$\max_{\theta_g} E_{z\sim p(z)} \log\left(D_{\theta_d}(G_{\theta_g}(z))\right)$$

→ 更新 generator，最小化 discriminator 出错的概率

- GAN training algorithm.

for number of training iterations do

  for k steps do.
    - Sample minibatch of m noise samples $\{z^{(1)} \dots z^{(m)}\}$ from noise prior $P_g(z)$
    - Sample minibatch of m examples $\{x^{(1)} \dots x^{(m)}\}$ from data generating distribution $P_{data}(x)$
    - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log\left(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))\right) \right]$$

  end for

  - Sample minibatch of m noise samples $\{z^{(1)} \dots z^{(m)}\}$ from $P_g(z)$
  - Update the generator by ascending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D_{\theta_d}(G_{\theta_g}(z^{(i)}))\right)$$

end for.

Wasserstein GAN balance the training times between 2 networks.

After train, use generator network to generate new images.

— Deep Convolutional GANs.

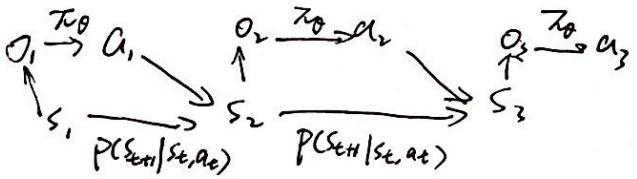
Generator is an upcampling network with fractionally-strided convolutions.
Discriminator is a convolutional network.

Architecture guidelines for stable Deep Conv GANs =

- Replace any pooling layers with stride conv in discriminator and fractional-strided convolutions in generator.
- Use batchnorm in both discriminator and generator
- Remove FC for deeper architectures
- Use ReLU in generator for all layers except for output, which uses tanh.
- Use LeakyReLU in discriminator for all layers.

Lecture 13 Visualizing and Understanding

Lecture 14 Deep Reinforcement Learning



Markov

sequential decision making

→ Definition:

A state s_t is Markov if and only if:

$$p(s_{t+1}|s_t) = p(s_{t+1}|s_1,\dots,s_t)$$

"The future is independent of the past given the present."

RL Algorithms
- Model-free RL
  - Policy optimization
    - DFO / Evolution.
    - Policy Gradients.
  - Q-Learning → (a value iteration method, DP)
  - policy Iteration
- Model-based RL
  - Learn the model
  - Given the model.

Markov Decision Process (MDP)

States: S Actions: A Transition function: $P(s'|s,a)$ (How long it will take.)

Reward function: $R(s,a,s')$ Discount factor: γ (horizon: H)

in state s take action a and land in s', how much reward get for that.

Goal: $\max_\pi E\left[\sum_{t=0}^{H} \gamma^t R(s_t,A_t,s_{t+1})|\pi\right]$

Deterministic policy: $a=\pi(s)$

Stochastic policy: $\pi(a|s)=p(A_t=a|s_t=s)$

↓

map from state to distributions of actions.

— Value Iteration

optimal Value Function:

(optimal) State-value function.

$$V^*(s) = \max_\pi E\left[\sum_{t=0}^{H} \gamma^t R(s_t,a_t,s_{t+1})|\pi, s_0=s\right]$$

↓

sum of discounted rewards when starting from state s if use the best possible policy

$$V_k^*(s) = \max_a \sum_{s'} p(s'|s,a)\left(R(s,a,s')+\gamma V_{k-1}^*(s')\right)$$

↓

optimal value for state s when $H=k$

$$V_H^* \xrightarrow{H\to\infty} V_\infty^* = V^*$$

Value Iteration Algorithm:

Start with $V_0^*(s)=0$ for all s

For $k=1\dots H$:

For all states s in S:

$$V_k^*(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left(R(s,a,s')+\gamma V_{k-1}^*(s')\right)$$

$$\pi_k^*(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s,a)\left(R(s,a,s')+\gamma V_{k-1}^*(s')\right)$$

action-value function Q-Values.

$Q^*(s,a)$ → expected utility starting in s, taking action a, and there-after acting optimally
(a don't have to be optimal action) ↓ 所以要 $\max_{a'}$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \underbrace{\gamma \max_{a'} Q^*(s',a')}_{V^*(s')})$$

$$= E[R_{t+1} + \gamma V^*(s') \mid S_t = s, A_t = a]$$

$$= E[R_{t+1} + \gamma \max_{a'} Q^*(s',a') \mid S_t = s, A_t = a]$$

$V^*(s') = \max_{a'} Q^*(s',a')$

Q-Value Iteration.

$$Q^*_{k+1}(s,a) \leftarrow \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max_{a'} Q^*_k(s',a'))$$

Q-value的好处在于可以评估当前状态下不同的a，~~以及取不同a后的不同s'下的~~

— Policy Iteration.

Policy Evaluation.

$$V^\pi_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V^\pi_k(s'))$$

Policy Iteration:

① policy evaluation for current policy π_k:

iterate until convergence $V^{\pi_k}_{i+1}(s) \leftarrow \sum_{s'} P(s'|s,\pi_k(s))[R(s,\pi_k(s),s') + \gamma V^{\pi_k}_i(s')]$

② policy improvement. (find the best action according to one-step lookahead)

$$\pi_{k+1}(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_k}(s')]$$

和 value iteration 区别在于 policy iteration中的 Policy evaluation 不会含有 max.
Value iteration 其实是 policy iteration 的改进版. (两步的 combination). 因为 Policy iteration 的每一步都有 policy evaluation, 这是一个"protracted iterative computation requiring multiple sweeps through the state set". 而 Value iteration 的改进在于每次迭代中只执行一次 sweep.

$$V^*_{i+1}(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*_i(s')]$$

sweep: one update for each state.

— Limitations.

(i) iteration over/storage for all states and actions: requires small, discrete state-action space → Q/V function fitting

(2) Update equations requires access to dynamics model ($P(s'|s,a)$) — unknown for model-free method.

↘ sampling-based approximations

— Sampling-based approximation:

$$Q_{k+1} \leftarrow \mathbb{E}_{s' \sim P(s'|s,a)} \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

⇑

Q-value iteration: $Q_{k+1}(s,a) \leftarrow \sum_{s'} P(s'|s,a) \left(R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right)$

Tabular Q-Learning: replace expectation by samples. (off-policy learning)

→ 隐式地找到最优 policy 通过 argmax Q function.

on-policy ↓ start with initial policy and collect samples according to it strictly.

Algorithm:

Start with $Q_0(s,a)$ for all s, a.

Get initial state s.

For $k = 1, 2 \ldots$ till convergence

  sample action a, get next state s'

  If s' is terminal:

    target = $R(s,a,s')$ ← immediate reward.

    Sample new initial state s'

  else:

    target = $R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$

  $Q_{k+1}(s,a) \leftarrow (1-\alpha) Q_k(s,a) + \alpha \cdot \text{target}$.

  $s \leftarrow s'$

How to sample actions?

1. choose random actions? → cover all states and actions.
2. Choose action that maximizes $Q_k(s,a)$ (greedily). → Choose actions greedily according to current estimate, ~~so that~~ which is almost perfect, so that chose good actions most of the time. ↓ may lead to insufficient exploration ~~if it initializes~~ during initial phase of learning.
3. ✓ trade-off 1 & 2. (exploring ⟷ exploiting)

ε-greedy: random action with prob ε, otherwise choose action greedily

Caveats:

- have to explore enough
- have to eventually make the learning rate α very small

value Iteration 不适合 sampling-Based approximation，因为不清楚如何在 max 里 draw samples.

而 Policy Evaluation 可以做 sampling-based approx ⟶ TD Learning (Temporal difference)

Policy Improvement 也不适合

当针对 s 很多时 (如围棋这种的游戏) ⋏ tabular method is not scalable.

甚至连续的环境 (离散化后的状态还是很多)

↓

- Approximate Q-Learning

传统 Q-Learning 在训练时需要记录所有的 Q 值在一个 table 里. Instead of a table, we have a parametrized Q function: $Q_\theta(s,a) \longrightarrow$ { linear function / complicated neural net. } ↗ 神经网络估计 Q 值, 并且在 states 之间 generalize.

> Learning rule:
>
> ① Remember: $\text{target}(s') = R(s,a,s') + \gamma \max_{a'} Q_{\theta_k}(s', a')$
>
> ② update:
> $$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \left[\frac{1}{2} \left(Q_\theta(s,a) - \text{target}(s') \right)^2 \right] \Big|_{\theta = \theta_k}$$

Tabular Q-Learning 为 Approximate Q-Learning 的特殊情况 当 $Q_\theta(s,a) = \theta_{sa}$.

$\overset{|A|}{\begin{bmatrix} \ddots \end{bmatrix}}|S|$ 假设 $\theta \in R^{|S| \times |A|}$, $Q_\theta(s,a) = \theta_{sa}$

$$\nabla_{\theta_{sa}} \left[\frac{1}{2} (Q_\theta(s,a) - \text{target}(s'))^2 \right] = \nabla_{\theta_{sa}} \left[\frac{1}{2} (\theta_{sa} - \text{target}(s'))^2 \right]$$
$$= \theta_{sa} - \text{target}(s')$$

$$\theta_{sa} \leftarrow \theta_{sa} - \alpha(\theta_{sa} - \text{target}(s'))$$
$$= (1-\alpha)\theta_{sa} + \alpha\, \text{target}(s')$$

Deep Q-Networks.

deep neural net 下的 approximate Q-learning 存在两个问题:

① 因为 network 在 states 之间具有泛化作用, 意味着若针对当前状态更新 θ, 同时会影响到别的状态, 使得 target 变得 non-stationary

② 随机梯度下降法 (应用于 NN) 有一个重要的假设是 data 是 iid 的. 而 RL 的在线学习的方式使得 target 之间有 correlations.

DQN.

Two main ideas for stabilizing Q-Learning: Experience replay

① Apply Q-updates on batches of past experience instead of online
- breaks local correlations from online updates
- makes the data distribution more stationary

② Use an older set of weights to compute the targets (target network)
- keeps the target function from changing too quickly

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i) \right)^2$$

alleviate correlations between Q-value and target.

Deep Q-Learning with experience replay Algorithm:

Initialize replay memory D to capacity N.
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$.

For episode = 1, ..., M do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1, ..., T do
    with probability ε select a random action a_t.
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$ } → ε-greedy
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D.
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D.
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ w.r.t the network param θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

---

other details: ① uses Huber loss instead of squared loss on Bellman error.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & |a| \le \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

② use RMSProp

③ ε Start at 1 → million frame → 0.1 or 0.05

— Policy Gradients.

直接寻找 policy $\pi_\theta(u|s)$ ⟶ 1. policy parameterized by parameter vector θ

$$\max_\theta E\left[\sum_{t=0}^{H} R(s_t)|\pi_\theta\right]$$

2. Stochastic policy class (smooth out the problem)

$\pi_\theta(u|s)$: probability of action u in state s.

连续 smooth out 是指.
如果有 deterministic policy (如 grid world) 改变 action 会跳
大幅度地改变输出. 当有 distribution of actions
时 action 的改变可以以某种连续的方式在 rewards 上. 这时我们可以用 Gradient descent 寻找 θ.

Why policy optimization?

1. often π can be simpler than Q or V
2. ~~Value function~~: doesn't prescribe actions (would need dynamics model.)
3. Q: need to be able to efficiently solve $\arg\max_u Q_\theta(s,u)$
   (challenge for continuous / high-dimensional action spaces) ⟶ NAF, Input ConvexNNs, Deep Energy Q.

	Policy Optimization	Dynamic Programming
Conceptually	Optimize what you care about	Indirect, exploit the problem structure self-consistency
~~Empirically~~ Empirically	More compatible with rich architectures (e.g. recurrence) More versatile More compatible with auxiliary objectives	More compatible with exploration and off-policy learning More sample-efficient when they work

Likelihood Ratio Policy Gradient.

τ denotes a state-action sequence $s_0, u_0 \dots s_H, u_H$.

$$R(\tau) = \sum_{t=0}^{H} R(s_t, u_t)$$

$$U(\theta) = E\left[\sum_{t=0}^{H} R(s_t, u_t); \pi_\theta\right] = \sum_\tau P(\tau;\theta) R(\tau)$$

The goal is ~~to~~ find θ:

$$\max_\theta U(\theta) = \max_\theta \sum_\tau P(\tau;\theta) R(\tau).$$

$$U(\theta) = \sum_\tau P(\tau;\theta) R(\tau)$$

$$\nabla_\theta U(\theta) = \nabla_\theta \sum_\tau P(\tau;\theta) R(\tau)$$

$$= \sum_\tau \nabla_\theta P(\tau;\theta) R(\tau) \qquad \log'(x) = \frac{1}{x}$$

$$= \sum_\tau \frac{P(\tau;\theta)}{P(\tau;\theta)} \nabla_\theta P(\tau;\theta) R(\tau)$$

$$= \sum_\tau P(\tau;\theta) \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} R(\tau)$$

$$= \sum_\tau P(\tau;\theta) \nabla_\theta \log P(\tau;\theta) R(\tau)$$

Approximate with the ~~empiri~~ empirical estimate for m sample paths under policy π_θ:

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)};\theta) R(\tau^{(i)})$$

unbiased estimate of the ∇: $E[\hat{g}] = \nabla_\theta U(\theta)$

↓

Valid even when: R is discontinuous and/or unknown

Sample space (of paths) is a discrete set.

The gradient tries to increase probability of paths with positive R and decrease the prob of paths with negative R.

! Likelihood ratio changes probabilities of experienced paths, does not try to change the paths (↔ path Derivative)

→ Decompose:

$$\nabla_\theta \log P(\tau^{(i)};\theta) = \nabla_\theta \log \left[\prod_{t=0}^{H} \underbrace{P(s_{t+1}^{(i)} \mid s_t^{(i)}, u_t^{(i)})}_{\text{dynamics model}} \cdot \underbrace{\pi_\theta(u_t^{(i)} \mid s_t^{(i)})}_{\text{policy}} \right]$$

$$= \nabla_\theta \left[\sum_{t=0}^{H} \log P(s_{t+1}^{(i)} \mid s_t^{(i)}, u_t^{(i)}) + \sum_{t=0}^{H} \log \pi_\theta(u_t^{(i)} \mid s_t^{(i)}) \right]$$

$$= \nabla_\theta \sum_{t=0}^{H} \log \pi_\theta(u_t^{(i)} \mid s_\theta^{(i)})$$

$$= \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(u_t^{(i)} \mid s_t^{(i)})$$

this is the neural net, which encodes the distribution of actions given states.

no dynamics model required.

→ Derivation from Importance Sampling.

$$U(\theta) = E_{\tau \sim \theta_{old}}\left[\frac{P(\tau|\theta)}{P(\tau|\theta_{old})} R(\tau)\right]$$

↓

Importance sampling means sample from the old policy $P(\tau|\theta_{old})$ to derive new $P(\tau|\theta)$ current policy θ.

The ratio $\frac{P(\tau|\theta)}{P(\tau|\theta_{old})}$ gives the correction.

$$\nabla_\theta U(\theta) = E_{\tau \sim \theta_{old}}\left[\frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta_{old})} R(\tau)\right]$$

$$\nabla_\theta U(\theta)|_{\theta=\theta_{old}} = E_{\tau \sim \theta_{old}}\left[\frac{\nabla_\theta P(\tau|\theta)|_{\theta_{old}}}{P(\tau|\theta_{old})} R(\tau)\right]$$

$$= E_{\tau \sim \theta_{old}}\left[\nabla_\theta \log P(\tau|\theta)|_{\theta_{old}} R(\tau)\right] \rightarrow \text{the same as former derivation}$$

↓ suggests we can also look at more than just gradient. e.g. can use importance sampled objectives as "surrogate" loss. (change your policy after your gradient update, use this equation to estimate how good that update was.)

→ So far, unbiased but very noisy

Fixes that lead to real-world practicality:

1. Baseline
2. Temporal structure
3. KL-divergence trust region/natural gradient.

↓ general trick, equally applicable to perturbation analysis and finite differences.

1. Baseline.

increase gradient of paths that better than average and decrease worse than average.

$$\nabla U(\theta) \approx \hat{g} = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)};\theta)(R(\tau^{(i)}) - b)$$

Remove terms that don't depend on current action can lower variance:

$$\frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})\left(\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)}) - b(s_t^{(i)})\right)$$

→ choices of b

- constant baseline: $b = E[R(\tau)] \approx \frac{1}{m}\sum_{i=1}^{m} R(\tau^{(i)})$
- optimal constant baseline: $b = \frac{\sum_i (\nabla_\theta \log P(\tau^{(i)};\theta))^2 R(\tau^{(i)})}{\sum_i (\nabla_\theta \log P(\tau^{(i)};\theta))^2}$
- Time based: $b_t = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=t}^{H-1} R(s_k^{(i)}, u_t^{(i)})$
- State-dependent:

$$b(s_t) = E[r_t + r_{t+1} + \cdots + r_{H-1}] = V^\pi(s_t)$$

↓

Increase the logprob of action proportionally to how much its returns are better than the expected return under the current policy

estimation of V^π:

Monte-Carlo.

Init $V^\pi_{\phi_0}$

Collect trajectories $\tau_1 \dots \tau_m$

Regress against empirical return: $\phi_{i+1} \leftarrow \arg\min_\phi \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H-1} \left(V^\pi_\phi(s_t^{(i)}) - \left(\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)}) \right) \right)^2$

TD estimate of V^π:

Bellman Equation. $V^\pi(s) = \sum_u \pi(u|s) \sum_{s'} P(s'|s,u)[R(s,u,s') + \gamma V^\pi(s')]$

Init $V^\pi_{\phi_0}$,

Collect data $\{s, u, s', r\}$

Fitted V iteration: $\phi_{i+1} \leftarrow \min_\phi \sum_{(s,u,s',r)} \| r + V^\pi_{\phi_i}(s') - V_\phi(s) \|_2^2 + \lambda \| \phi - \phi_i \|_2^2$

→ Recall the likelihood Ratio PG Estimator

$$\frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(u_t^{(i)} | s_t^{(i)}) \cdot \left(\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)}) - V^\pi(s_k^{(i)}) \right)$$

Estimation of Q from single roll-out

$Q^\pi(s,u) = E[r_0 + r_1 + r_2 + \dots | s_0 = s, u_0 = u]$

↳ high variance per sample based / no generalization

↳ reduce variance by discounting $E[r_0 + \gamma r_1 + \gamma^2 r_2 \dots | s_0 = s, u_0 = u]$

↳ by function approximation → critic

Reducing Variance by Function Approximation.

$Q^{\pi,T}(s,u) = E[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots | s_0 = s, u_0 = u]$ $(1-\lambda)$

$= E[r_0 + \gamma V^\pi(s_1) | s_0 = s, u_0 = u]$ $(1-\lambda)\lambda$

$= E[r_0 + \gamma r_1 + \gamma^2 V^\pi(s_2) | s_0 = s, u_0 = u]$ $(1-\lambda)\lambda^2$

$= \dots$ $(1-\lambda)\lambda^3$

Async Advantage Actor Critic (A3C): $\hat{Q}$ = one of the above choices (e.g. k=5 step look ahead)

Generalized Advantage Estimation (GAE): $\hat{Q}$ = lambda exponentially weighted average of all the above.

Actor - Critic with A3C or GAE.

init π_{θ_0} $V_{\phi_0}^{\pi}$

collect roll-outs $\{s, u, s', r\}$ and $\hat{Q}_i(s, u)$

update: $\phi_{i+1} \leftarrow \min_{\phi} \sum_{\{s,u,s',r\}} \| \hat{Q}_i(s,u) - V_{\phi}^{\pi}(s) \|_2^2 + \varepsilon \| \phi - \phi_i \|_2^2$ ~~A~~ TD.

$$\theta_{i+1} \leftarrow \theta_i + \alpha \frac{1}{m} \sum_{k=1}^{m} \sum_{t=0}^{H-1} \nabla_{\theta} \log \pi_{\theta_i}(u_t^{(k)} | s_t^{(k)}) \left(\hat{Q}_i(s_t^{(k)}, u_t^{(k)}) - V_{\phi_i}^{\pi}(s_t^{(k)}) \right)$$

↓

Initialize policy param θ. critic param ϕ.

For iteration = 1, 2 ... do

Sample m trajectories under the current policy

$\Delta\theta \leftarrow 0$

For $i = 1 \dots m$ do

For $t = 1 \dots T$ do

$$A_t = \sum_{t' \geq t} \gamma^{t'-t} \cdot r_t^{(i)} - V_{\phi}(s_t^{(i)})$$

$$\Delta\theta \leftarrow \Delta\theta + A_t \cdot \nabla_{\theta} \log \pi_{\theta}$$

$$\Delta\phi \leftarrow \sum_i \sum_t \nabla_{\phi} \| A_t^{(i)} \|^2$$ ← Monte - Carlo

$\theta \leftarrow \alpha \Delta\theta$

$\phi \leftarrow \beta \Delta\phi$

End for

Lecture 16 Adversarial Examples and Adversarial Training

— What are adversarial examples?

1) 选择任一图片
2) 选择任一类别（错误类别）
3) 修改图片来最大化该类别
4) Repeat until network is fooled.

最终生成的 adversarial example 对于 network 来说是不同类别，但人眼分别不出且 network 往往对错误的类别有很高的 confidence (e.g. 0.999)

Not just for neural nets. → Linear models (Logistic regression, softmax regression, SVMs)
→ Decision Trees
→ Nearest neighbors.

— Why do they happen?

最开始怀疑是因为 overfitting，但是如果因为 overfitting，则生成的 mismatching 会有一定的随机性。而实际发生的现象是不同的模型会 misclassify 相同的 adversarial examples，并会 assign 相同的 class 给它们。另一个现象是将 input 和 adversarial example 之间的 offset vector 加到干净的图片上，这个图片也会变为 adversarial example. (almost always)
所以这是一个 systematic effect 而不是 random effect.

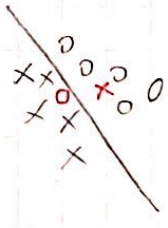
↓

真正的原因是 underfitting. → Modern deep nets are very piecewise Linear

激活函数如 ReLU, Maxout 都是 piecewise linear 的。
carefully tuned sigmoid 也在类线性区激活。
甚至 LSTM 使用加法来累计和记录信息 over time 也很线性。

这里的线性是指从输入到输出的映射是线性的。从参数到输出的映射是非线性的，因为 weight matrices 是相乘的。它也使得训练 neural network 变得很难。
不同层的

但对输入到输出的 mapping 更加线性，意味着 optimization problems that aim to optimize the input to the model are much more easier than optimization problems that aims to optimize the parameters.



The Fast Gradient Sign Method.

$$J(\tilde{x}, \theta) \approx J(x, \theta) + (\tilde{x} - x)^T \nabla_x J(x)$$ → Cost w.r.t. input x.

Maximize $J(x, \theta) + (\tilde{x} - x)^T \nabla_x J(x)$

Subject to $\|\tilde{x} - x\|_\infty \leq \varepsilon$ ⟶ 保证 pixels 的变化对于人眼不可分辨.

$$\Rightarrow \quad \tilde{x} = x + \varepsilon \cdot \text{sign}(\nabla_x J(x))$$

其它生成 adversarial examples 的方法，Nicholas Carlini's attack based on multiple steps of the Adam optimizer.

FGSM 对于无 defenses 的普通神经网络有超过 99% 的 attack success rate.

「FGSM 表明只需要找到与 gradient 有大的点积的方向，并在该方向上移动很小距离就可以欺骗模型.」

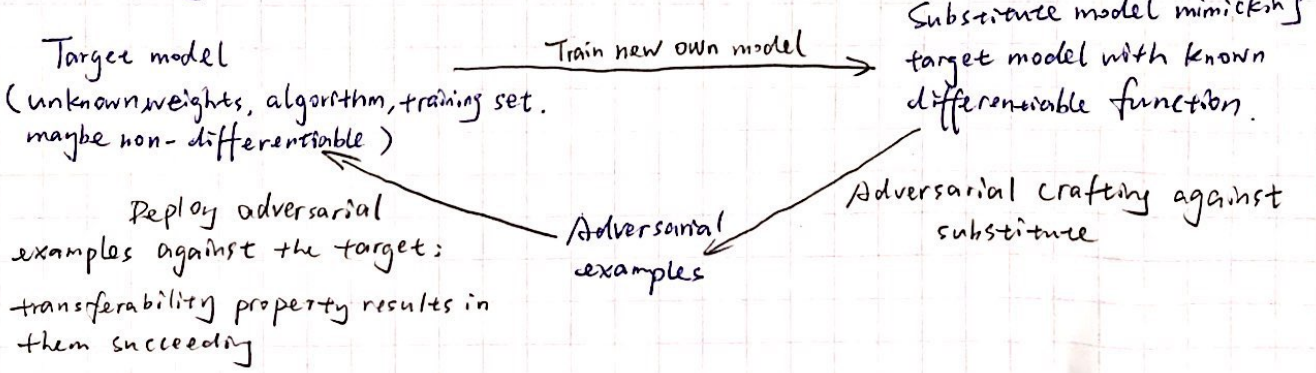
另外需要注意的是 ~~Adversa~~ Adversarial examples are not noise. 原因在于，在高维空间里，random vector 平均有 zero dot product with reference vector. (参考 FGSM 中 $\varepsilon \cdot \text{sign}(\nabla_x J(x))$，adversarial examples 选择最大化 cost).

Adversarial region (subspace) 指和 gradient 同时有很大内积的正交子空间.
MNIST 有平均 25 维的 adversarial dimensions. adversarial dimensions 的占比越大，意味着越容易生成 adversarial examples，且不同模型越容易 intersect 而生成相同 adversarial examples.
(random noise. / subspaces 占比大的不同模型.)

— How can they be used to compromise machine learning systems?

由于 adversarial examples 具有 transferability，意味着对于不同 dataset，甚至不同 machine learning technique (DNN, LR, SVM, DT, KNN)，Adversary 具有可转移性.
具有可转移性的原因在于，模型即使不同，dataset 即使不同，但最终学习到的 function 是类似的，其对于类似 adversarial examples 都脆弱.

Transferability Attack:



substitute model 可以使用多种模型的 ensemble 来加强 transferability.

—— What are the defenses?

Generative Modeling is not sufficient to solve the problem.

如果能够真实地知道 $P(x)$ 分布，则可以判断出哪些是 adversarial example. 但现实是 ~~即使知道 posterior $P(y=1|x)$ 也无法对齐相同(类似)的 $P(x)$~~.

计算 posterior $P(y=1|x)$ 可能更重要. 如果一个够好的生成模型可以得到真实的 image 以及准确的 posterior 分布，则这种试是可行的. 但准确的分布是很难得到的，通常得到的只是它的近似，使得 posterior distribution to extrapolate linearly again.

Universal approximator theorem: Neural nets can represent (approximate) continuous functions on compact subset of R^n.

现在我们可以把 NN 训练出具有 linear decision function，如果我们想有 step function？

↓

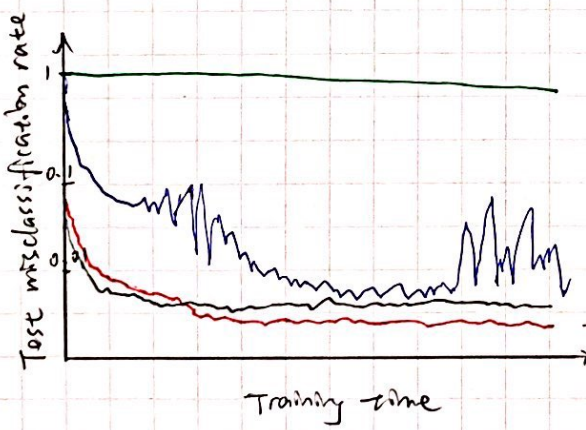
Training on Adversarial Examples.

↓

对于每个输入 x 都加上 adversarial example 来训练 (map to right label)

↓

这样可以大体上避免被相同的 adversary 攻击.



— Train = clean Test = clean
— Train = clean Test = Adv
— Train = Adv Test = clean
— Train = Adv Test = Adv

} Test on the same kind of adv examples that trained on.

红色比蓝色还要低，说明这里 training on adv 帮助原来的模型更好地 regularize.

↓

good regularizer

如果原模型 overfitting，training on adv 可以帮助其更少的 overfitting.

Adversarial trained neural nets have the best empirical success rate on adversarial examples of any machine learning models.

↓

Deep NN can be trained to be nonlinear, so it can be a path to a solution.

— Conclusion.

Attacking is easy.

Defending is difficult.

Adversarial training provides regularization and semi-supervised learning.

The out-of-domain input problem is a bottleneck for model-based optimization generally.