训练神经网络

主讲:邓伟洪
http://www.pris.net.cn/introduction/teacher/dengweihong
模式识别与智能系统实验室

人工智能学院
北京邮电大学

本节课内容

- 激活函数
- 批归一化
- 迁移学习
- 更新规则
- 学习率调整
- 数据增广

概述

1.一次性设置

激活函数, 预处理, 权重初始化, 正则化, 梯度检查

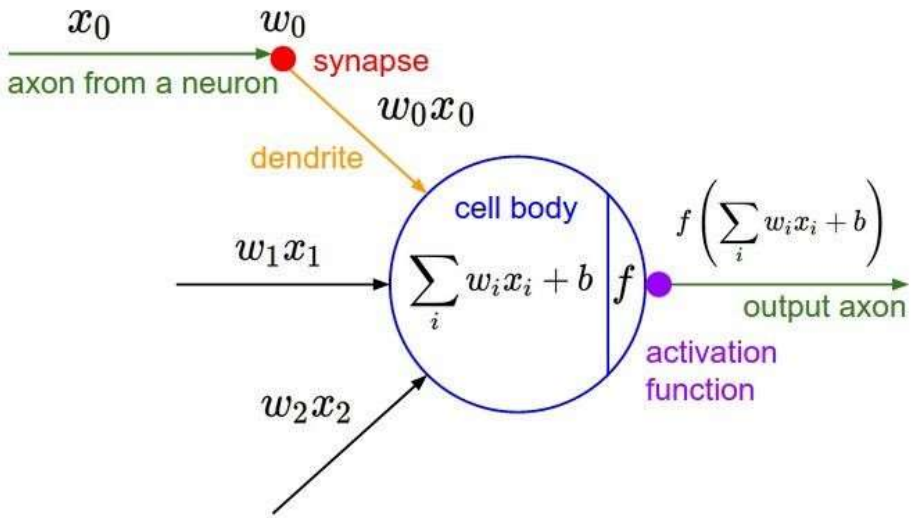
2.训练动态

迁移学习, 训练过程监控, 参数更新, 超参数优化

3.评价

模型集成, 测试时增强

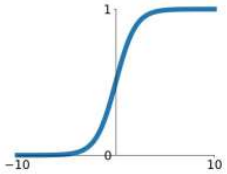
激活函数


x_0
w_0
axon from a neuron
synapse
$w_0 x_0$
dendrite
cell body
$w_1 x_1$
$\sum_i w_i x_i + b$
f
$f\left(\sum_i w_i x_i + b\right)$
output axon
activation function
$w_2 x_2$

激活函数

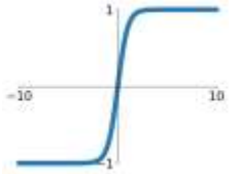
Sigmoid

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



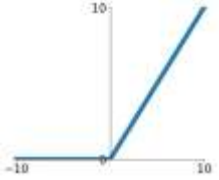
tanh

$$\tanh(x)$$



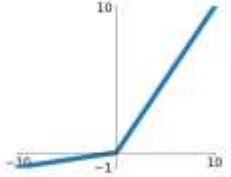
ReLU

$$\max(0, x)$$



Leaky ReLU

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

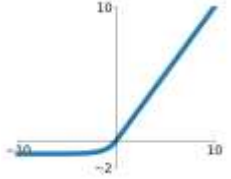


Maxout

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

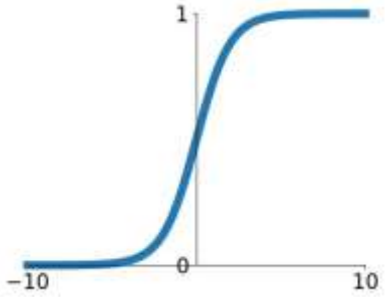
ELU

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



激活函数

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



Sigmoid

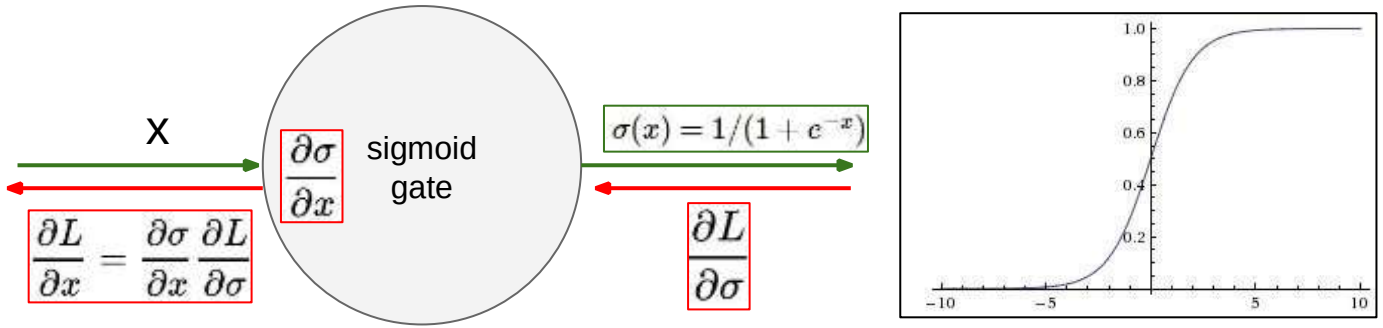
将数字压缩到范围[0,1]

由于它们对神经元的饱和“发射率”有很好的解释，因此在历史上很受欢迎

3个问题

1.饱和神经元“杀死”梯度

激活函数



x=-10的时候发生什么？

x=0的时候发生什么？

x=10的时候发生什么？

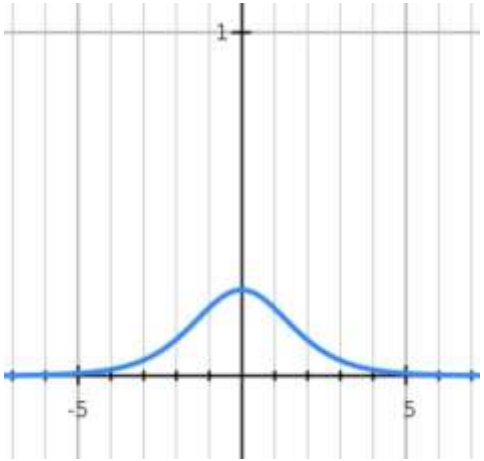
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1-\sigma(x))$$

激活函数

x=-10的时候发生什么？

x=0的时候发生什么？

x=10的时候发生什么？



$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x))$$

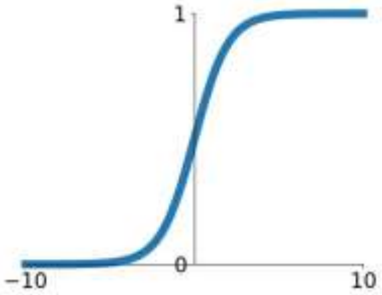
激活函数

3个问题

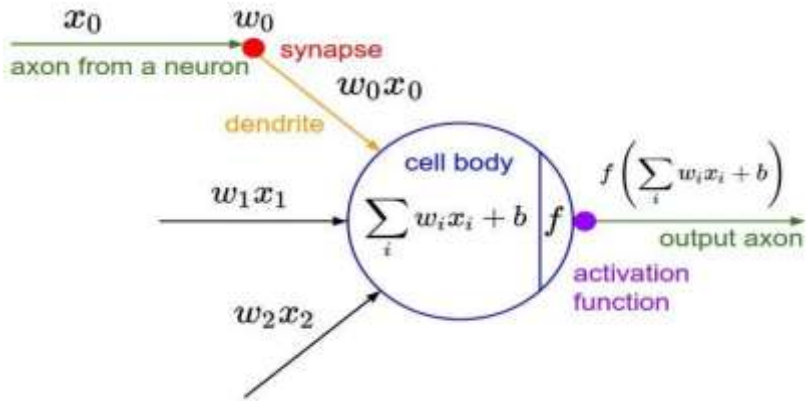
2.Sigmoid输出不以0为中心

思考一下如果神经元的输入一直是正的会发生什么？

w的梯度会变成什么样子?



Sigmoid



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

激活函数

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

Sigmoid函数的局部梯度总是正的

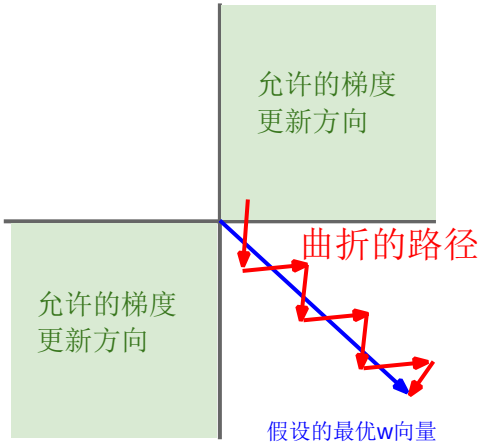
我们已经假设输入x总是正的

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

因此，所有w_i的梯度的符号和上游标量梯度的符号是相同的

激活函数

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

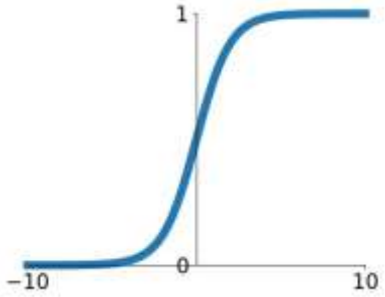


根据上面的分析，w的梯度一直是正的或者一直是负的

对于单个元素，采用minibatch可以缓解这种问题

激活函数

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



Sigmoid

将数字压缩到范围[0,1]

由于它们对神经元的饱和“发射率”有很好的解释，因此在历史上很受欢迎

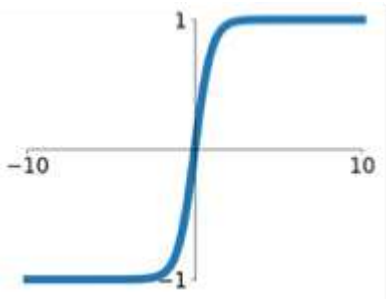
3个问题

1.饱和神经元“杀死”梯度

2.Sigmoid输出不以0为中心

3.指数运算有一点计算昂贵

激活函数



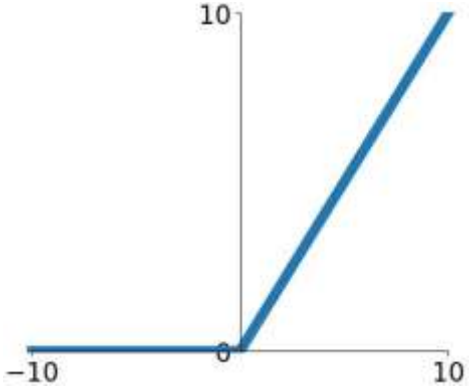
tanh(x)

将数字压缩到范围[-1,1]

以0为中心(很好的特性)

饱和时仍然“杀死”梯度

激活函数



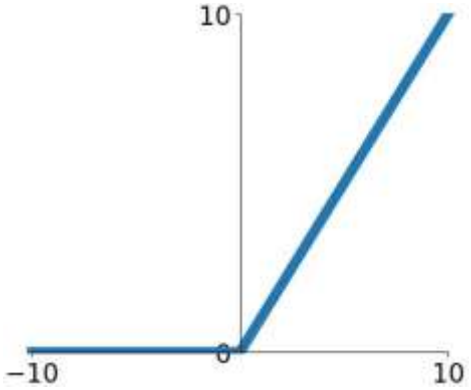
ReLU
(线性整流单元)

f(x) = max(0,x)

- 不会饱和(在正值部分)
- 计算效率很高
- 实际中比sigmoid/tanh收敛更快 (6倍)
- 输出不以0为中心

[Krizhevsky et al., 2012]

激活函数



ReLU
(线性整流单元)

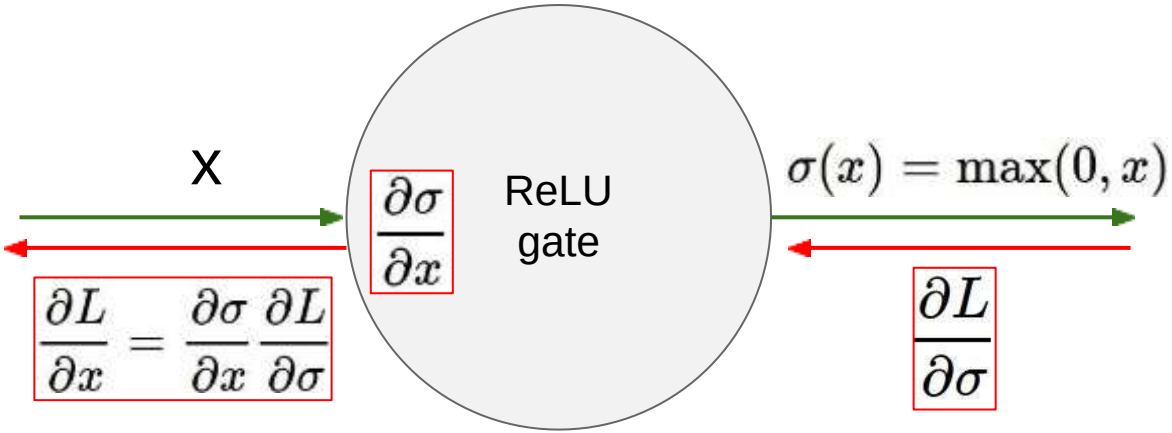
f(x) = max(0,x)

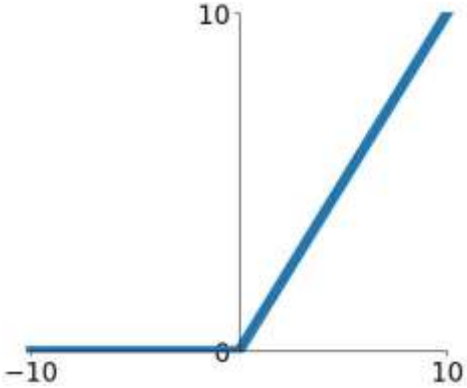
- 不会饱和(在正值部分)
- 计算效率很高
- 实际中比sigmoid/tanh收敛更快 (6倍)

- 输出不以0为中心
- 另一个缺点:

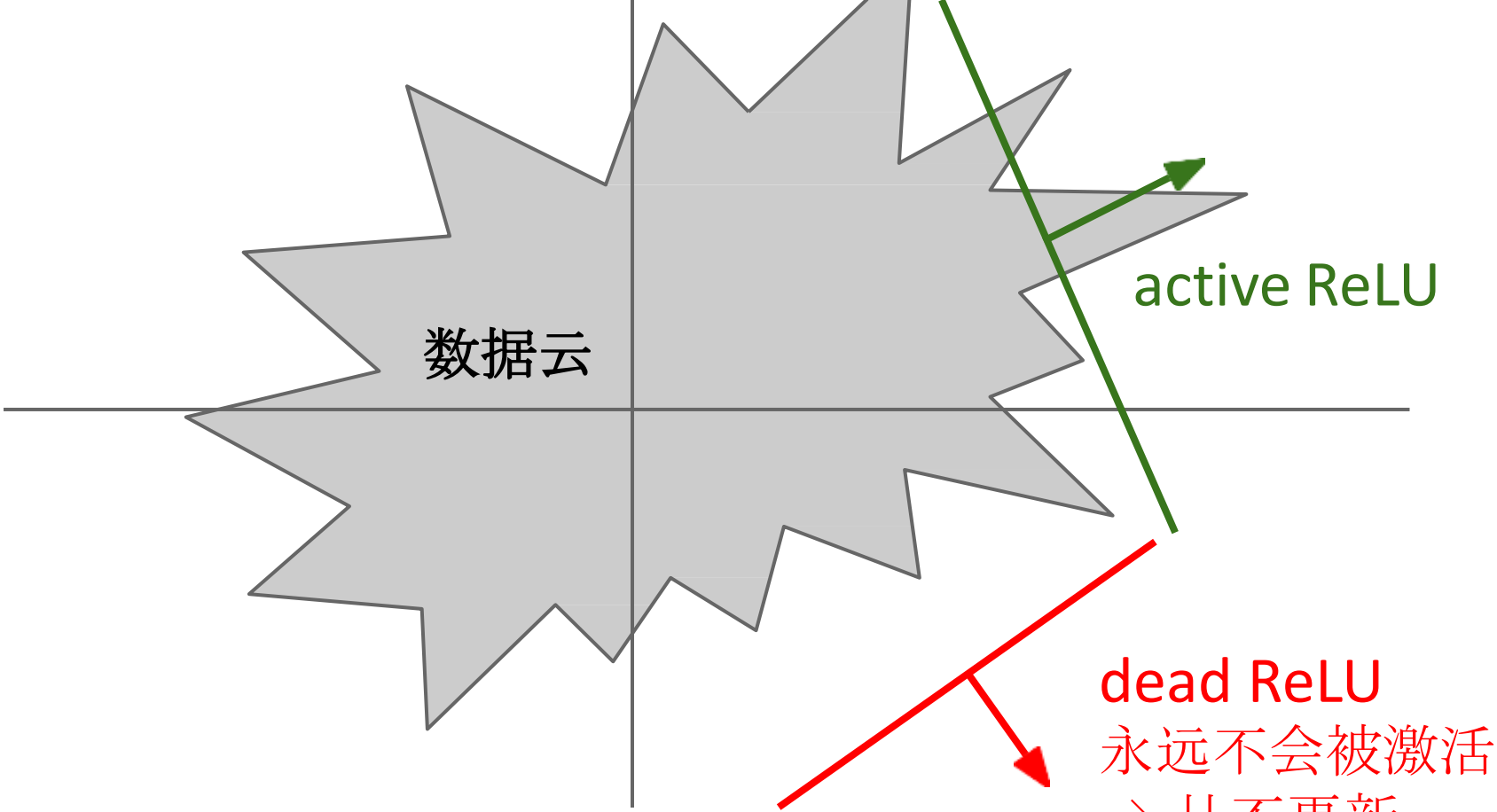
  提示: x < 0时的梯度是什么?

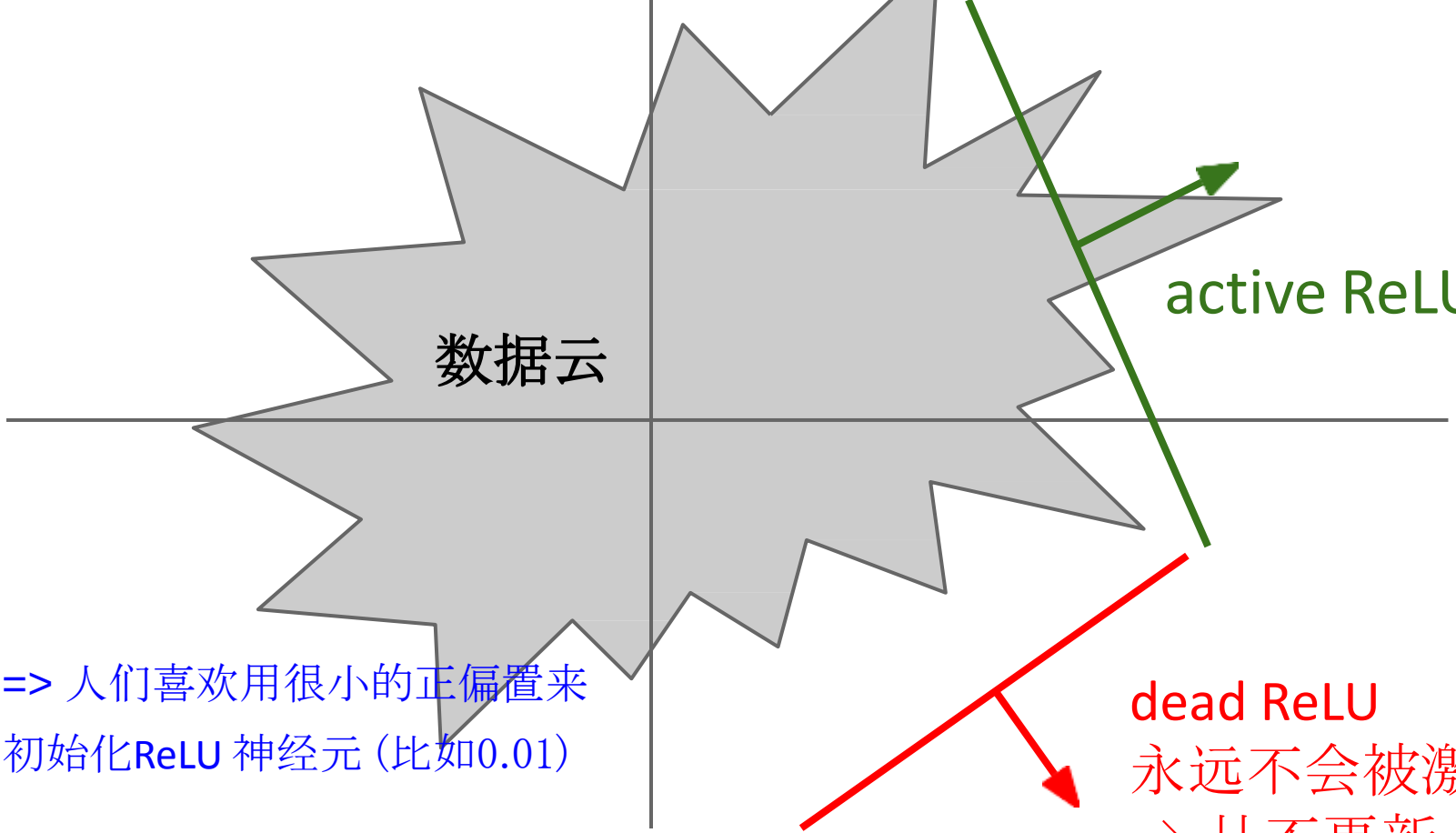
[Krizhevsky et al., 2012]





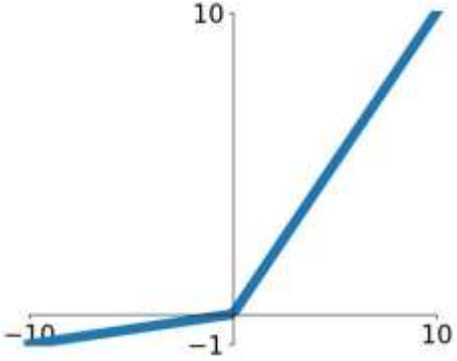
当x = −10的时候会发生什么？
当x = 0的时候会发生什么？
当x = 10的时候会发生什么？


数据云
active ReLU
dead ReLU
永远不会被激活
=> 从不更新



=> 人们喜欢用很小的正偏置来初始化ReLU 神经元 (比如0.01)

激活函数



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

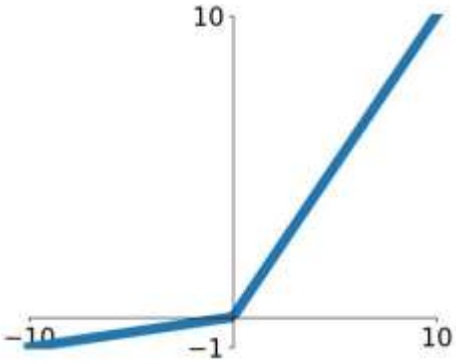
- 不会饱和
- 计算效率高
- 实际中比sigmoid/tanh 收敛更快! (6倍)
- 不会 “死亡”.

Leaky ReLU

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

激活函数

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



- 不会饱和
- 计算效率高
- 实际中比sigmoid/tanh 收敛更快! (6倍)
- 不会“死亡”.

Leaky ReLU

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

参数整流单元(**PReLU**)

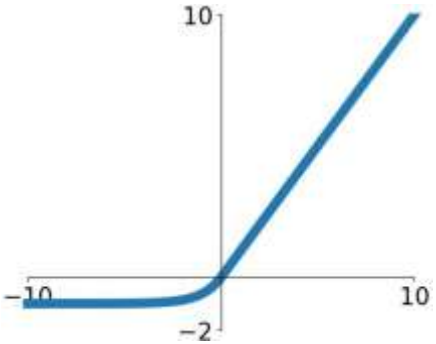
$$f(x) = \max(\alpha x, x)$$

反向传播进alpha(参数)

激活函数

[Clevert et al., 2015]

Exponential Linear Units (ELU)



（Alpha 默认值= 1）

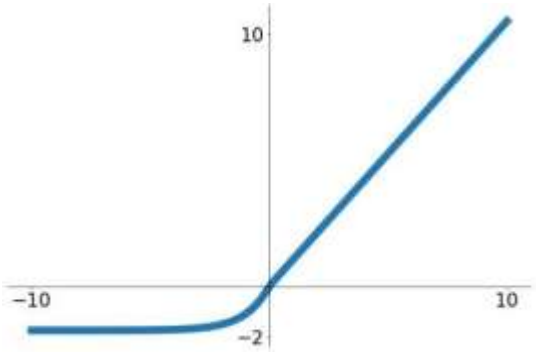
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$

- 拥有ReLU的所有优点
- 接近0均值输出
- 与Leaky ReLU相比，负饱和状态增加了对噪声的鲁棒性
- 需要计算指数

激活函数

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



- 经过放缩的ELU，更适合深度网络
- 自归一化特性
  - 不使用批归一化就可以训练深度SELU网络
  - （后续进行更多讨论）

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

α = 1.6733, λ = 1.0507

Maxout “Neuron”

[Goodfellow et al., 2013]

- 没有使用基础形式的点积-> 非线性Generalizes ReLU and Leaky ReLU
- 线性区域! 不会饱和! 不会死亡!

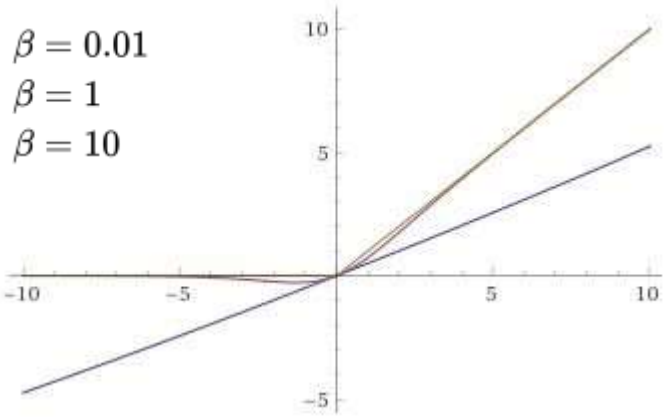
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

问题: 参数/神经元的数量翻倍

激活函数

[Ramachandran et al. 2018]

Swish



$$f(x) = x\sigma(\beta x)$$

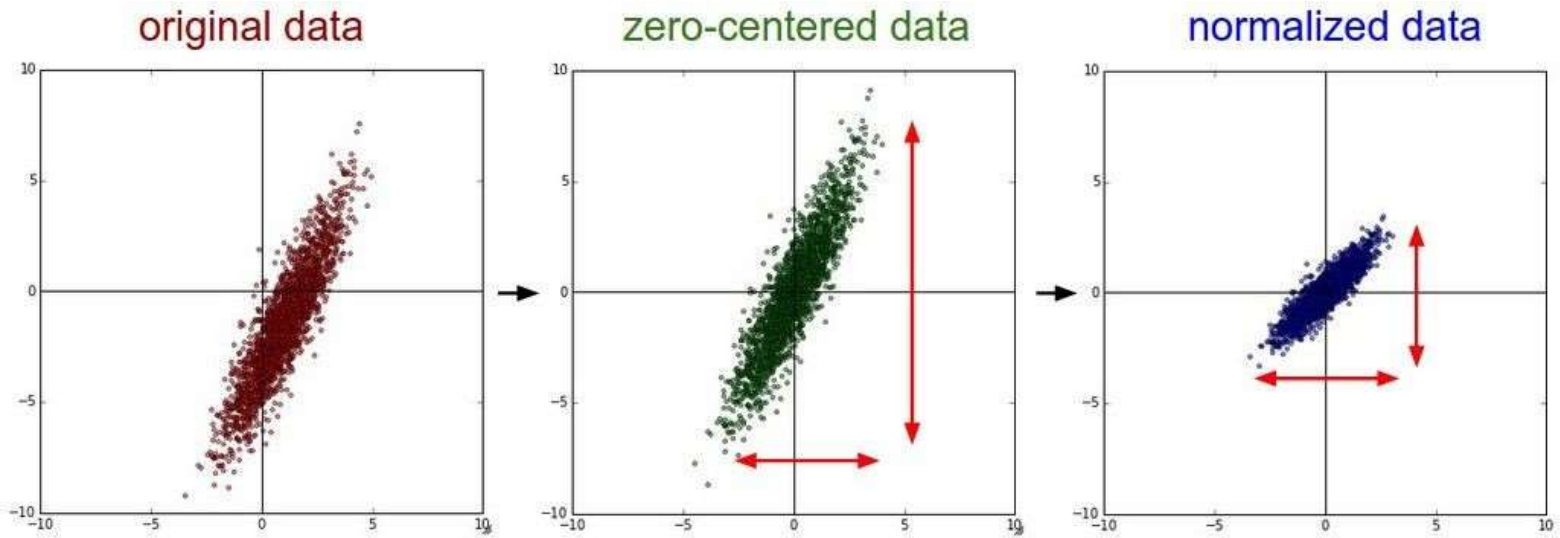
- 他们训练了一个神经网络来生成和测试不同的非线性。
- Swish在CIFAR-10的准确度上优于其他所有选项

简单总结，在实际中：

- 使用ReLU. 谨慎选择学习率
- 尝试使用Leaky ReLU / Maxout / ELU / SELU
- 挤出一些边际增益
- 不要使用sigmoid / tanh

数据预处理

数据预处理



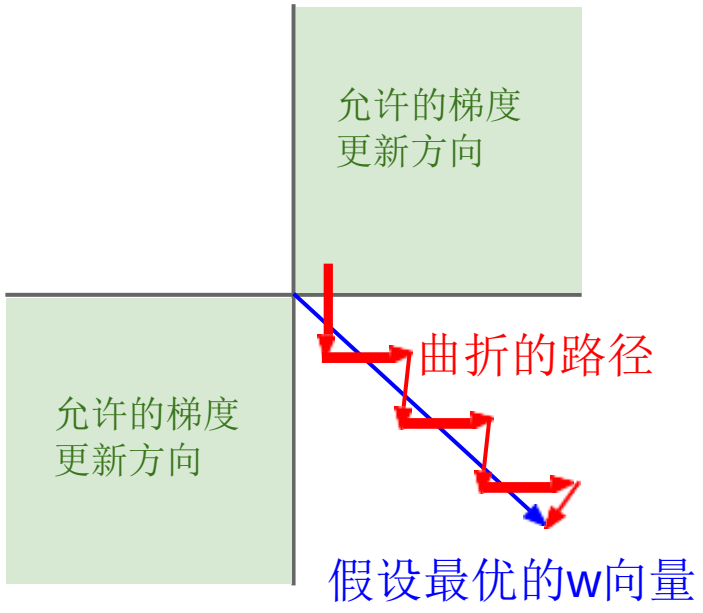
(假设X [NxD] 是一个数据矩阵)

```
X -= np.mean(X, axis = 0)
```

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

回顾:考虑一下如果神经元的输入
都是正的会发生什么？

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

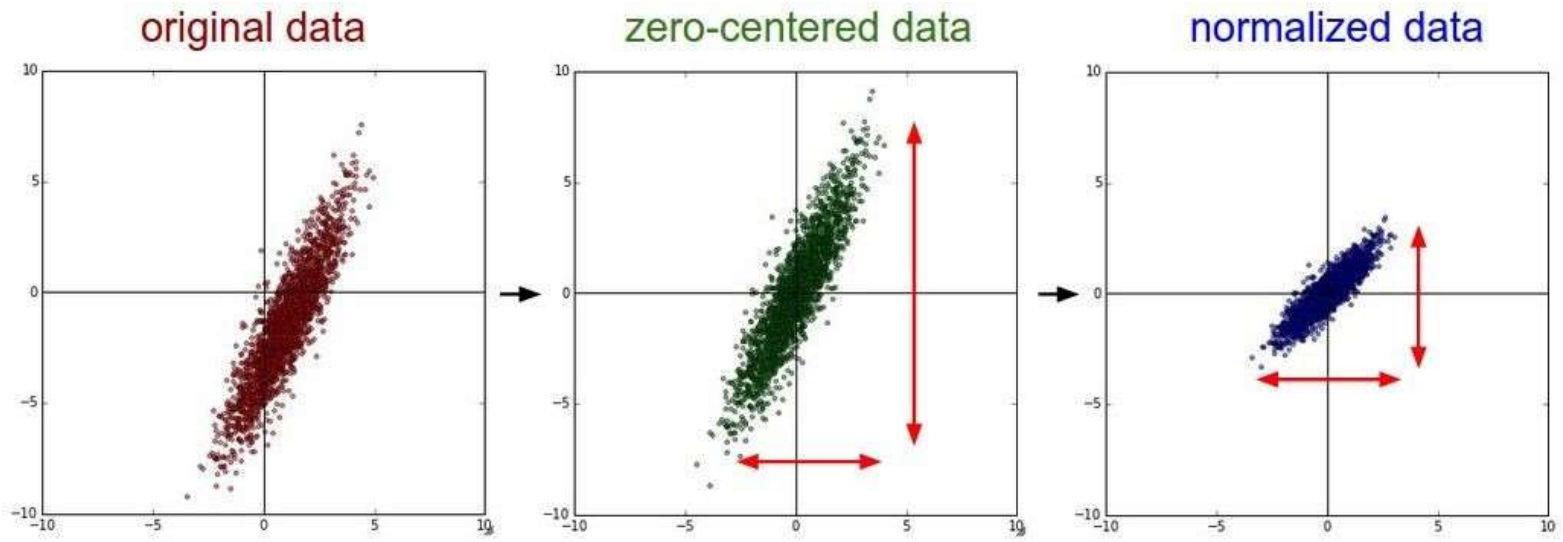


w的梯度会是什么样的呢?

总是正的或者总是负的:(

(这也是为什么我们想要以0为中心的数据!)

数据预处理



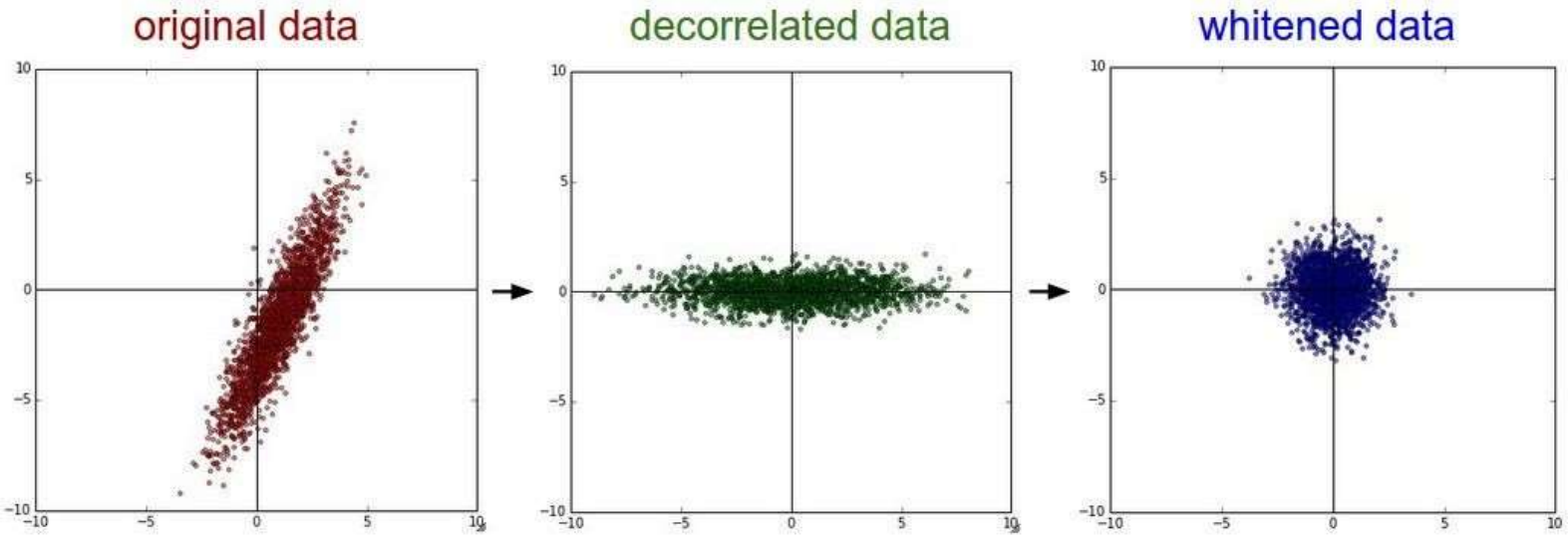
(假设X [NxD] 是一个数据矩阵)

```
X -= np.mean(X, axis = 0)
```

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

数据预处理

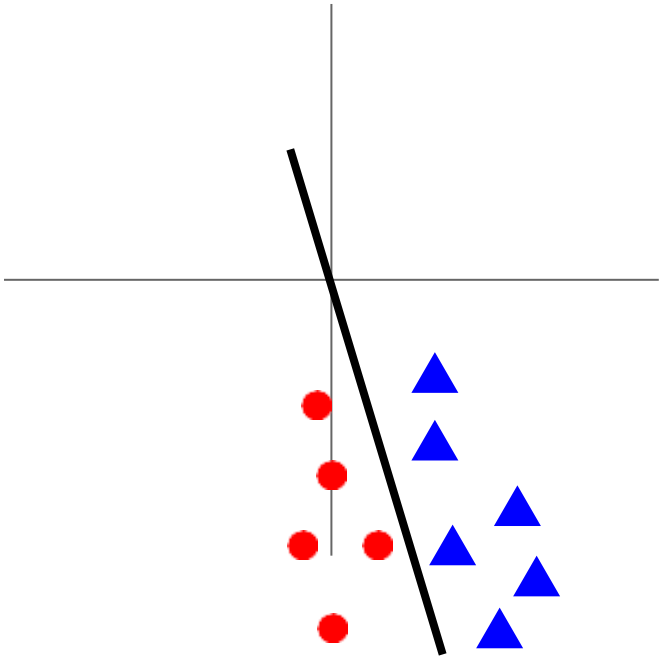
在实际中，你也许会遇到**主成分分析(PCA)和数据白化(Whitening)**



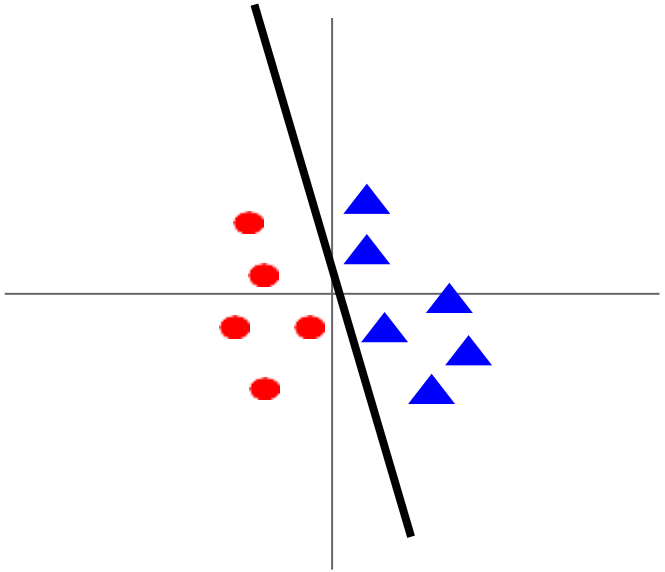
（数据具有对角协方差矩阵）　　（协方差矩阵是单位矩阵）

数据预处理

归一化之前: 分类损失对权重矩阵的变化非常敏感; 难以优化



归一化之后: 降低了分类损失对权重小变化的敏感程度; 更容易优化



实际中对于图像：只选取图像中心

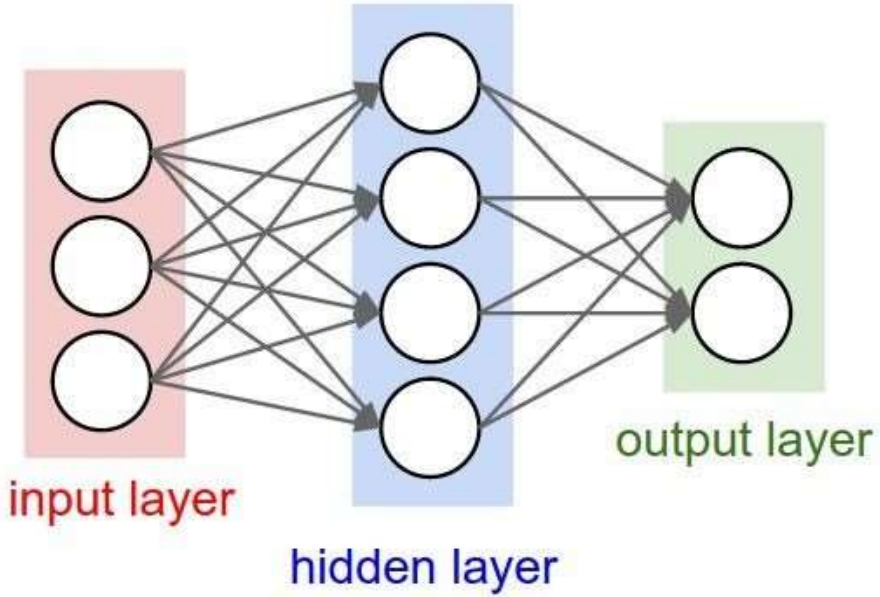
例子：考虑 CIFAR-10 中[32,32,3]的图像

- 减去平均图像(例子：AlexNet)（平均图像= [32,32,3] 数组）
- 减去每一个通道的平均值(例子：VGGNet)（沿着每个通道的平均值= 3 个固定值）
- 减去每一个通道的平均值并且除以每一个通道的方差(例子：ResNet) (每个通道的平均值= 3 个固定值)

不常用PCA 或者 whitening

权重初始化

- 问题: 用常数初始化W会发生什么?



- 最初的想法: 数值小的随机数字

(以0为均值, 以0.01为标准差的高斯分布)

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

应用于小网络效果还可以, 应用于更深的网络会出现问题

权重初始化: 激活数据

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

隐藏层大小为4096的6层网络的前向传播

最后一层的激活会发生什么?

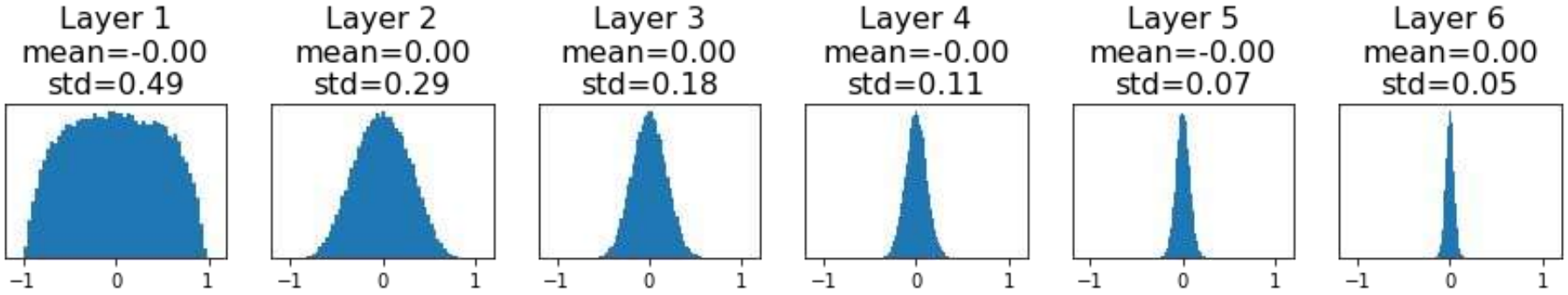
权重初始化:激活数据

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

隐藏层大小为4096的6层网络的前向传播

对于更深的网络层，所有激活都趋向于零

问题: 梯度 dL/dW是什么样的？



权重初始化:激活数据

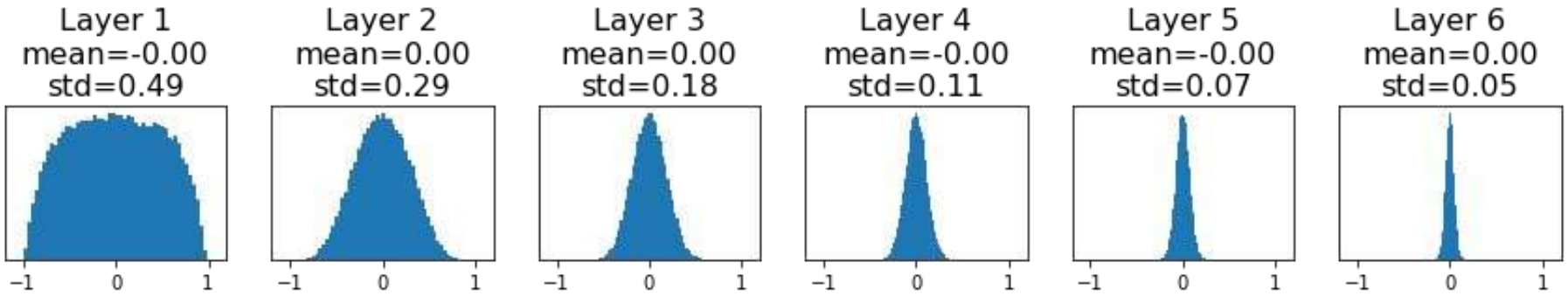
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

隐藏层大小为4096的6层网络的前向传播

对于更深的网络层，所有激活都趋向于零

问题: 梯度 dL/dW是什么样的？

答案: 梯度全是0，没有进行学习



权重初始化:激活数据

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

将初始权重的标准差
从 0.01 增加到0.05

最后一层的激活会发生什么?

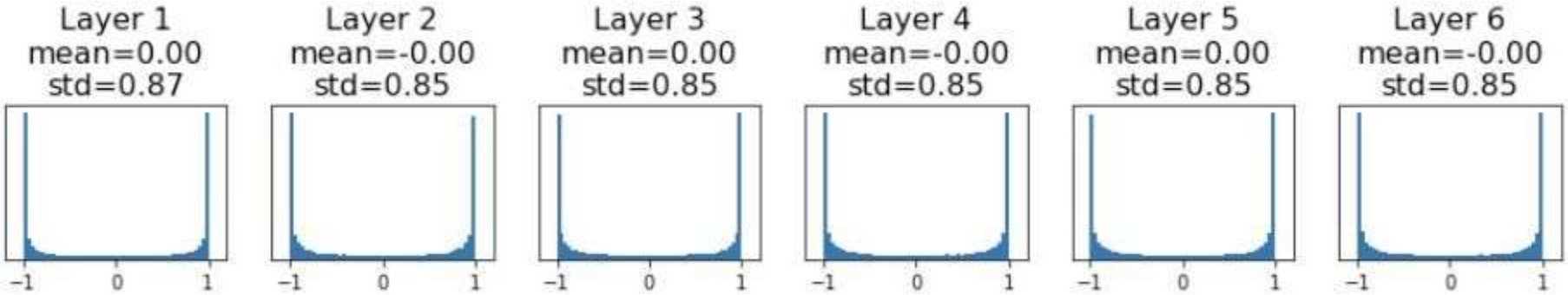
权重初始化:激活数据

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

将初始权重的标准差
从 0.01 增加到0.05

所有的激活都饱和了

问题:梯度是什么样子的？



权重初始化:激活数据

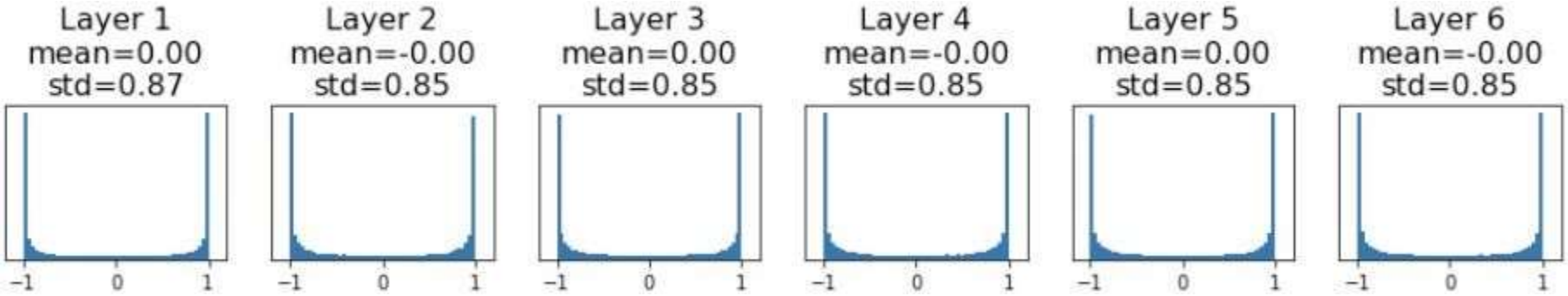
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

将初始权重的标准差
从 0.01 增加到0.05

所有的激活都饱和了

问题:梯度是什么样子的？

答案:局部梯度都是0，没有进行学习



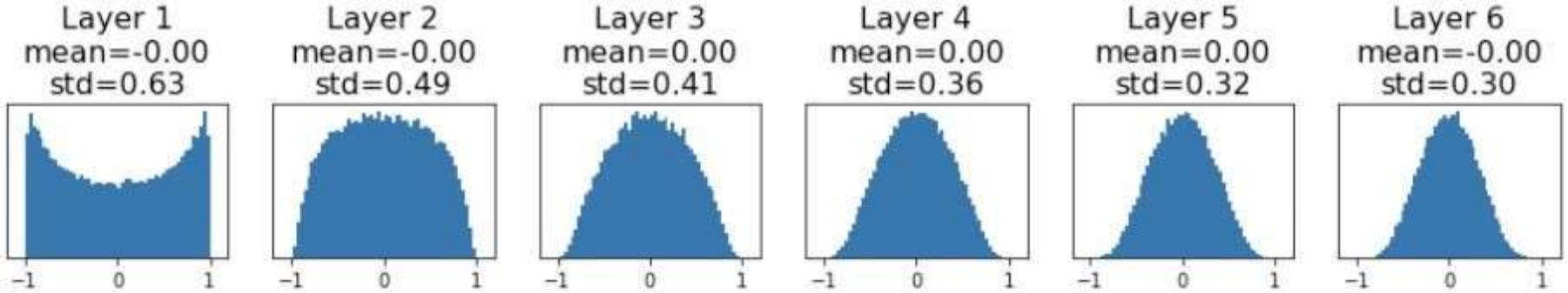
权重初始化:“Xavier”初始化

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

“Xavier” 初始化:
std = 1/sqrt(Din)

“刚刚好”: 激活在所有的层都得到了很好的缩放！

对于卷积层来说, Din = filter_size^2 * input_channels



Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

权重初始化: “Xavier” Initialization

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

“Xavier” initialization:
std = 1/sqrt(Din)

“刚刚好”: 激活在所有的层都得到了很好的缩放！

对于卷积层来说,
Din =filter_size^2 * input_channels

y = Wx
h = f(y)

推导:

$Var(y_i) = Din * Var(x_i w_i)$ [假设 x, w 独立同分布]

$= Din * (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2)$ [假设 x, w相互独立]

$= Din * Var(x_i) * Var(w_i)$ [假设 x, w以0为均值]

如果$Var(w_i) = 1/Din$ 那么 $Var(y_i) = Var(x_i)$

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

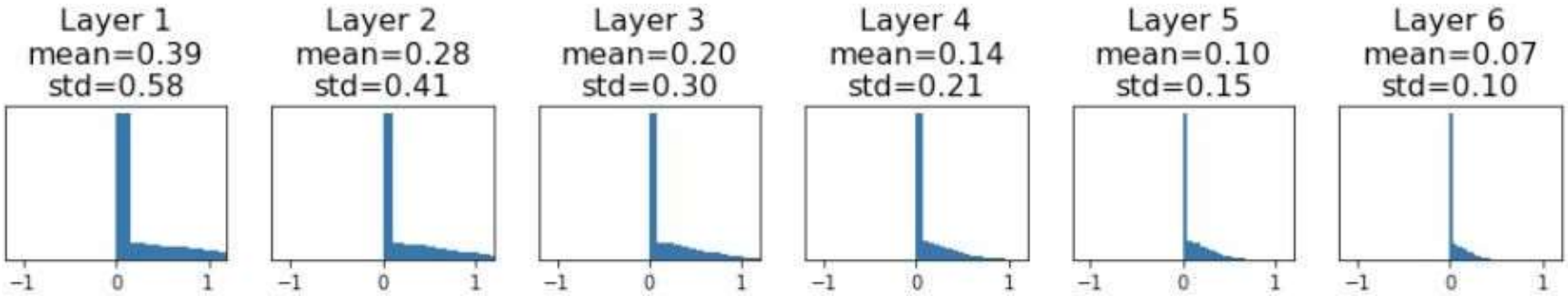
权重初始化: ReLU怎么样？

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

将tanh改为ReLU

Xavier 假设0中心激活函数

激活再次崩溃为零，没有学习

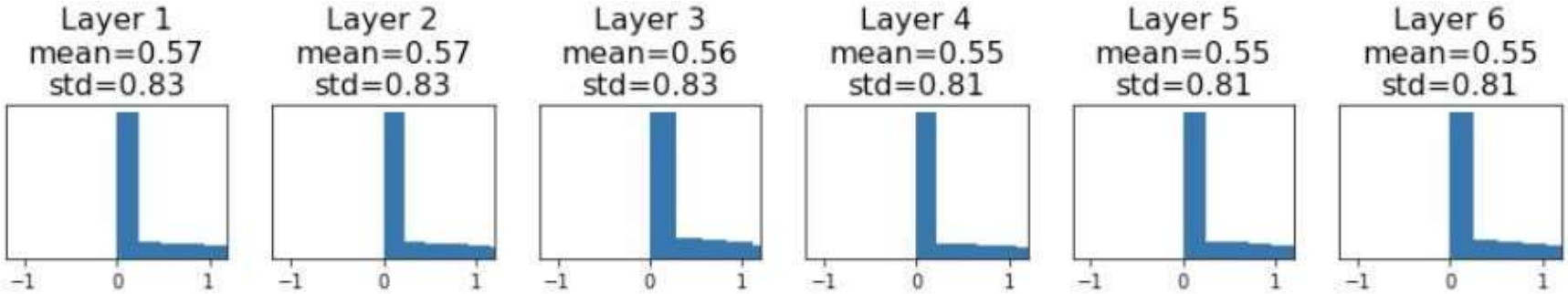


权重初始化: Kaiming / MSRA 初始化

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

ReLU correction: std = sqrt(2 / Din)

“刚刚好”: 激活在所有的层都得到了很好的缩放！



He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

寻找合适的初始化是一个活跃的研究领域…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019

批归一化

批归一化

[Ioffe and Szegedy, 2015]

“你想要零均值单位方差激活？直接这样做就可以了。”

考虑某个层上的一批激活. 要使每个维度均值为0, 方差为1,应用下面的式子:

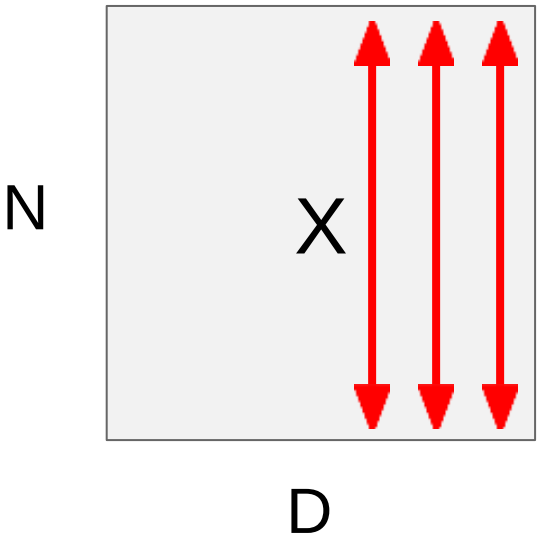
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

这是一个普通的可微函数...

批归一化

[Ioffe and Szegedy, 2015]

输入: $x : N \times D$



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

每个通道的均值,
形状是D

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

每个通道的方差,
形状是D

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

归一化后的x,
形状是N x D

问题: 如果零均值单位方差是一个很难的约束怎么办?

批归一化

[Ioffe and Szegedy, 2015]

输入: $x : N \times D$

可学习的放缩和移位参数:

$$\gamma, \beta : D$$

当 $\gamma = \sigma$, $\beta = \mu$时
会恢复为恒等函数!

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

每个通道的均值,
形状是D

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

每个通道的方差,
形状是D

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

归一化后的x,
形状是N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

批归一化:测试时

估计取决于小批量, 测试时不能进行批归一化！

输入: $x : N \times D$

可学习的放缩和移位参数:

$$\gamma, \beta : D$$

当 $\gamma = \sigma$, $\beta = \mu$时
会恢复为恒等函数！

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

每个通道的均值,
形状是D

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

每个通道的方差,
形状是D

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

归一化后的x,
形状是N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

批归一化：测试时

输入： $x : N \times D$

$$\mu_j = \text{(Running) 在训练中看到的值的平均}$$

每个通道的均值，形状是D

可学习的放缩和移位参数： $\gamma, \beta : D$

$$\sigma_j^2 = \text{(Running) 在训练中看到的值的平均}$$

每个通道的方差，形状是D

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

归一化后的x，形状是N x D

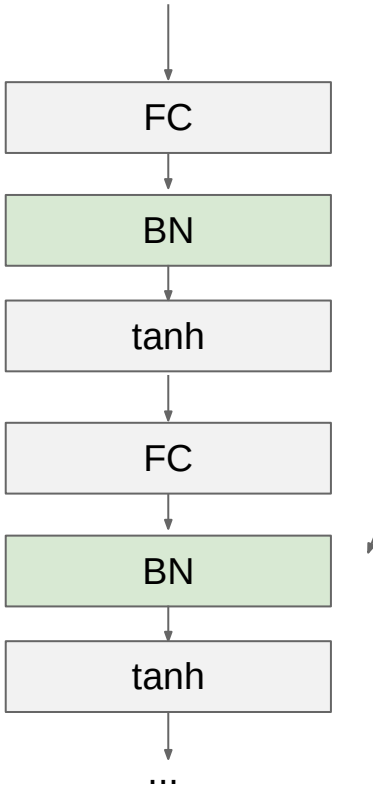
$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

输出，形状是N x D

在测试的时候批归一化变成了线性操作！ 可以和之前的全连接层和卷积层进行混合

批归一化

[Ioffe and Szegedy, 2015]

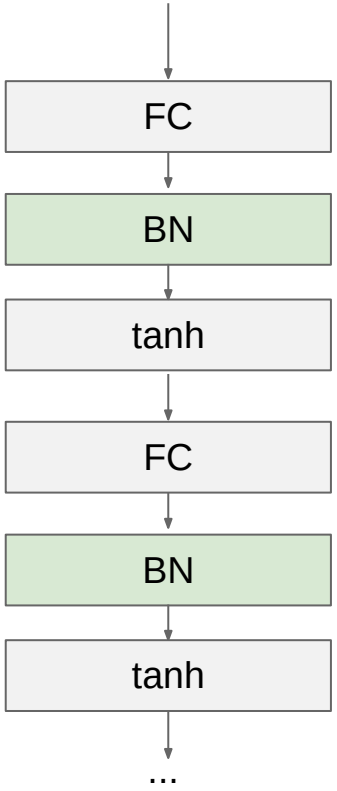


通常在全连接层或卷积层的后面插入，在非线性前面插入。

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

批归一化

[Ioffe and Szegedy, 2015]



- 使深度网络更容易训练!
- 改善了梯度流动
- 允许使用更高的学习率, 加速收敛
- 网络对于初始化更加鲁棒
- 在训练中作为正则化
- 测试时的开销是0: 可以和卷积层结合!
- 在训练和测试中表现不同: 这是一个非常常见的bug来源!

卷积层的批归一化

全连接网络的批归一化

$x: N \times D$

归一化 ↓

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

卷积网络的批归一化
(Spatial Batchnorm, BatchNorm2D)

$x: N \times C \times H \times W$

归一化 ↓ ↓ ↓

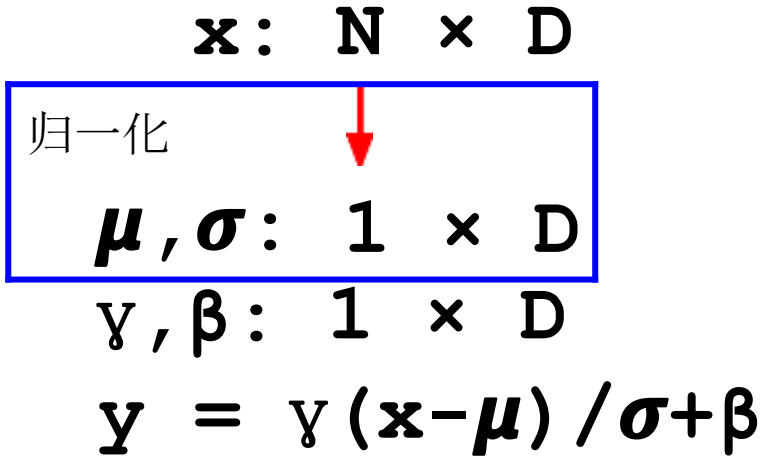
$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$

层归一化

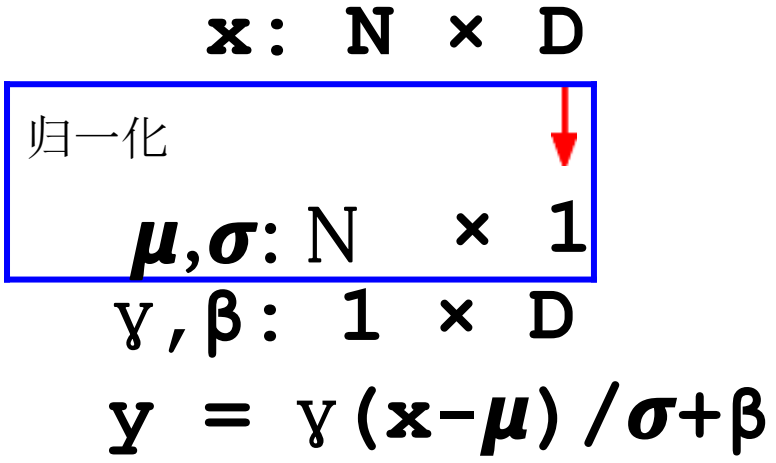
全连接网络的批归一化



全连接网络的层归一化

训练和测试的行为相同!

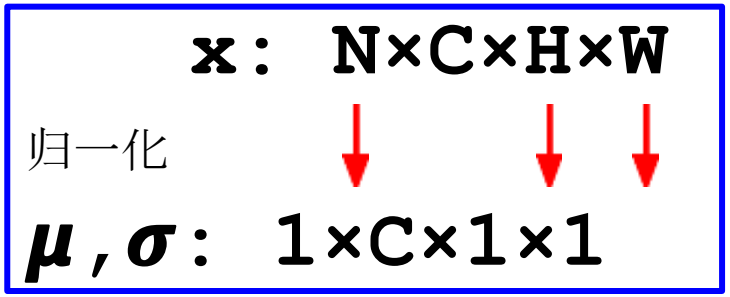
可以用于递归网络



Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

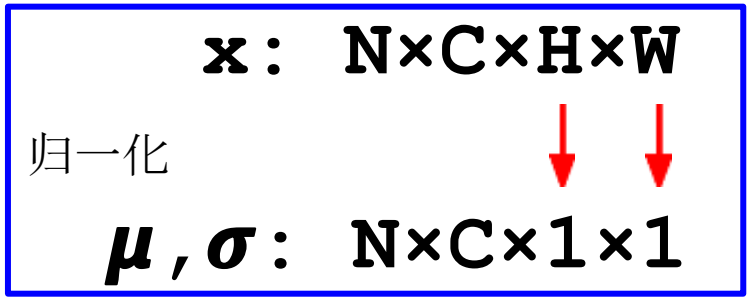
实例归一化

卷积网络的批归一化



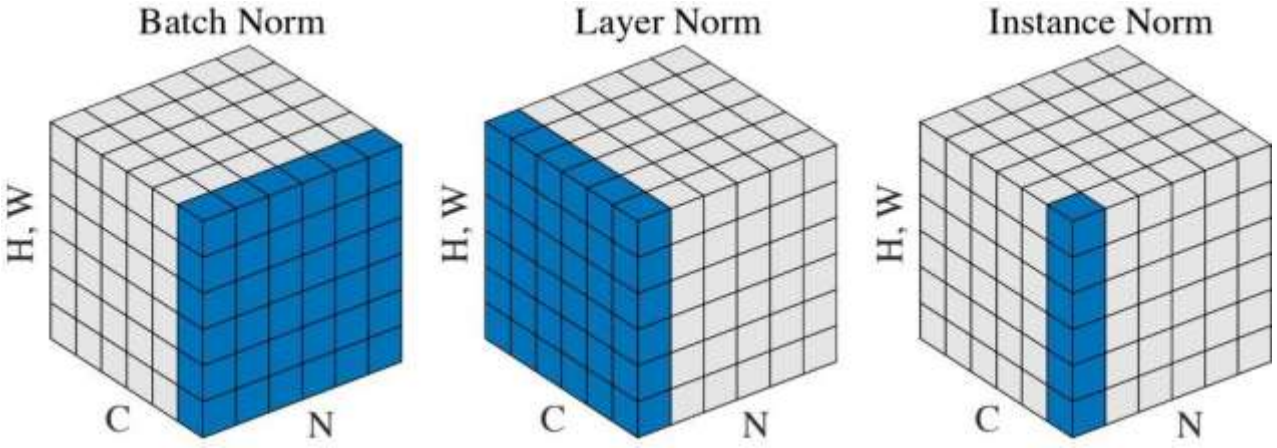
卷积网络的实例归一化

训练/测试的行为相同!



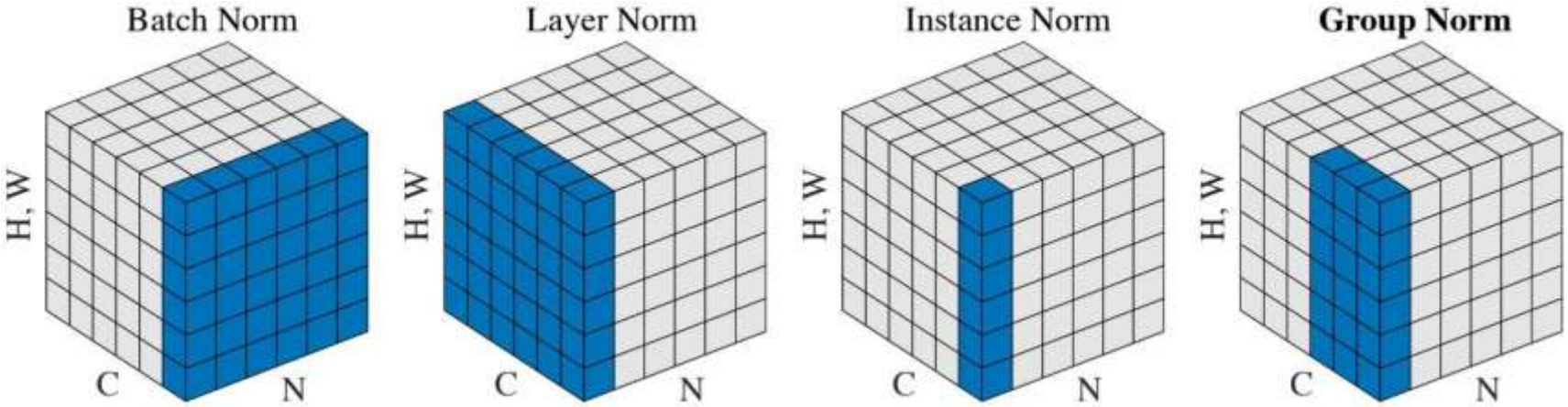
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

归一化层的对比



Wu and He, “Group Normalization”, ECCV 2018

组归一化



Wu and He, “Group Normalization”, ECCV 2018

迁移学习

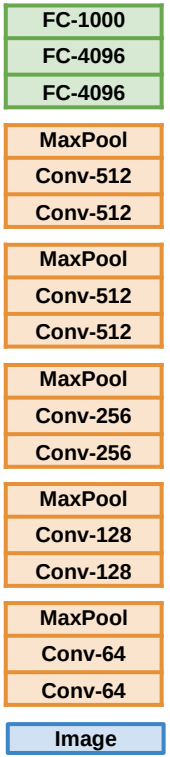
“如果要训练/使用CNNs，你需要很多数据”

“如果要训练/使用CNNs，你需要很多数据”

被否定了

CNNs的迁移学习

1. 在Imagenet上训练

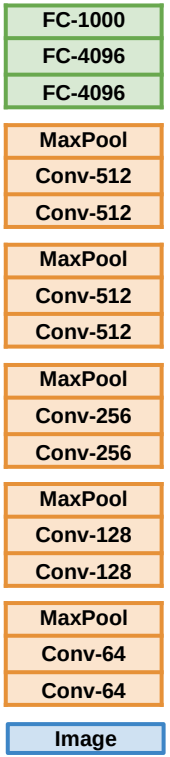


Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014 Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

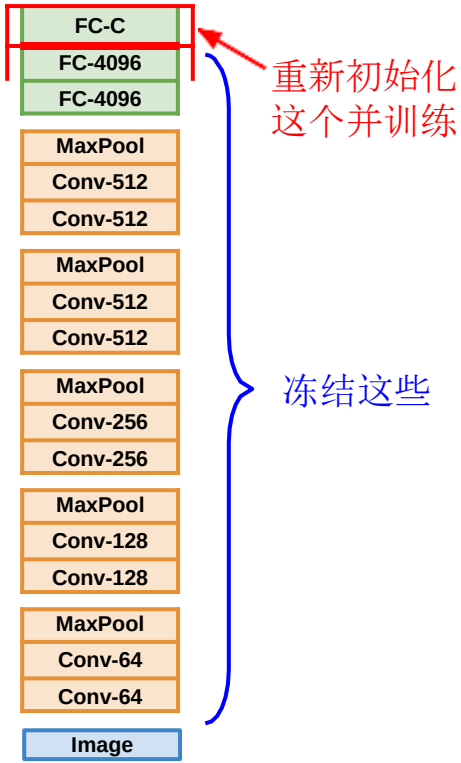
CNNs的迁移学习

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014 Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

1. 在Imagenet上训练



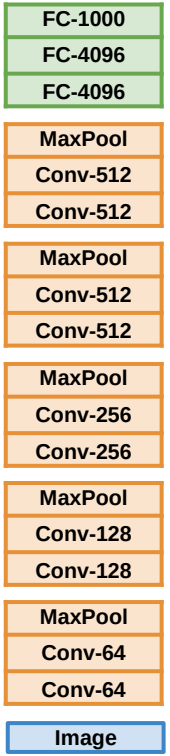
2. 小数据集(C个类别)



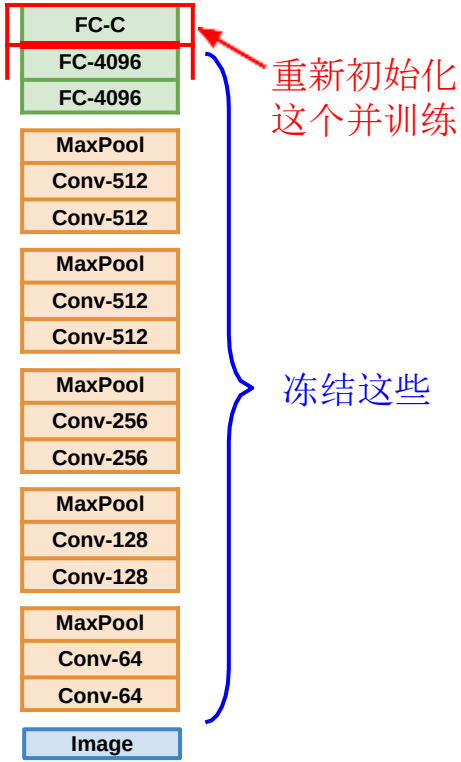
CNNs的迁移学习

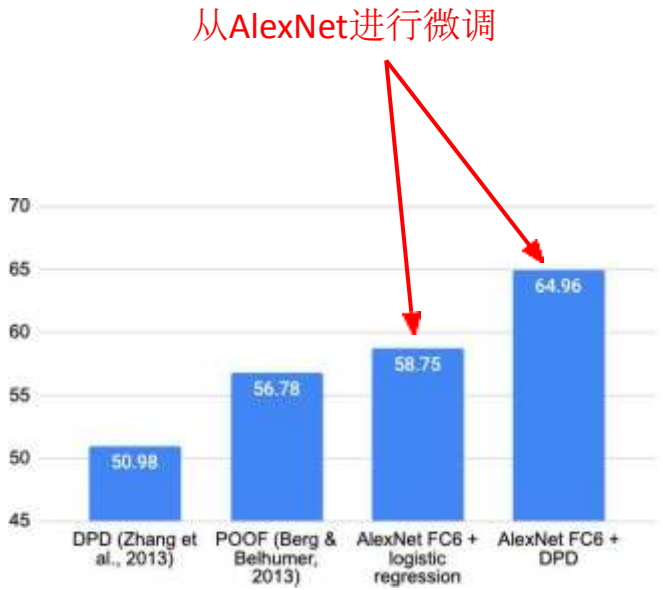
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014 Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. 在Imagenet上训练



2. 小数据集(C个类别)

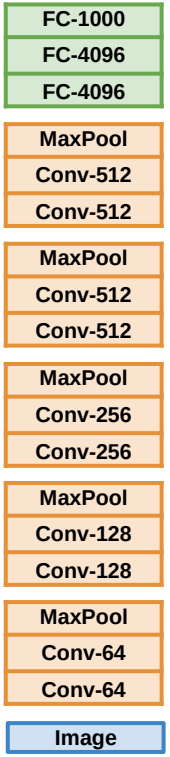




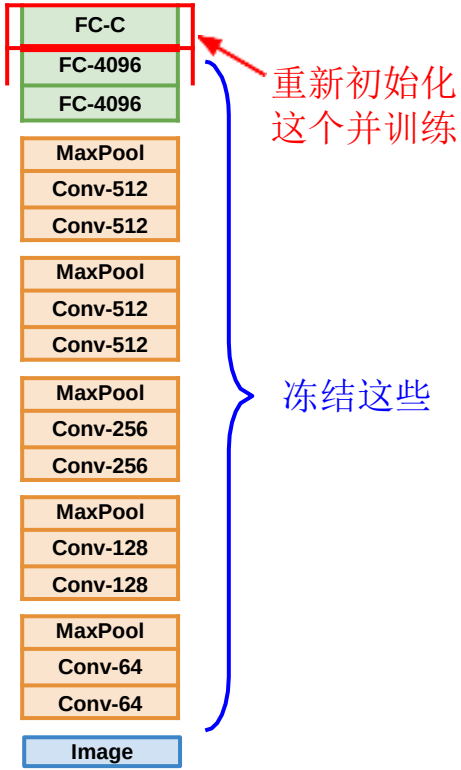
CNNs的迁移学习

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014 Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

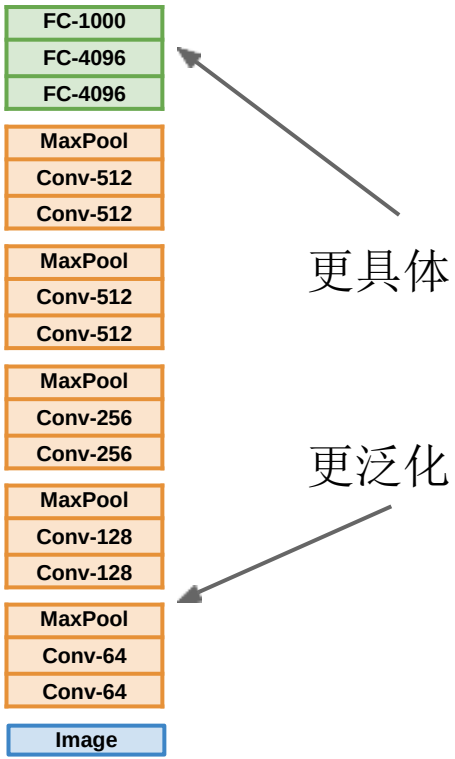
1. 在Imagenet上训练



2. 小数据集(C个类别)



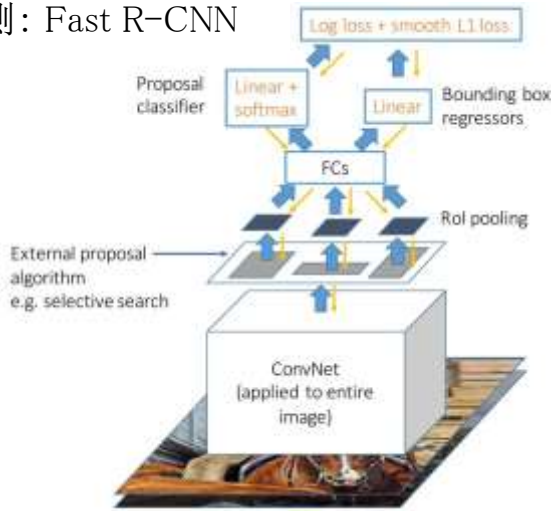
3. 大数据集



	很相似的数据集	很不同的数据集
很少的数据集	在顶层使用线性分类器	你有麻烦了… 尝试不同阶段的线性分类器
很多的数据集	对一些层进行微调	微调更多的层

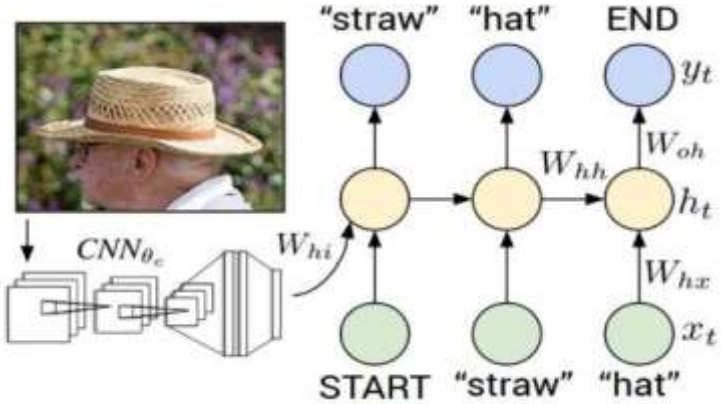
CNNs的迁移学习是无处不在的 …
(这是常态, 不是例外)

目标检测: Fast R-CNN



Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

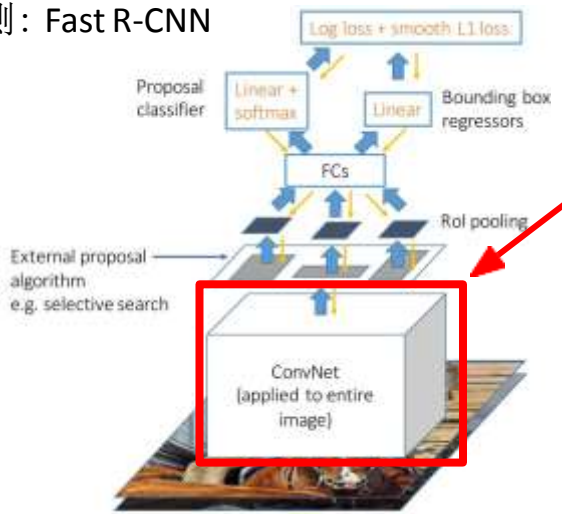
图像标注: CNN + RNN



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

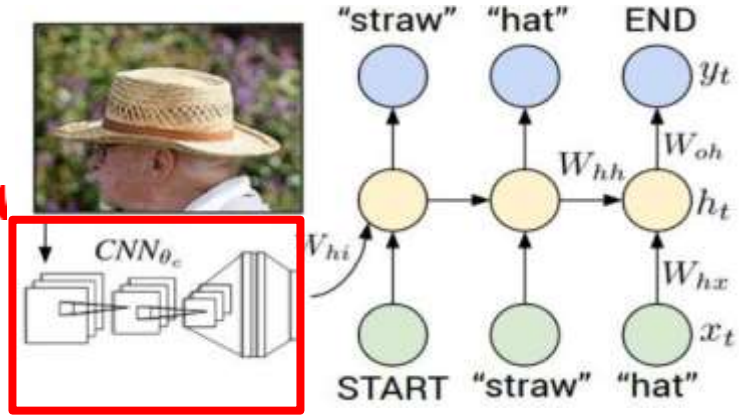
CNNs的迁移学习是无处不在的 …
(这是常态, 不是例外)

目标检测: Fast R-CNN



在ImageNet上进行预训练的CNN

图像标注: CNN + RNN

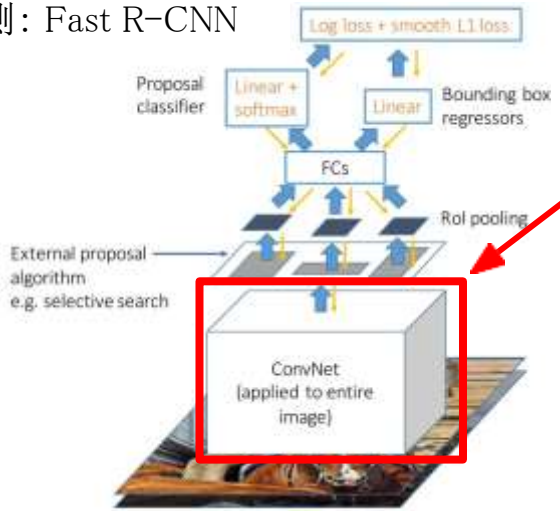


Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

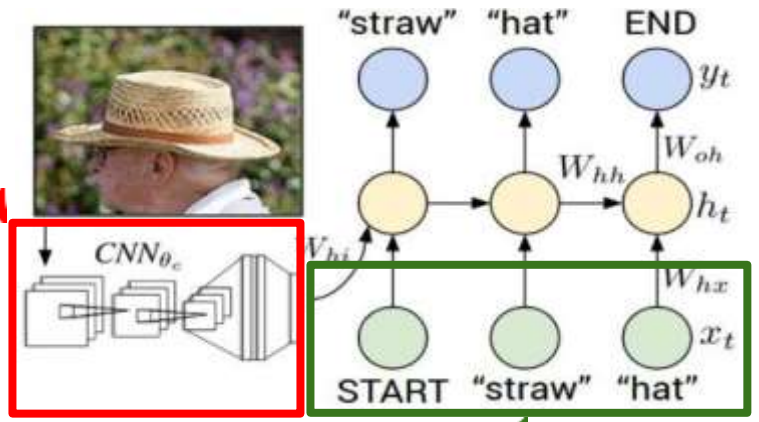
CNNs的迁移学习是无处不在的 …
(这是常态，不是例外)

目标检测：Fast R-CNN



在ImageNet上进行预训练的CNN

图像标注：CNN + RNN



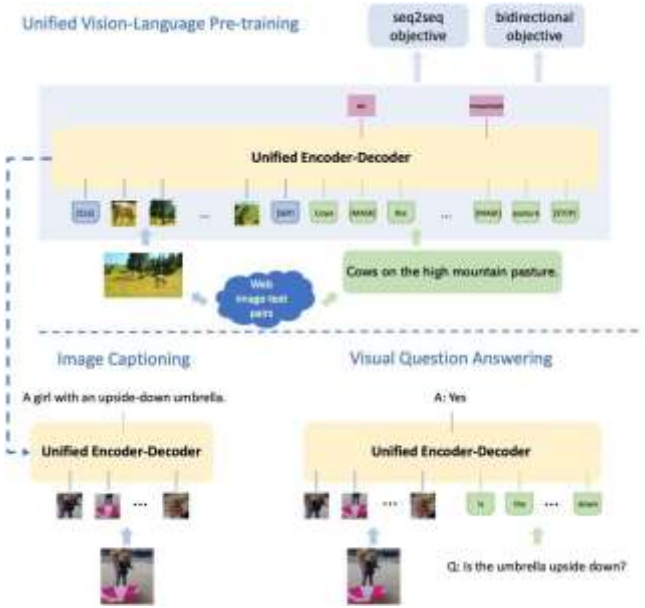
用word2vec预训练的词向量

Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

CNNs的迁移学习是无处不在的 …

(这是常态, 不是例外)



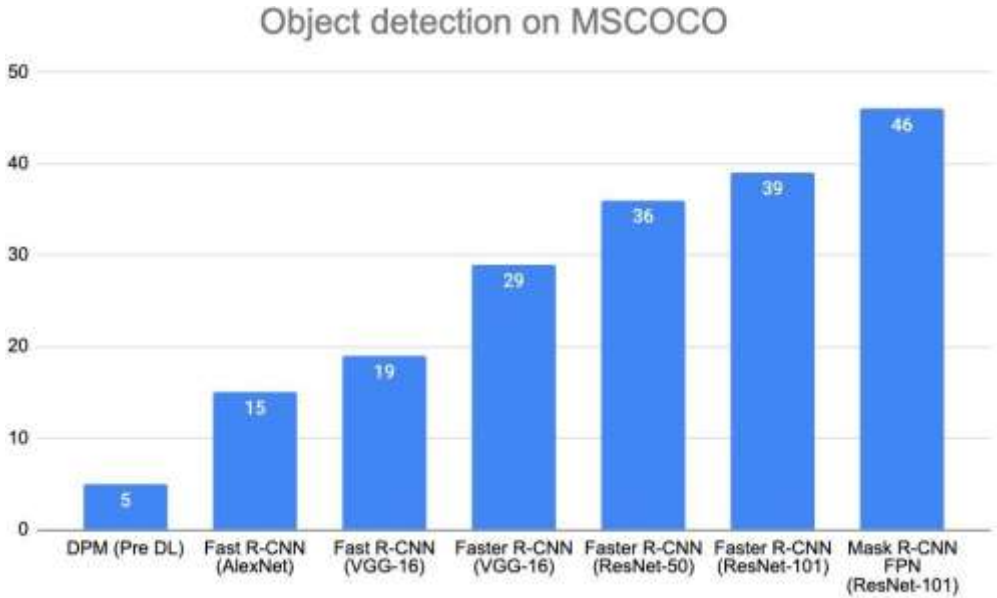
1. 在ImageNet上训练CNN
2. 微调 (1) 用于视觉基因组上的目标检测
3. 在大量文本上训练 BERT 语言模型
4. 结合(2) 和 (3), 共同训练图像/语言模型
5. 微调 (4) 用于图像标注, 视觉问题回答, 等等.

Zhou et al, "Unified Vision-Language Pre-Training for Image Captioning and VQA" CVPR 2020
Figure copyright Luowei Zhou, 2020. Reproduced with permission.

Krishna et al, "Visual genome: Connecting language and vision using crowdsourced dense image annotations" IJCV 2017
Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" ArXiv 2018

CNNs的迁移学习-

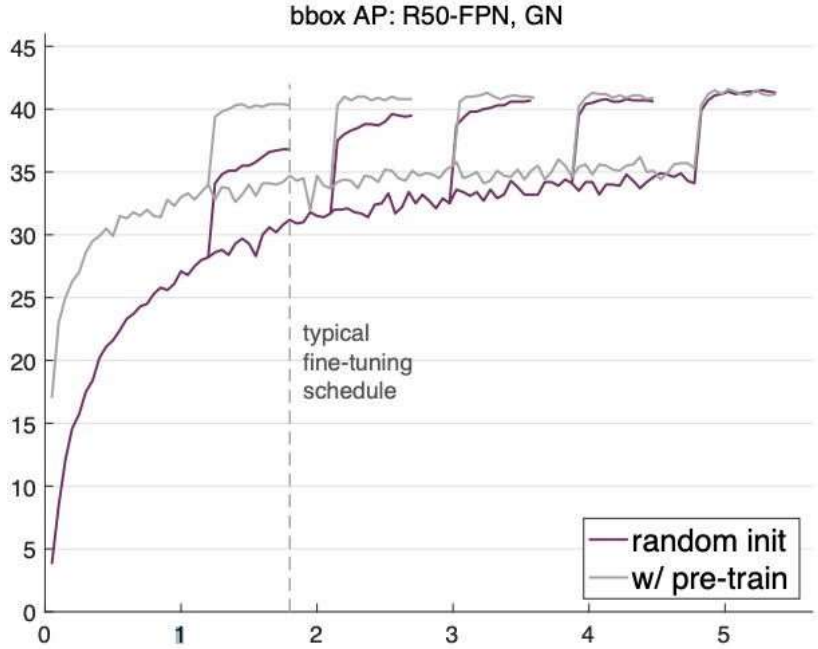
网络结构对迁移学习的影响



Girshick, “The Generalized R-CNN Framework for Object Detection”, ICCV 2017 Tutorial on Instance-Level Visual Recognition

CNNs的迁移学习是无处不在的 …

但最近的研究结果显示, 这可能并不总是必要的!



He et al, “Rethinking ImageNet Pre-training”, ICCV 2019
Figure copyright Kaiming He, 2019. Reproduced with permission.

从头开始训练和从在ImageNet上预训练好的模型开始训练用于目标检测效果一样好。

但是需要花费2倍到3倍的时间进行训练。

他们还发现收集更多的数据比对相关任务进行微调要好。

用于你的项目：
如果你有一个有趣的数据集但是它<~1百万张图片？

1. 找到一个有相似数据的非常大的数据集，
   在它上面训练一个非常大的卷积网络
2. 迁移学习到你的数据集

深度学习框架提供预训练模型的“模型动物园”，
因此你不需要自己再重复训练。

TensorFlow:https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision

Source: AI & Deep Learning Memes For Back-propagated Poets

总结

- 激活函数(使用ReLU)
- 数据预处理(图像: 减去平均值)
- 权重初始化(使用 Xavier/Kaiming 初始化)
- 批归一化(使用它!)
- 迁移学习 (如果你会用的话请使用它!)

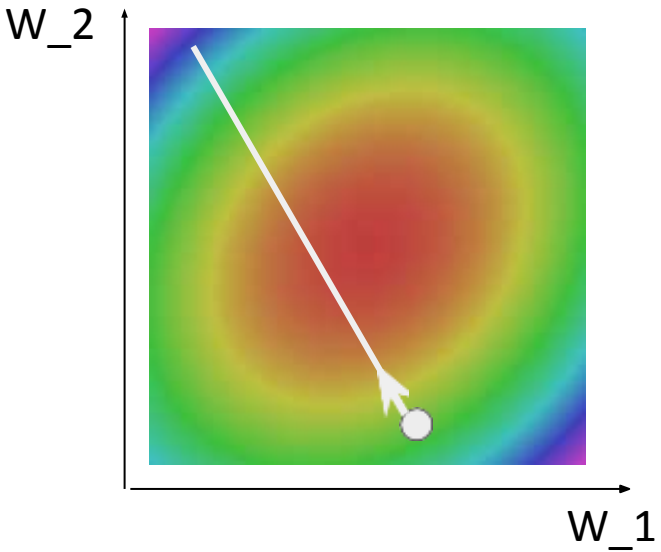
本节课内容

- 改进训练误差:
  - (更好的) 优化器
  - 学习率调整
  - 改进测试误差:
  - 正则化
  - 选择超参数

(更好的) 优化
器

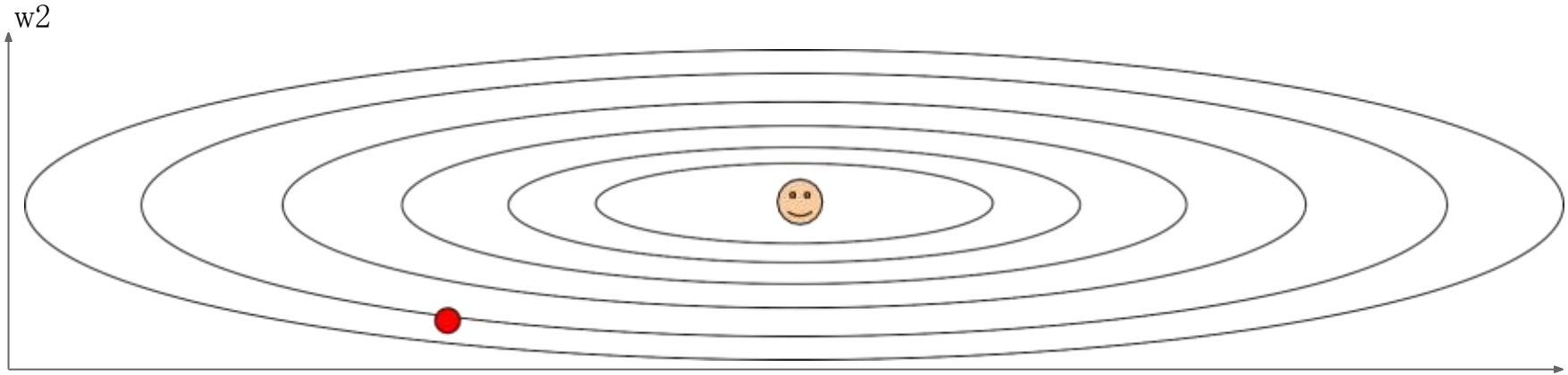
优化

```
# Vanilla Gradient Descent

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



优化: SGD的问题

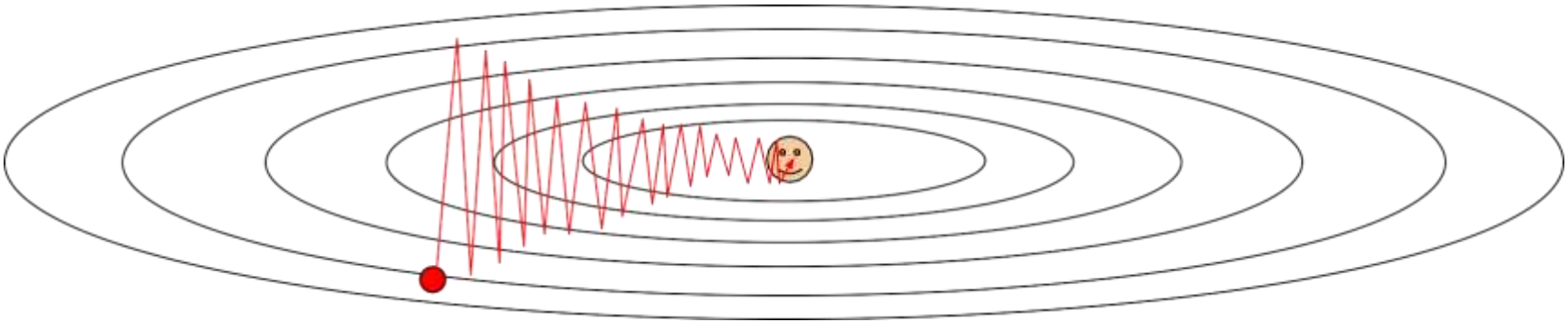
如果损失函数在一个方向变化很快却在另一个方向变化很慢怎么办?
梯度下降做了什么呢?



损失函数的条件数较大: Hessian矩阵的最大值与最小值的比值较大。

优化: SGD的问题

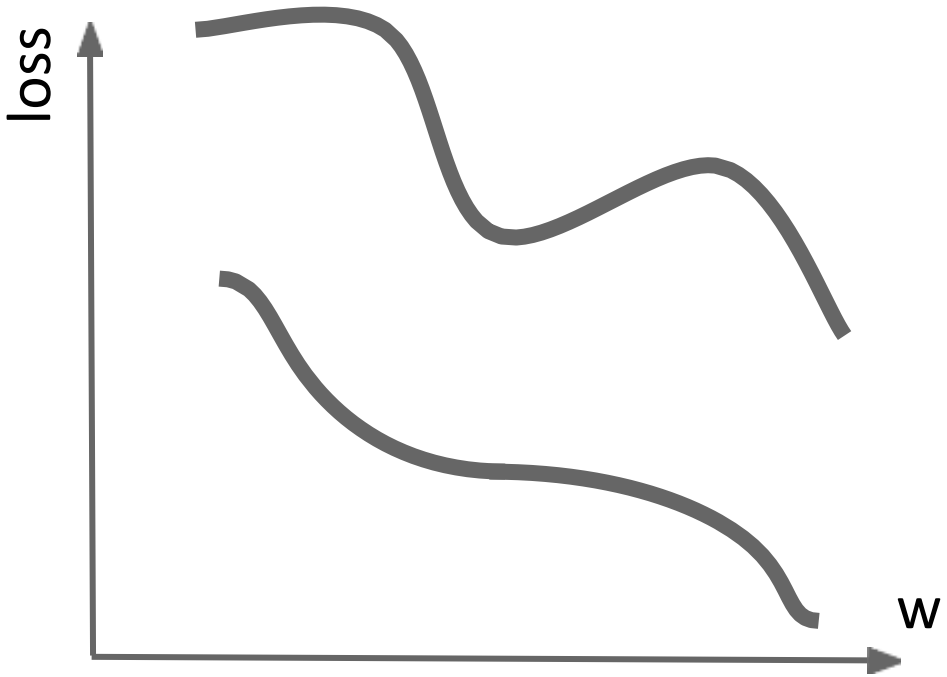
如果损失函数在一个方向变化很快却在另一个方向变化很慢怎么办?
梯度下降做了什么呢?
在梯度小的维度缓慢前进, 在梯度陡峭的维度抖动



损失函数的条件数较大: Hessian矩阵的最大值与最小值的比值较大。

优化: SGD的问题

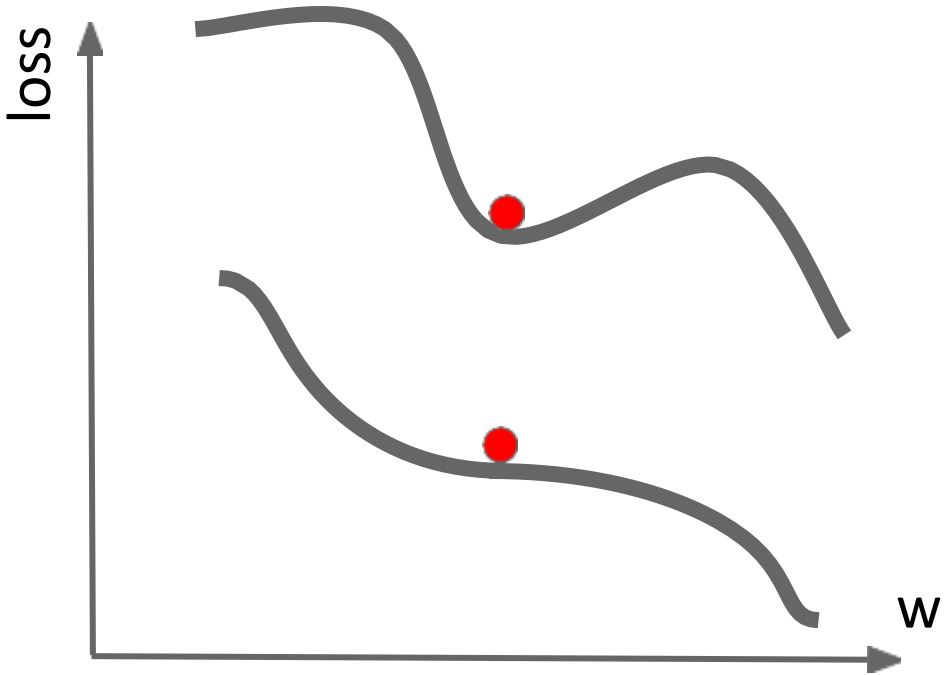
如果损失函数有**局部最小值**或者**鞍点**呢?



优化: SGD的问题

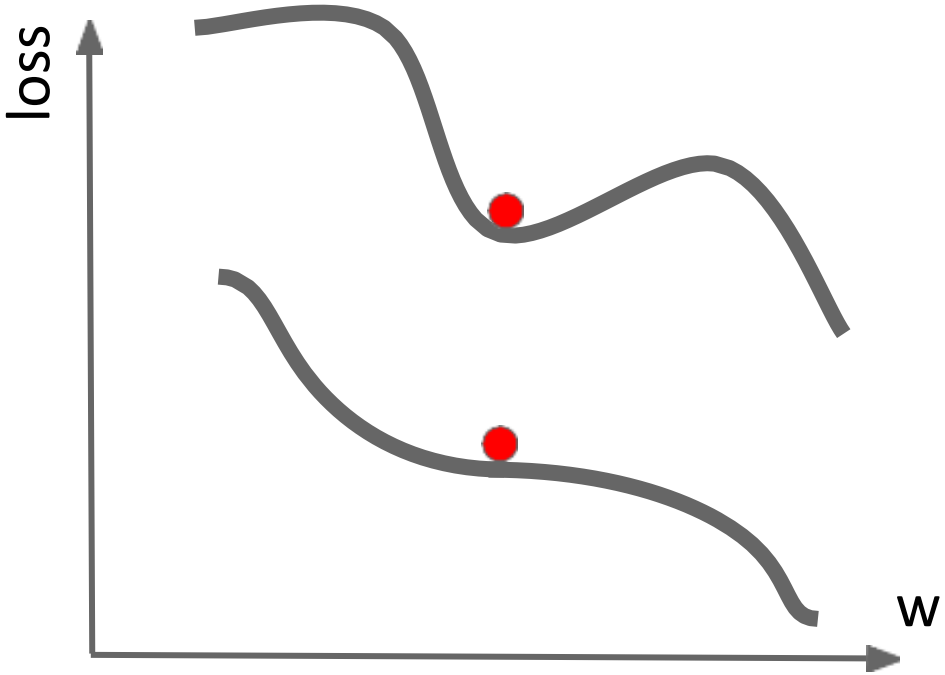
如果损失函数有
局部最小值或者
鞍点呢?

梯度为0,
梯度下降被阻塞了



优化: SGD的问题

如果损失函数有
局部最小值或者
鞍点呢?



鞍点在高维空间更常见

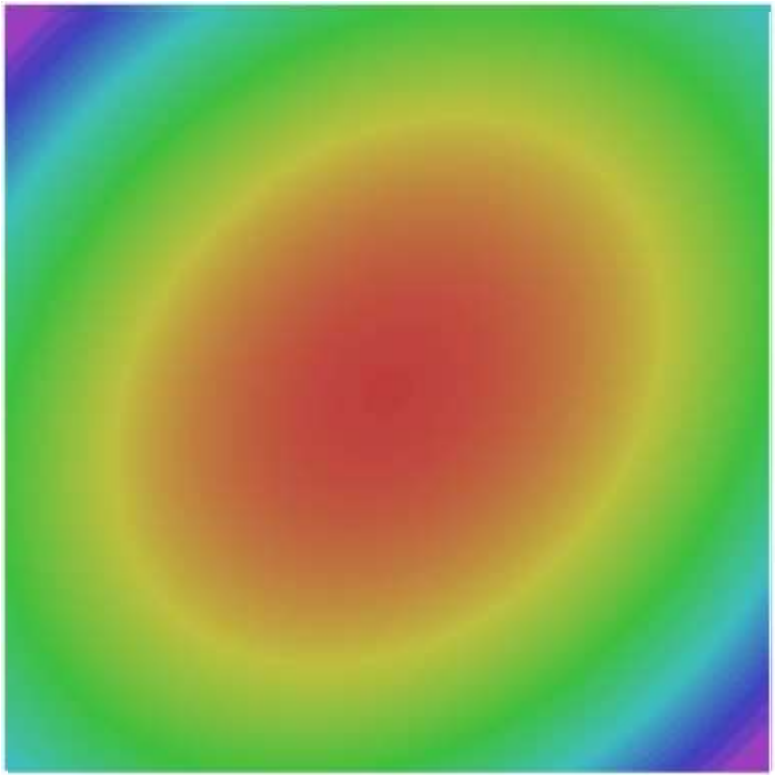
Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

优化: SGD的问题

我们的梯度来自于小批量，
所以它们可能是嘈杂的!

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

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



SGD + Momentum

SGD

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

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

SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

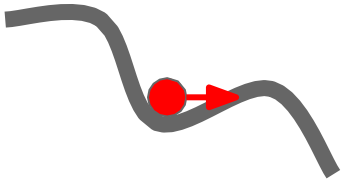
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- 引入“速度”作为梯度的渐变平均值
- rho 引入 “摩擦力”; 典型的rho取0.9或者0.99

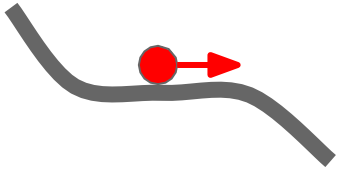
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

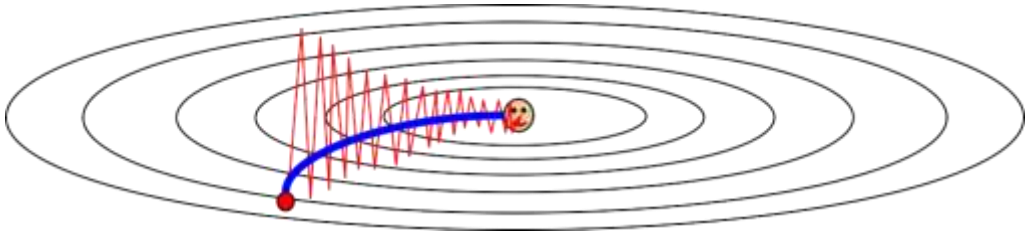
局部最小值



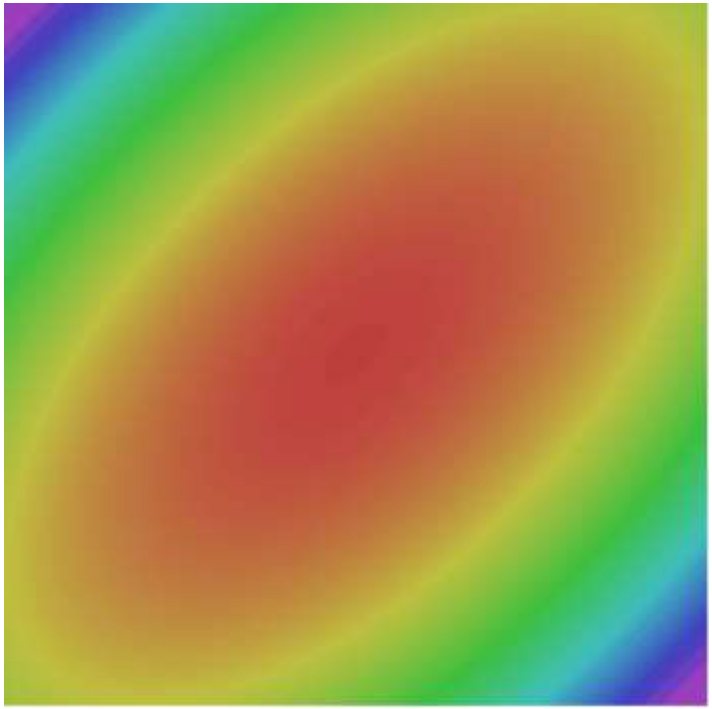
鞍点



振荡



梯度噪声



SGD SGD+Momentum

SGD + Momentum

SGD+Momentum

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

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

SGD+Momentum

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

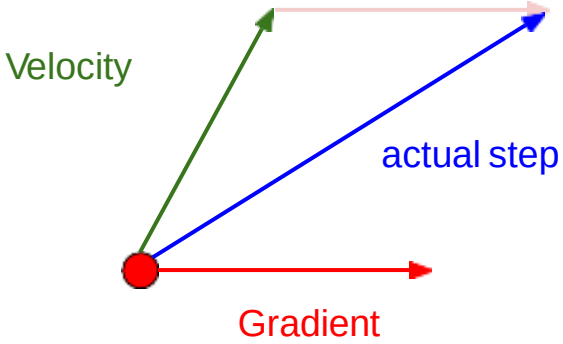
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

你可能会看到SGD+Momentum有不同的形式,
但它们在给出相同的x序列后是等价的。

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

SGD + Momentum

Momentum update:

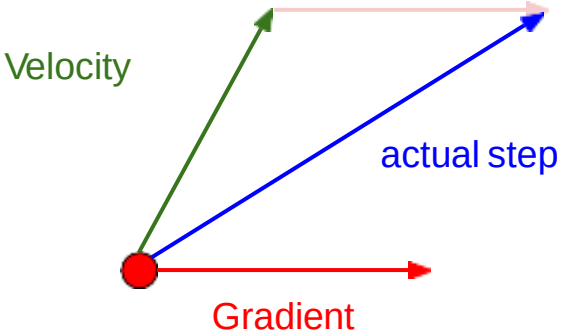


将当前点的梯度与速度结合，得到用于更新权重的步长

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

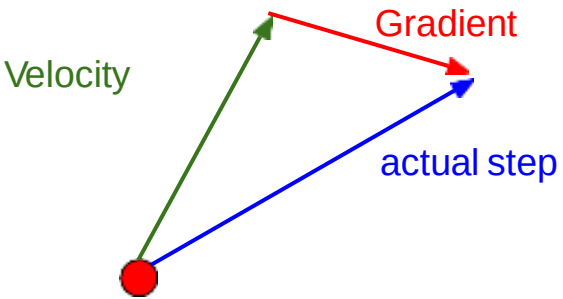
Nesterov Momentum

Momentum update:



将当前点的梯度与速度结合，
得到用于更新权重的步长。

Nesterov Momentum

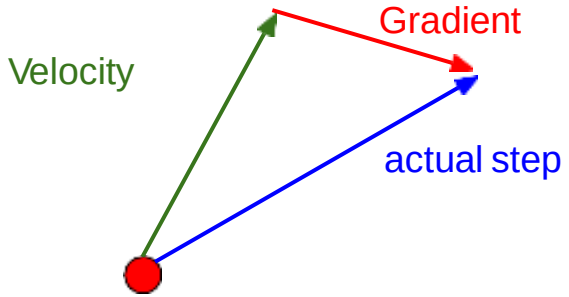


“向前看”找到使用速度进行更新将要到达的点；
在那里计算梯度并将它和速度混合得到实际的更新方向。

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Nesterov Momentum

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

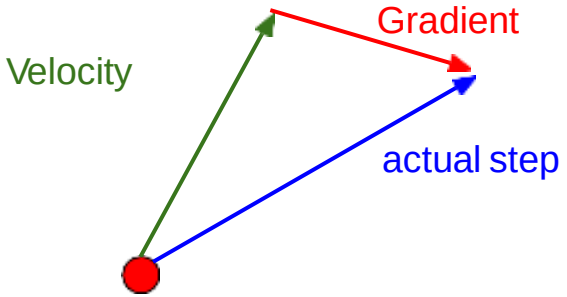


“向前看”找到使用速度进行更新将要到达的点;
在那里计算梯度并将它和速度混合得到实际的更新方向。

Nesterov Momentum

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

令人厌烦的是，我们通常希望以$x_t, \nabla f(x_t)$ 的形式进行更新。



“向前看”找到使用速度进行更新将要到达的点；
在那里计算梯度并将它和速度混合得到实际的更新方向。

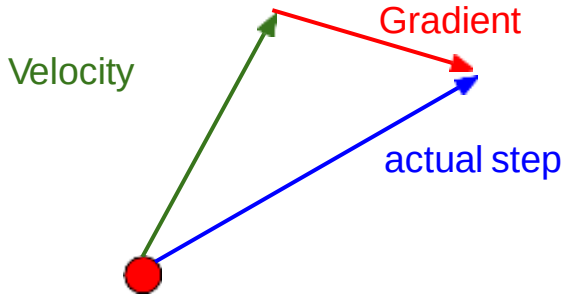
Nesterov Momentum

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

令人厌烦的是，我们通常希望以$x_t, \nabla f(x_t)$ 的形式进行更新。

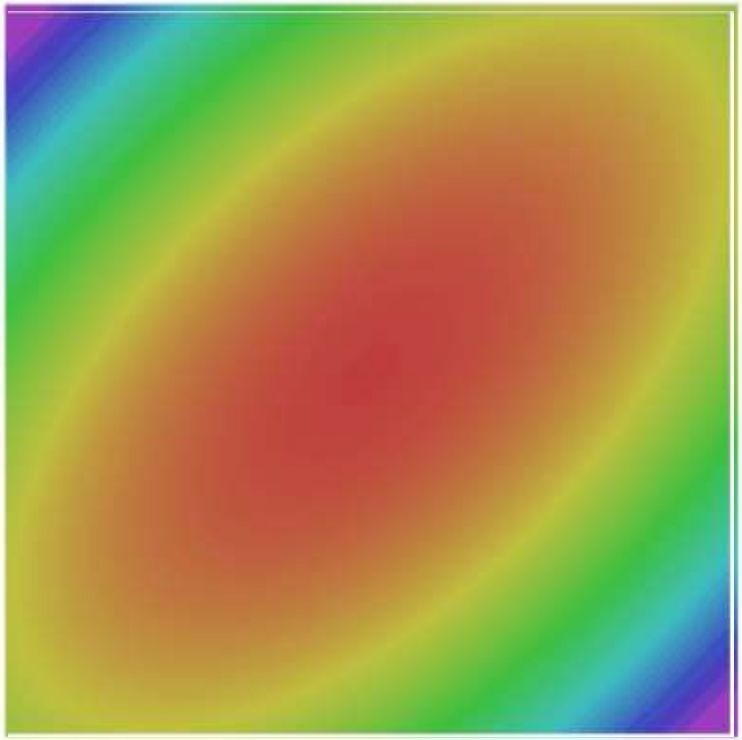
进行代换: $\tilde{x}_t = x_t + \rho v_t$

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



“向前看”找到使用速度进行更新将要到达的点;
在那里计算梯度并将它和速度混合得到实际的更新方向。

Nesterov Momentum



SGD

SGD+Momentum

Nesterov

AdaGrad

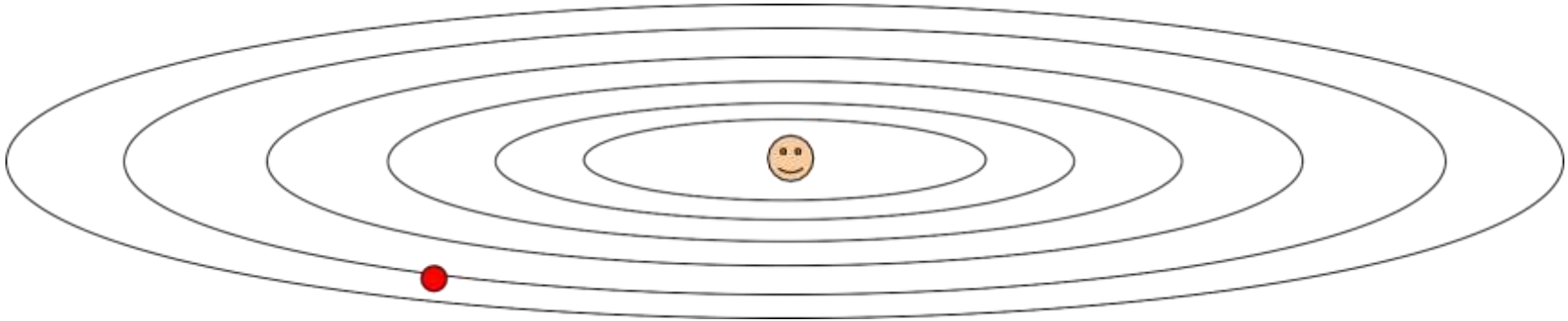
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

根据每个维度上的历史平方和，逐元素对梯度进行缩放。
称为“逐参数学习率”或者“自适应学习率”。

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

AdaGrad

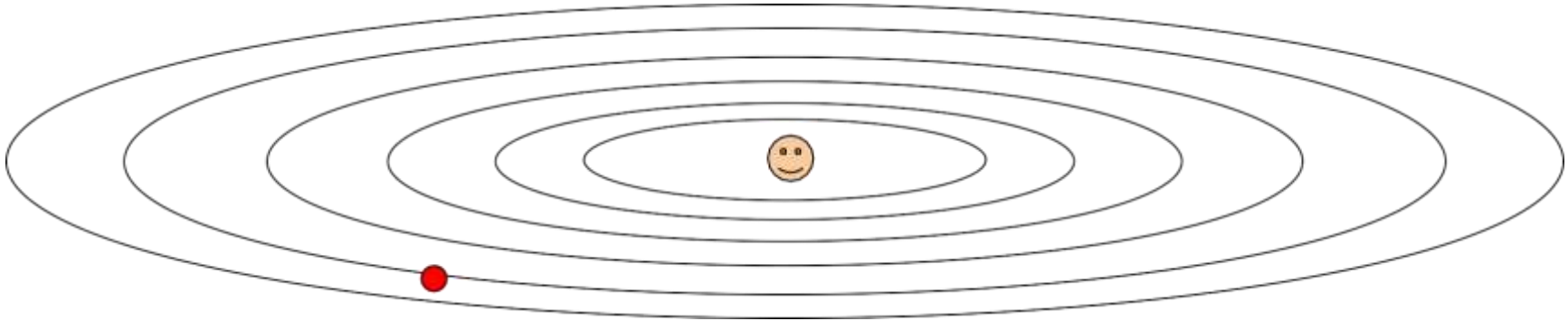
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



问题:使用AdaGrad进行梯度更新是什么样子的?

AdaGrad

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

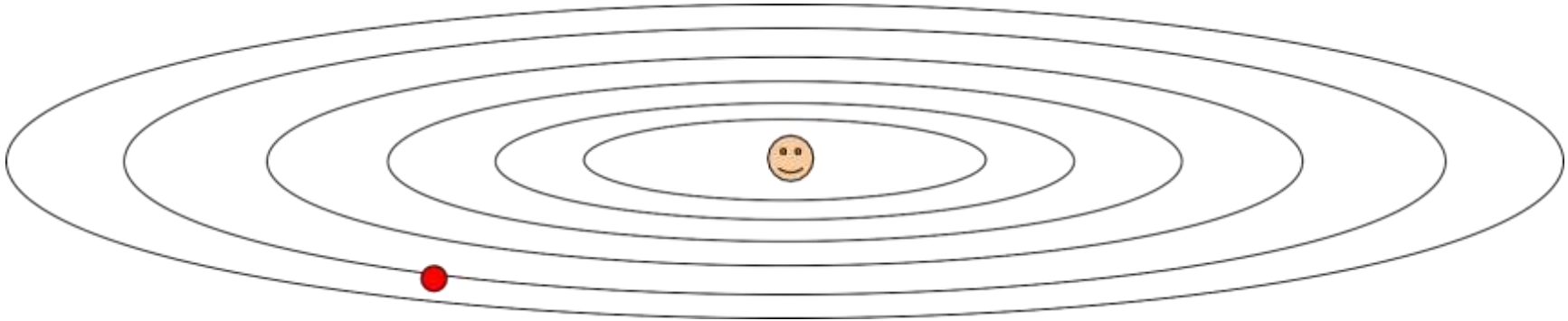


问题：使用AdaGrad进行梯度更新是什么样子的?

沿着“陡峭”方向的更新被阻碍; 沿着“平坦”方向的更新被加速。

AdaGrad

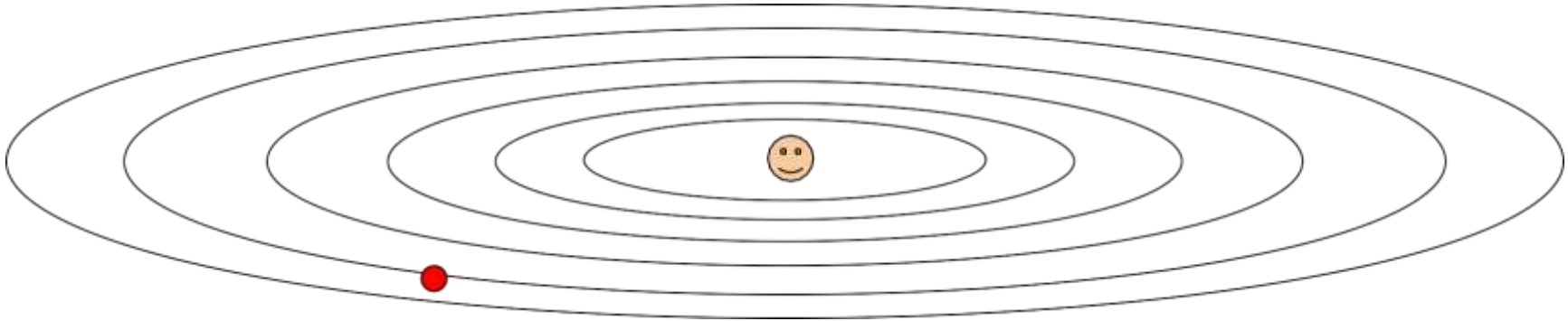
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



问题2: 步长值随着时间的变化会变成什么样?

AdaGrad

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



问题2: 步长值随着时间的变化会变成什么样?

衰减至0

RMSProp: “Leaky AdaGrad”

AdaGrad

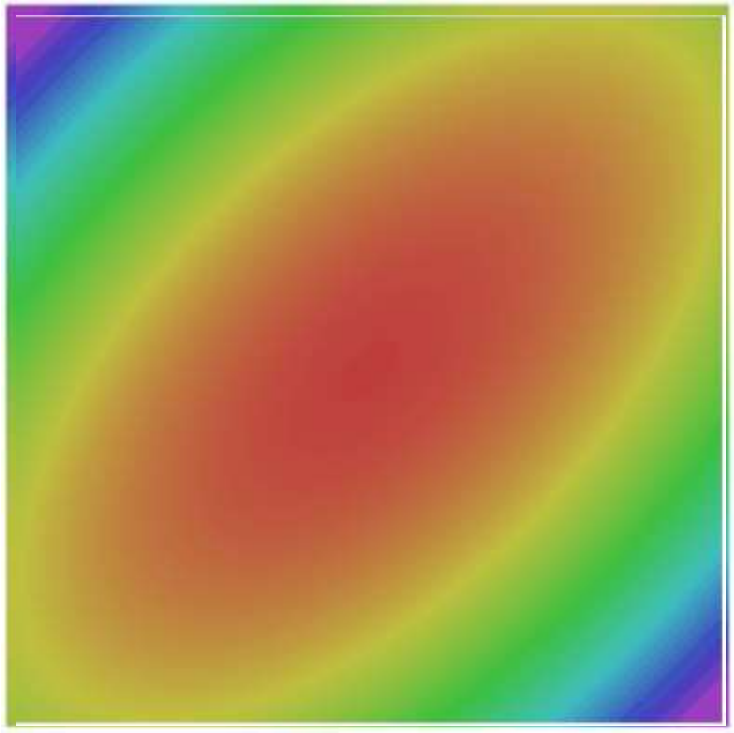
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

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

Tieleman and Hinton, 2012

RMSProp



SGD

SGD+Momentum

RMSProp

AdaGrad

(由于衰减的学习率而被阻塞)

Adam (基本形式)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Adam (基本形式)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

有点像结合了Momentum的RMSProp

问题: 第一步会发生什么?

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Adam (完整形式)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

偏差修正，由于first moment 和 second moment开始于0。

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Adam (完整形式)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

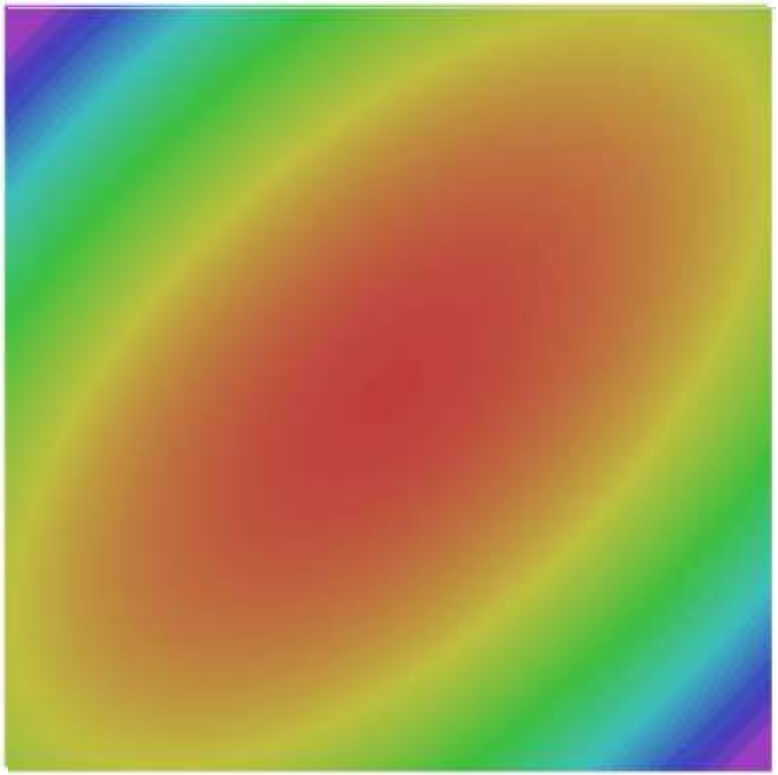
AdaGrad / RMSProp

偏差修正，
由于first moment 和 second moment开始于0。

Adam 设置beta1 = 0.9,
beta2 = 0.999, learning_rate = 1e−3 或者 5e−4
对于很多模型来说是个很好的开始!

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Adam



SGD

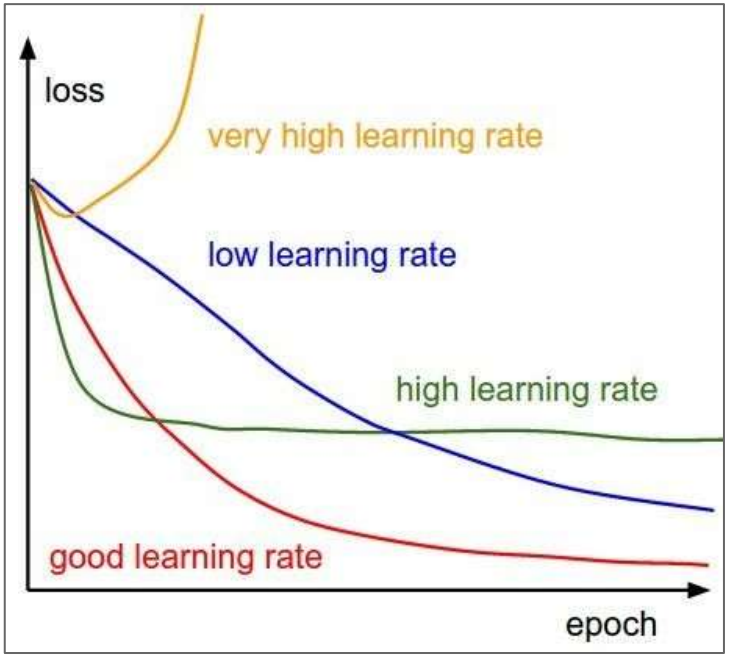
SGD+Momentum

RMSProp

Adam

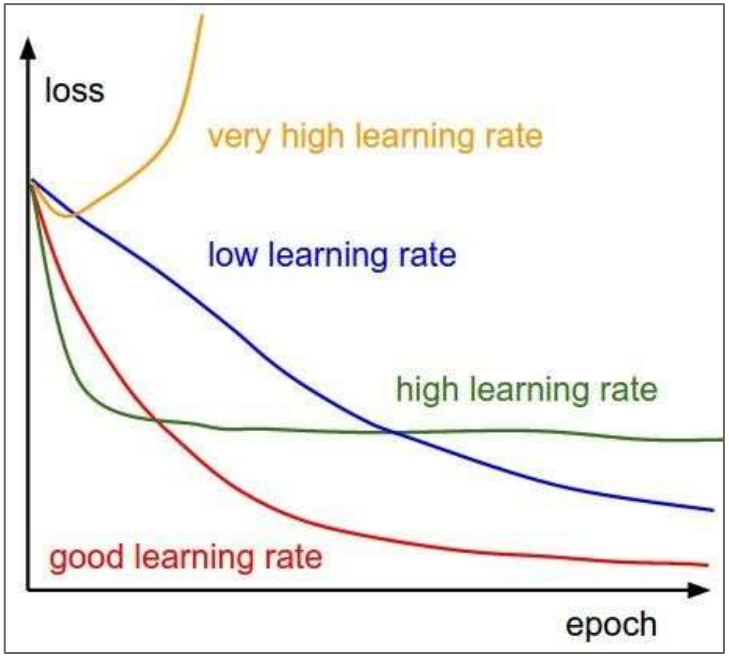
学习率调整

SGD, SGD+Momentum, Adagrad, RMSProp, Adam
都有**学习率**这一超参数。



问题: 使用哪一个学习率最好?

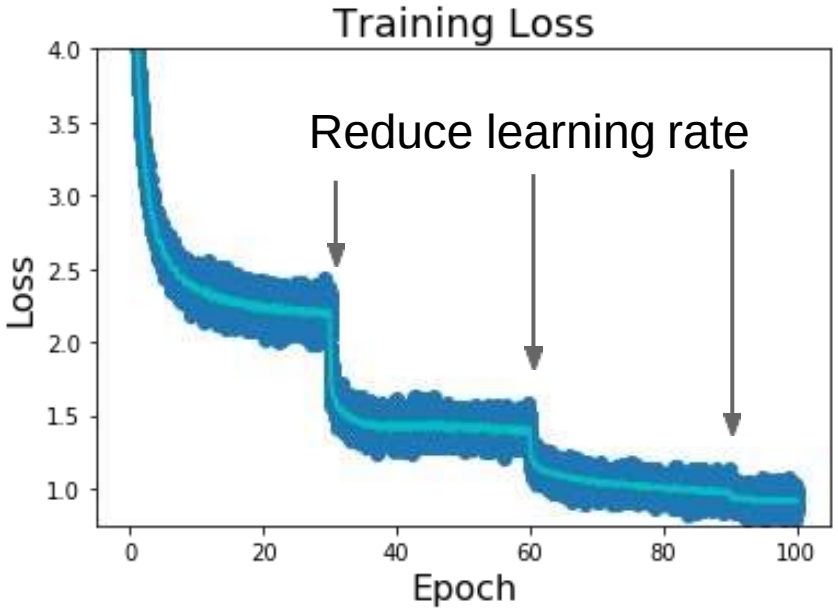
SGD, SGD+Momentum, Adagrad, RMSProp, Adam
都有**学习率**这一超参数。



问题: 使用哪一个学习率最好?

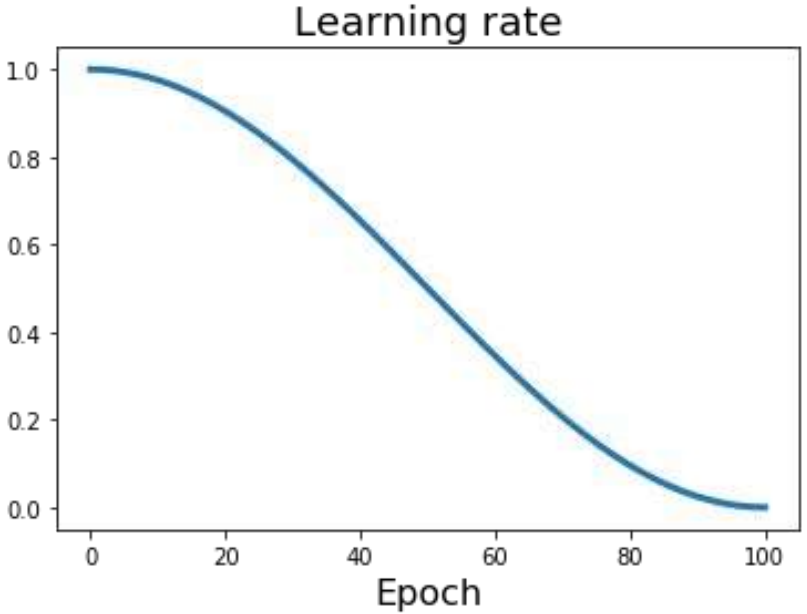
答案: 全部使用！开始的时候使用大的学习率, 随着时间进行衰减。

学习率衰减



步骤:在几个固定的点衰减学习率。
例子:对于ResNets, 分别在 30, 60 和 90个周期之后将学习率乘以0.1。

学习率衰减



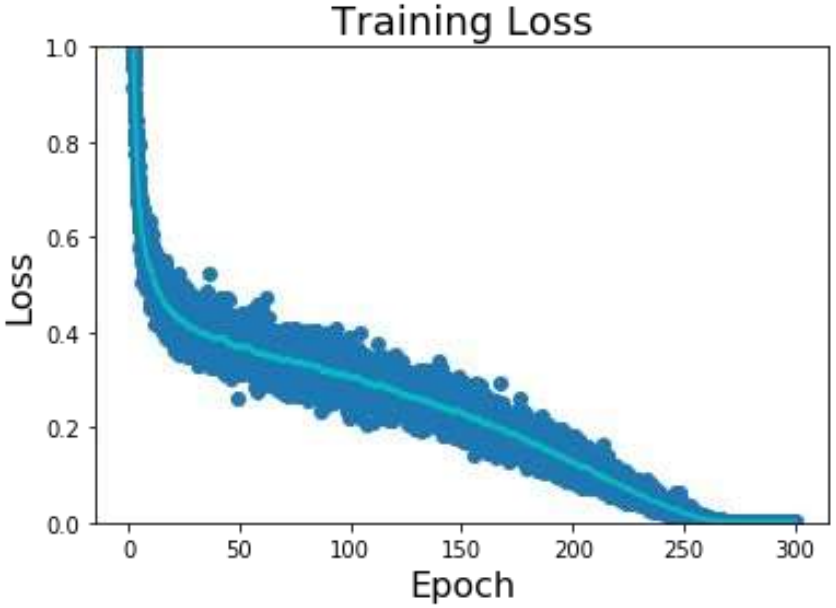
步骤:在几个固定的点衰减学习率。
例子:对于ResNets, 分别在 30, 60 和 90个周期之后将学习率乘以0.1。

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

α_0 : 初始学习率
α_t : 周期t的学习率
T : 周期的总数

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child at al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

学习率衰减



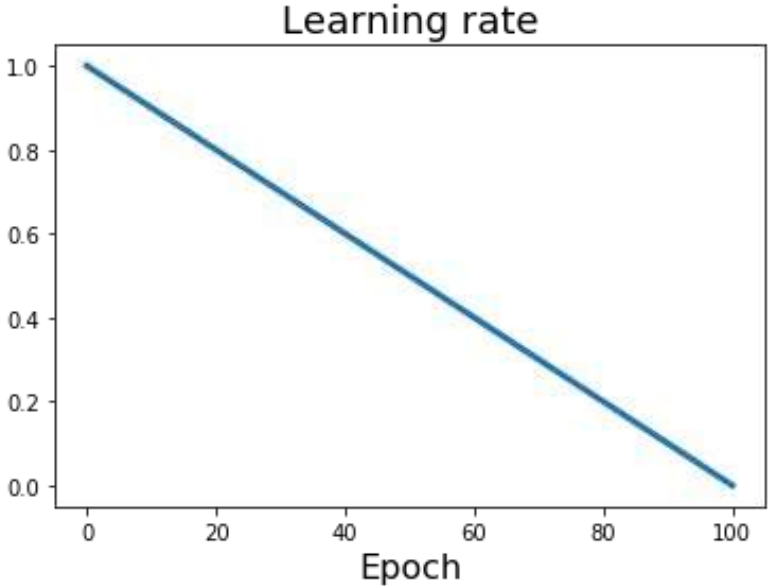
步骤:在几个固定的点衰减学习率。
例子:对于ResNets, 分别在 30, 60 和 90个周期之后将学习率乘以0.1。

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

α_0 : 初始学习率
α_t : 周期t的学习率
T : 周期的总数

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child at al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

学习率衰减



步骤:在几个固定的点衰减学习率。
例子:对于ResNets, 分别在 30, 60 和 90个周期之后将学习率乘以0.1。

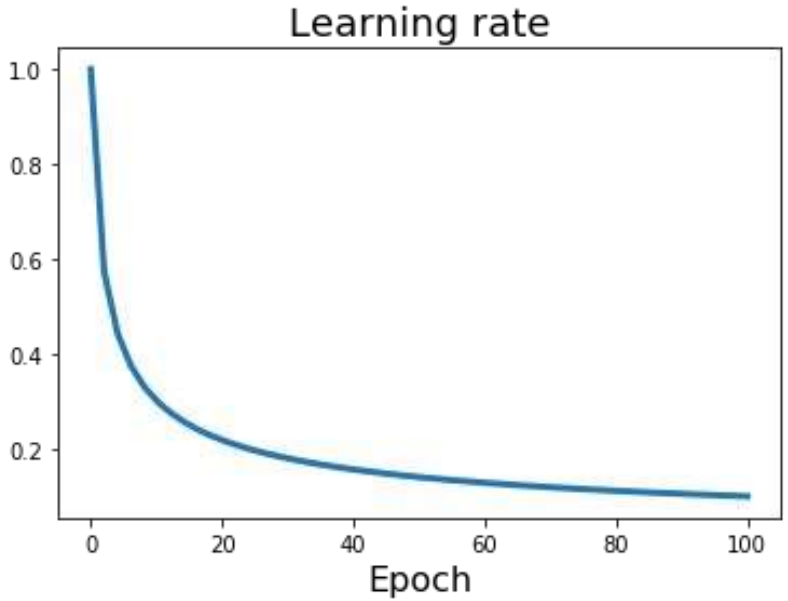
余弦: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

线性: $\alpha_t = \alpha_0(1 - t/T)$

α_0 : 初始学习率
α_t : 周期t的学习率
T : 周期的总数

Devlin et al, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, 2018

学习率衰减



步骤:在几个固定的点衰减学习率。
例子:对于ResNets, 分别在 30, 60 和 90个周期之后将学习率乘以0.1。

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

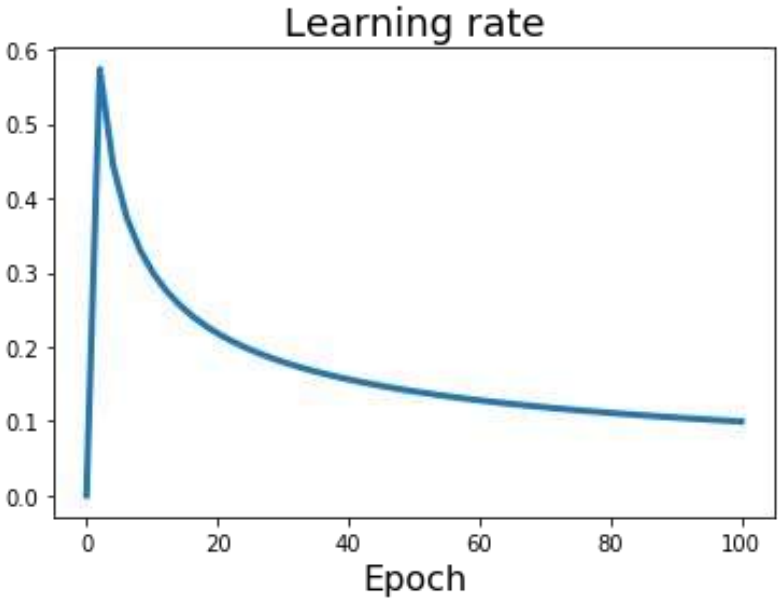
线性: $\alpha_t = \alpha_0(1 - t/T)$

反向平方根: $\alpha_t = \alpha_0/\sqrt{t}$

α_0 : 初始学习率
α_t : 周期t的学习率
T : 周期的总数

Vaswani et al, “Attention is all you need”, NIPS 2017

学习率衰减:线性热身

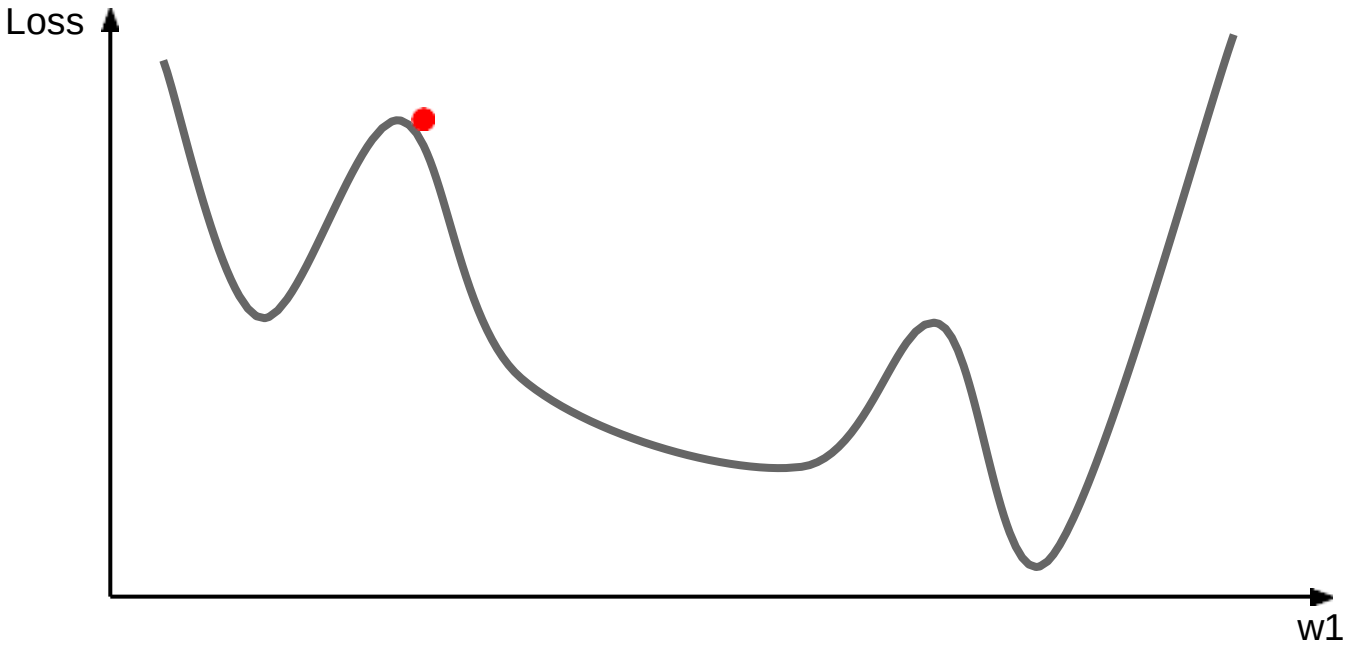


高的初始学习率会导致损失激增;
在第一个约5000次迭代中从0开始线性增加学习率可以防止这种情况发生。

经验法则:如果你将批大小增加N倍，也要将初始学习速率增加N倍。

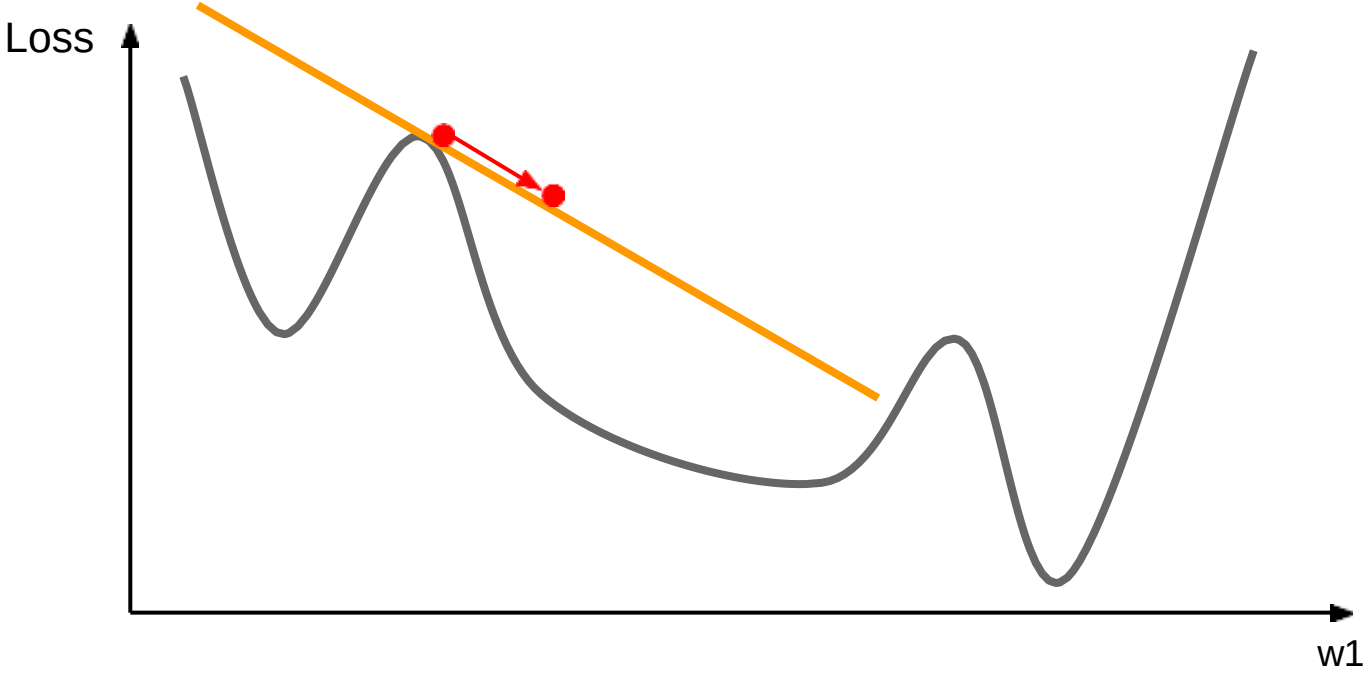
Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017

一阶优化



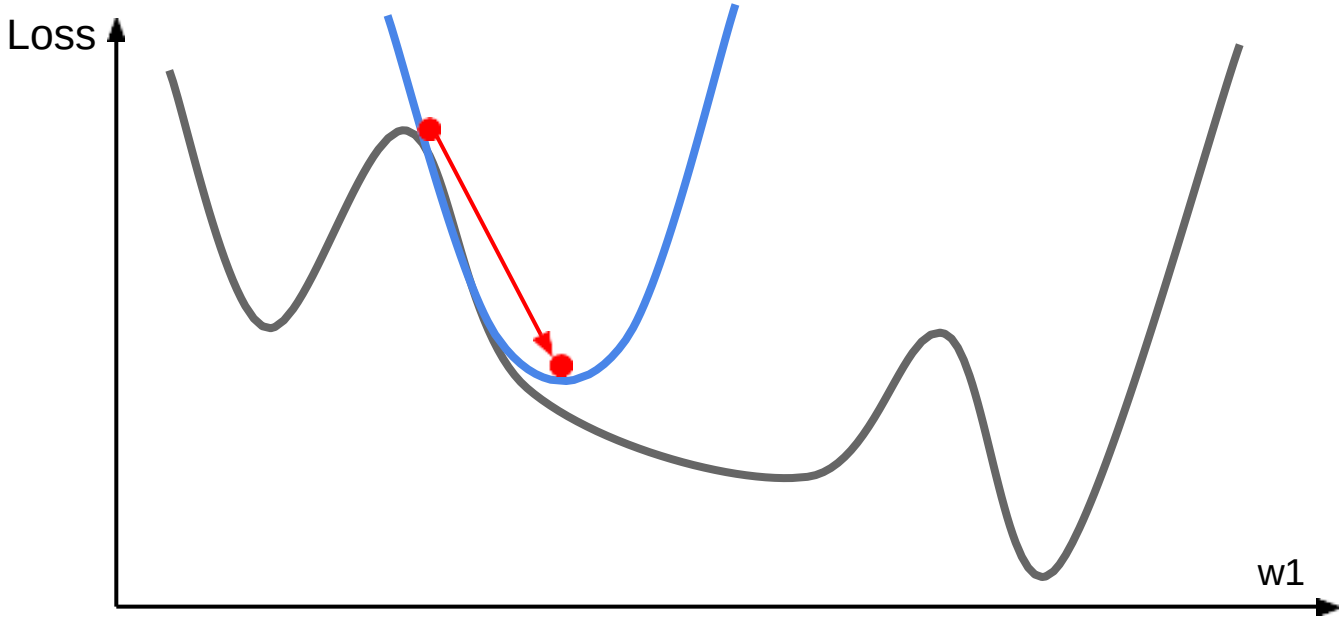
一阶优化

(1) 使用梯度形式的线性逼近
(2) 步进来减小逼近误差



二阶优化

(1) 使用**梯度**和**海塞矩阵**来进行二次逼近
(2) 步进到近似值的最小值



二阶优化

二阶泰勒展开:

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

通过求解临界点, 我们得到了牛顿参数更新:

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

问题:为什么这种方法不适于深度学习?

二阶优化

二阶泰勒展开:

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

通过求解临界点, 我们得到了牛顿参数更新:

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

海塞矩阵有O(N^2)的元素,
反转它的复杂度为O(N^3)
N为千万或亿的量级

问题:为什么这种方法不适于深度学习?

二阶优化

$$\theta^* = \theta_0 - \boldsymbol{H}^{-1} \nabla_{\theta} J(\theta_0)$$

- Quasi-Newton方法(**BGFS**最受欢迎):
  不直接求逆海塞矩阵*(复杂度O(n^3))*, 而是用正定矩阵近似逆海塞矩阵 *(复杂度O(n^2)).*

- **L-BFGS**(有限存储容量BFGS):
  不形成/存储全部的逆海塞矩阵

L-BFGS

- **通常在全批处理，确定性模式下工作得很好**
  即：如果你有一个单一的确定性f(x)那么L-BFGS可能会很好地工作

- **不能很好地迁移到迷你批处理。**
  结果较差。将二阶方法应用于大规模，随机环境是一个活跃的研究领域。

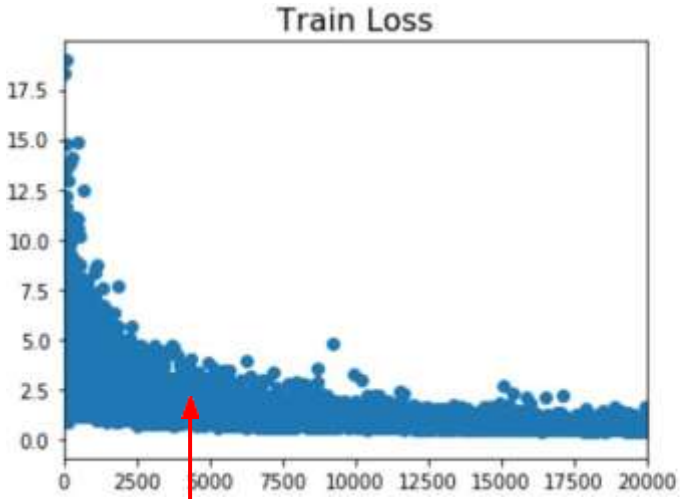
Le et al, “On optimization methods for deep learning, ICML 2011”
Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017

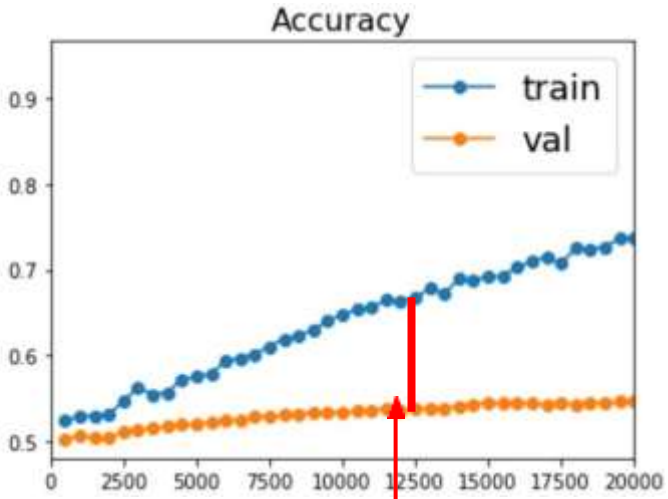
实际中:

- **Adam** 在很多情况下是很好的默认选择;
  即使学习速率恒定, 它也经常能正常工作。
- **SGD+Momentum** 可以超过Adam但是需要对学习率和学习率调整(learning rate schedule)进行更多调整。
  - 尝试使用余弦学习率计划, 它的参数非常少！
- 如果你可以在全批上做更新那么尝试一下
  L-BFGS (不要忘记关闭所有的噪音源)

改进测试误差

测试误差之外

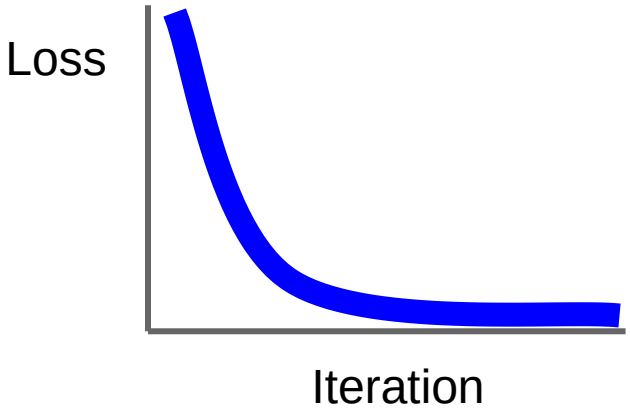


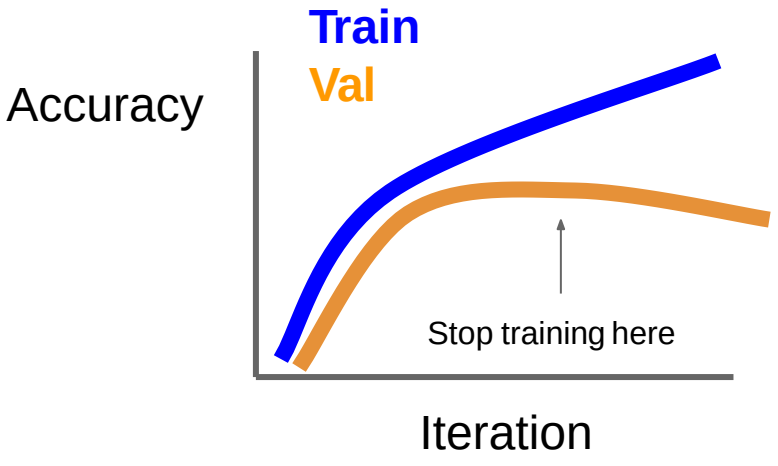


更好的优化算法有助于减少训练损失

但我们真正关心的是新数据的误差
——如何缩小和训练损失的差距?

早停法: 总是使用它





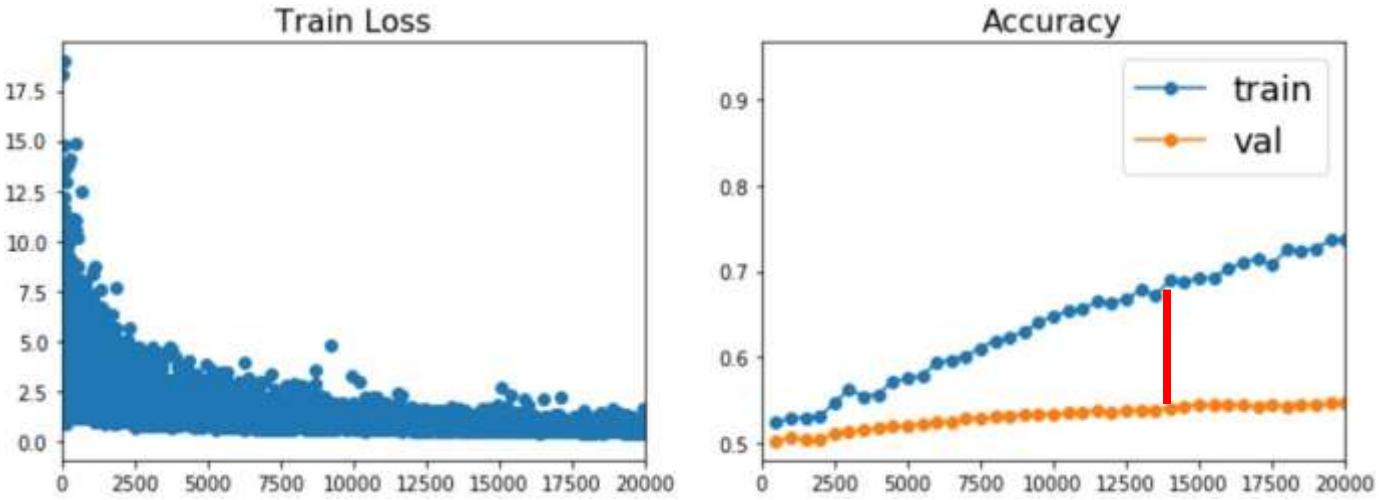
当验证集的精确度降低或训练较长时间时，停止对模型的训练，但始终跟踪在验证集上工作得最好的模型快照。

模型集成

1. 训练多个独立模型
2. 在测试时平均它们的结果
   (对预测的概率分布取平均值, 然后选择argmax)

享受2%的额外性能

如何提高单模型的性能?



正则化

正则化: 在损失函数中加入正则项

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

常用的:

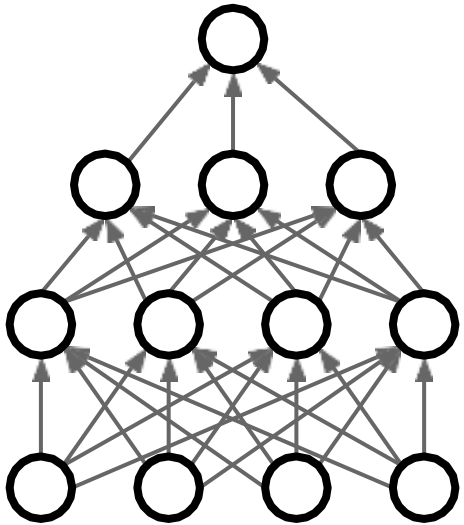
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (权重衰减)

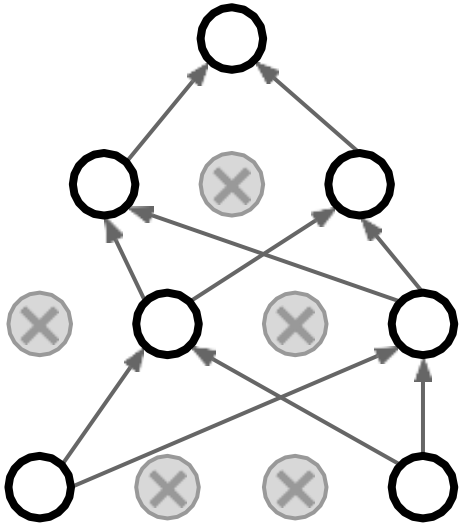
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

正则化: Dropout

在每一次正向传递中, 随机设置一些神经元为零
掉落概率是一个超参数, 常用0.5



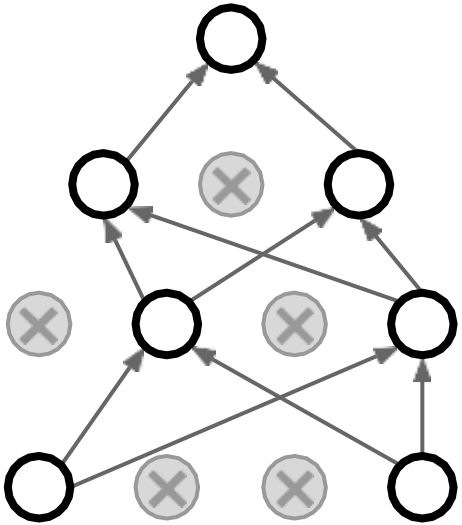


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

正则化: Dropout

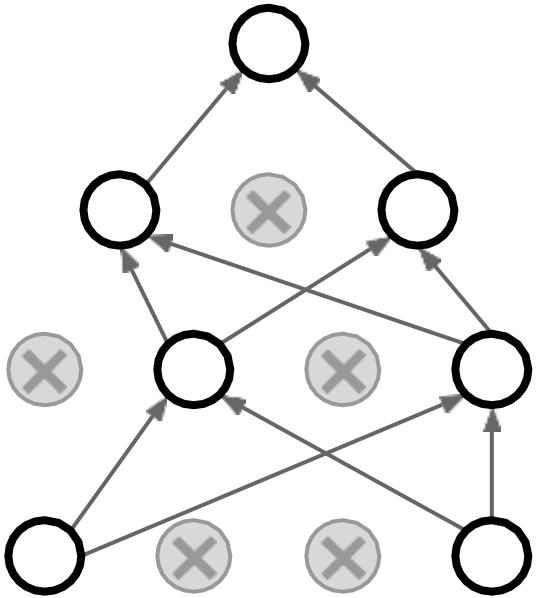
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

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

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

使用dropout的3层网络前向传播示例



正则化: Dropout

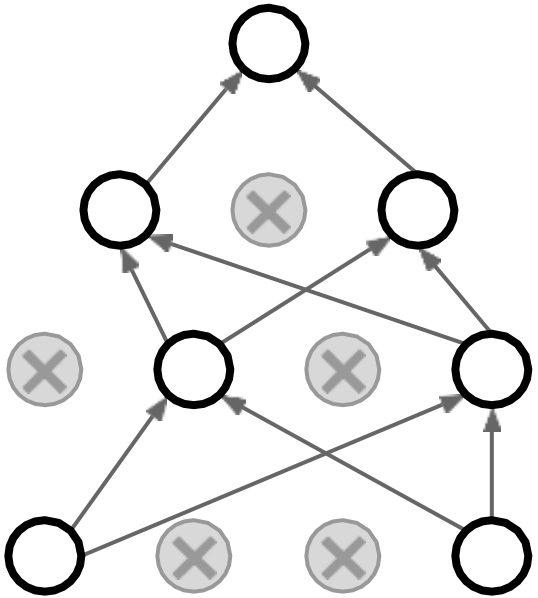
这为什么是个好主意呢?



强制网络具有冗余表示
阻止特征的协同适应

正则化：Dropout

这为什么是个好主意呢?



另一种解释:

Dropout训练了大量的模型集合(共享参数)。

每个二进制掩膜是一个模型。

一个有4096个单元的全连接层有
$2^{4096} \sim 10^{1233}$个可能的掩膜!

宇宙中只有 $\sim 10^{82}$ 个原子...

Dropout：测试时

Dropout使输出随机化!

输出
(标签)

输入
(图像)

$$y = f_W(x, z)$$

随机掩膜

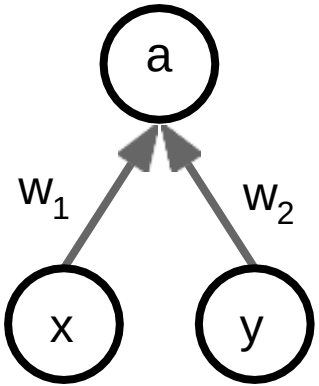
想要“平均”测试时的随机性

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

但是这个积分看起来很难…

Dropout：测试时

想要近似右面的积分 $$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

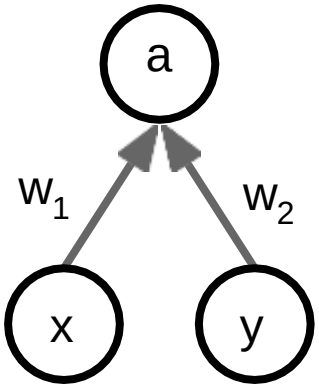


考虑单个神经元。

在测试时我们有: $$E\big[a\big] = w_1 x + w_2 y$$

Dropout：测试时

想要近似右面的积分 $$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



考虑单个神经元。

在测试时我们有: $$E\big[a\big] = w_1 x + w_2 y$$

在训练时我们有:

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

在测试时，**乘以**dropout 概率

Dropout：测试时

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

在测试时，所有的神经元总是活跃的

=> 我们必须放缩激活输出，这样才能使每个神经元

测试时的输出= 训练时的预期输出

Dropout 总结

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

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

def train_step(X):
  """ X contains the data """

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

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

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

在训练时丢弃

在测试时放缩

更常用的形式: “反向dropout”

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

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

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

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

测试时保持不变!

正则化: 通用模式

训练: 添加一些随机性

$$y = f_W(x, z)$$

测试: 平均来消除随机性(一些时候是近似)

$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z) dz$$

正则化: 通用模式

训练: 添加一些随机性

$$y = f_W(x, z)$$

测试: 平均来消除随机性(一些时候是近似)

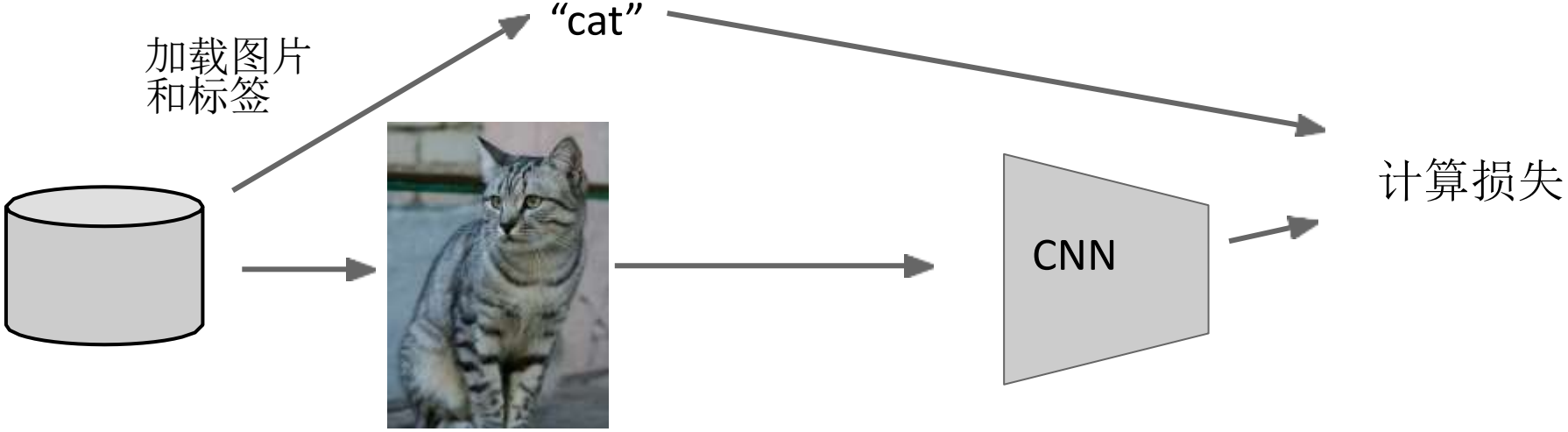
$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z) dz$$

例子: 批归一化

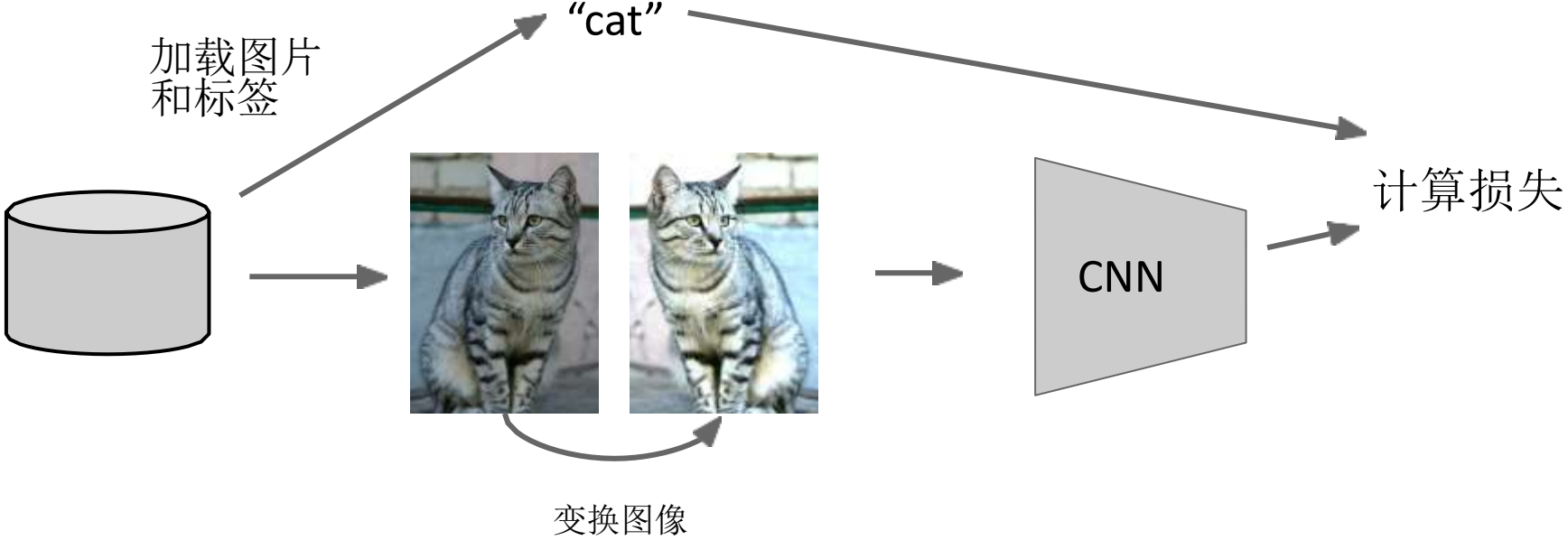
训练: 使用随机小批的统计数据进行归一化

测试: 使用固定的统计数据归一化

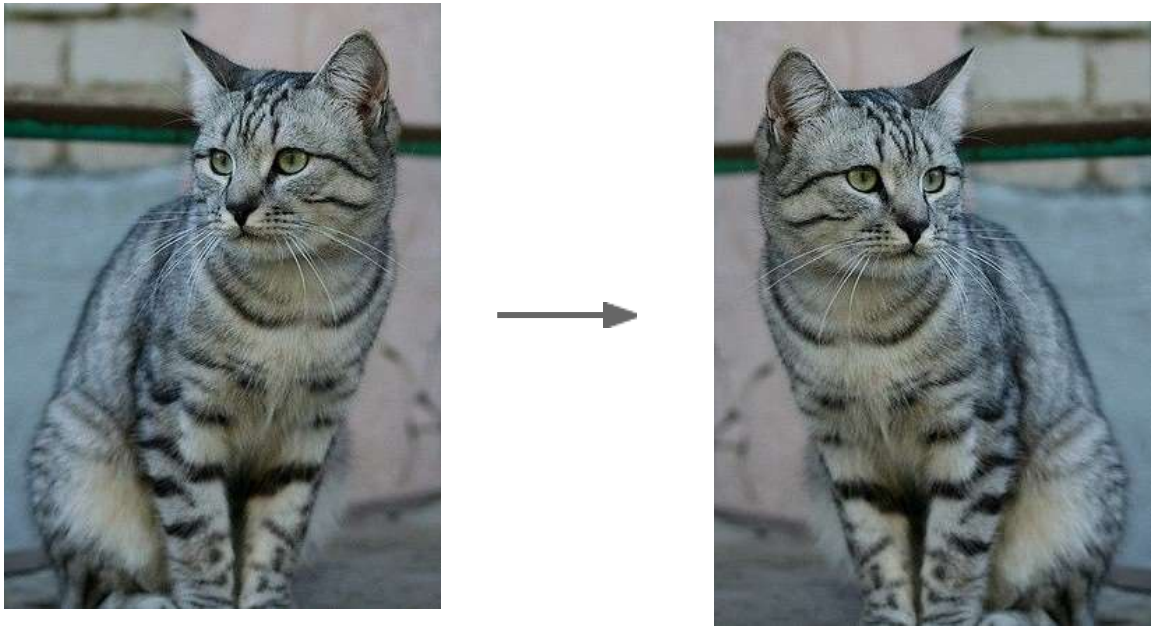
正则化: 数据增广



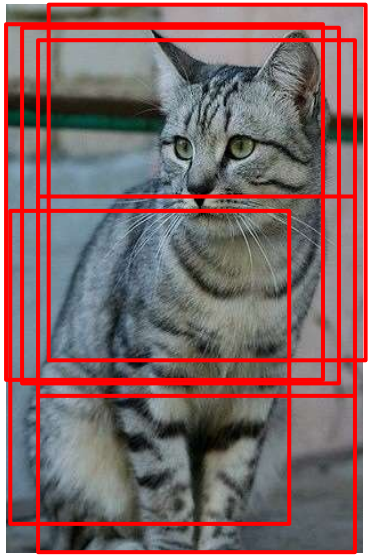
正则化: 数据增广



数据增广:**水平翻转**



数据增广:随机裁剪和缩放

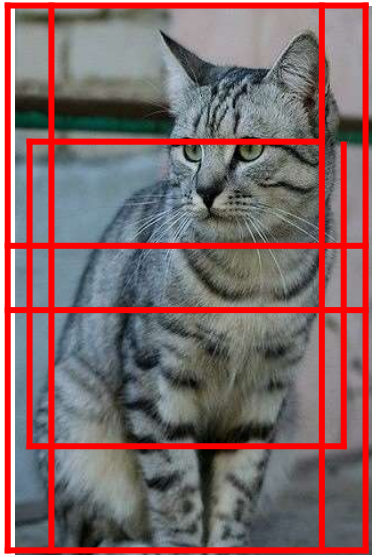


训练: 从随机裁剪 / 缩放中取样

ResNet:

1. 从[256, 480]中随机选取L
2. 调整训练图像大小, 短边= L
3. 随机选取224 x 224 的切片

数据增广:随机裁剪和缩放



训练:随机取样裁剪 / 缩放

ResNet:

1. 从[256, 480]中随机选取L
2. 调整训练图像, 短边= L
3. 随机选取224 x 224 的切片

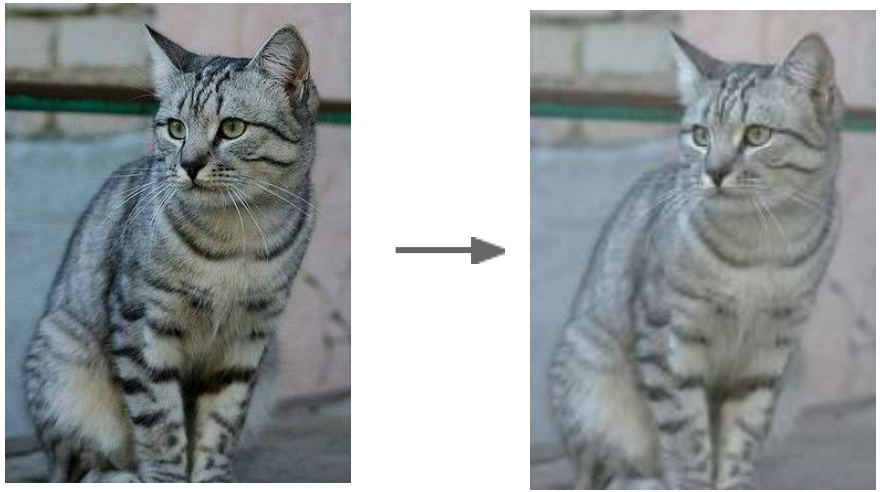
测试: 平均一套固定的裁剪切片

ResNet:

1. 按5个尺寸调整图像大小: {224, 256, 384, 480, 640}
2. 对于每个尺寸大小, 使用10个224 x 224切片: 4 corners + center, + flips

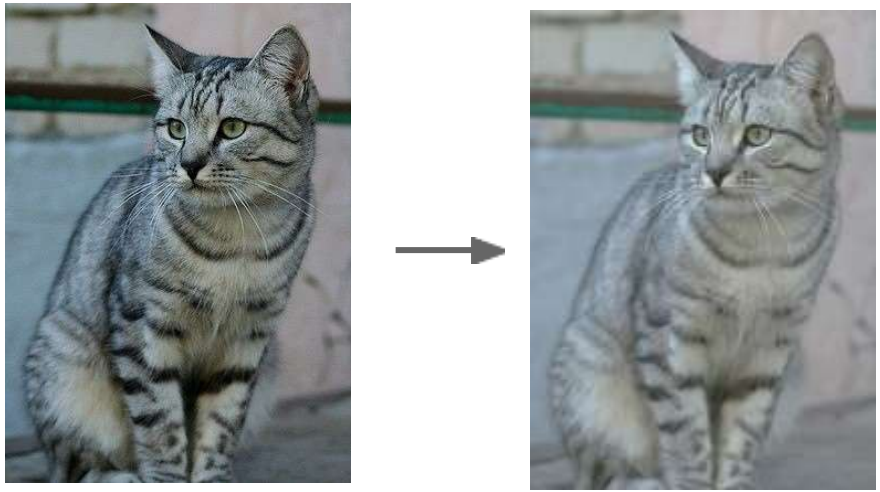
数据增广:颜色抖动

简单形式: 随机化对比度和亮度



数据增广:颜色抖动

简单形式: 随机化对比度和亮度



更加复杂的形式:

1. 对训练集中所有的[R, G, B]像素进行主成分分析
2. 沿主成分方向取样“色偏”
3. 为训练图像的所有像素添加偏移量

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

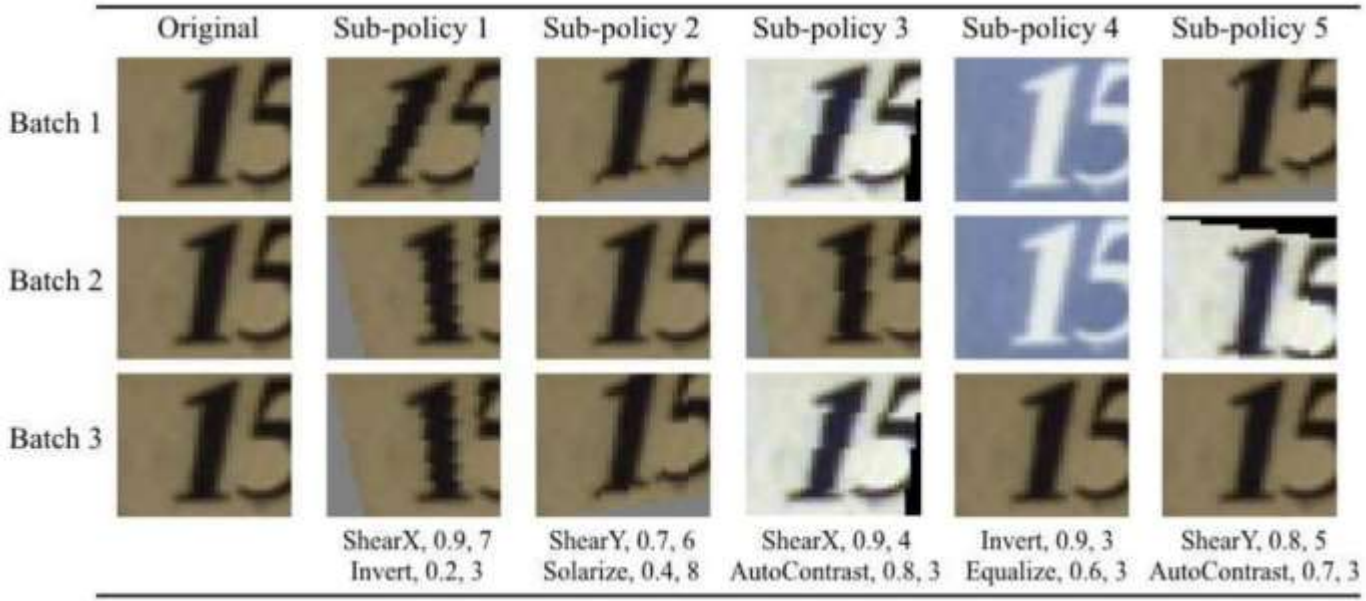
数据增广

对你的问题要有自己的创意!

随机混合/结合:

- 转化
- 旋转
- 拉伸
- 剪切
- 透镜扭曲, …(疯狂操作)

自动数据增广



Cubuk et al., “AutoAugment: Learning Augmentation Strategies from Data”, CVPR 2019

正则化: 通用模式

训练: 加入随机噪声
测试: 将噪声边缘化

例子:

Dropout
批归一化
数据增广

正则化: DropConnect

训练: 丢掉神经元之间的连接(将权重设置为0)

测试: 使用所有的连接

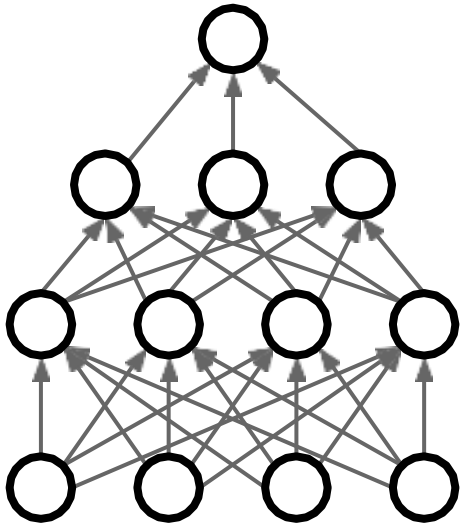
例子:

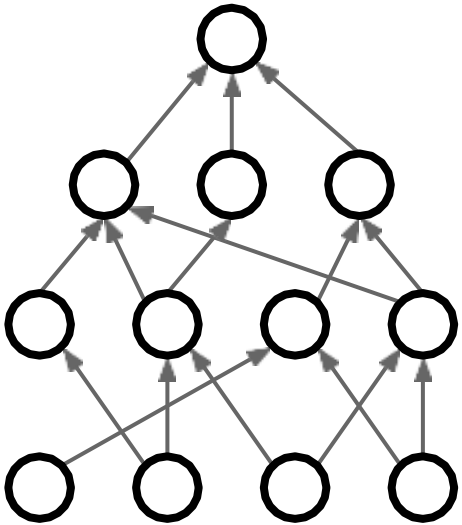
Dropout

批归一化

数据增广

DropConnect





Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

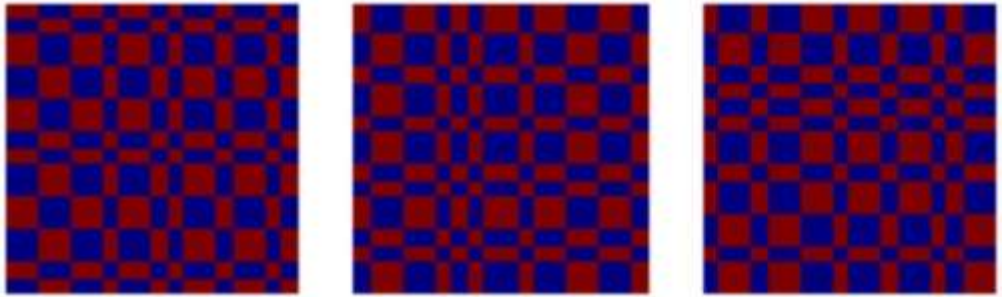
正则化: 分数阶池化

训练: 使用随机的池化区域
测试: 对几个地区的预测进行平均

例子:

Dropout
批归一化
数据增广

DropConnect
分数阶最大值池化



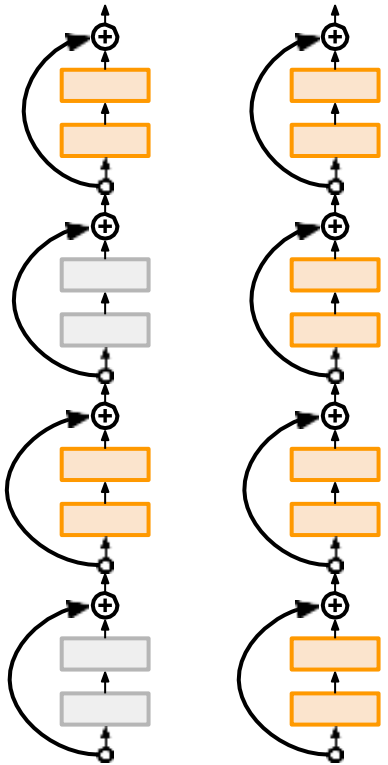
Graham, “Fractional Max Pooling”, arXiv 2014

正则化: 随机深度

训练: 跳过网络中的一些层
测试: 使用所有的层

例子:

Dropout
批归一化
数据增广
DropConnect
分数阶最大值池化
随机深度

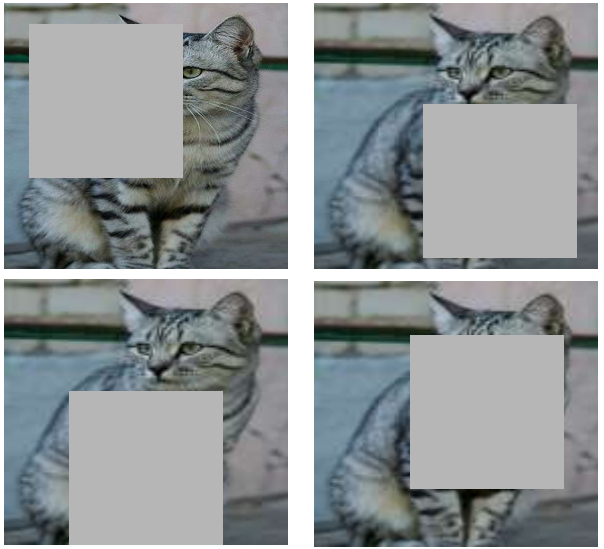


Huang et al, “Deep Networks with Stochastic Depth”, ECCV 2016

正则化: Cutout

训练:将随机图像区域设为零
测试:使用全部图像

例子:
Dropout
批归一化
数据增广
DropConnect
分数阶最大值池化
随机深度
Cutout / 随机裁剪



对于像CIFAR一样的小数据集效果很好,
像ImageNet一样的大数据集很少使用

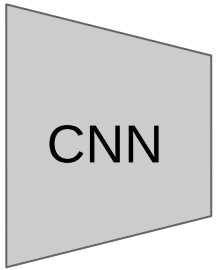
DeVries and Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout”, arXiv 2017

正则化: 混合

训练: 对随机混合的图像进行训练
测试: 使用原始图像

例子:
Dropout
批归一化
数据增广
分数阶最大值池化
随机深度
Cutout / 随机裁剪
混合

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog



Target label:
cat: 0.4
dog: 0.6

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

正则化 - 实际中

训练: 加入随机噪声
测试: 将噪声边缘化

例子:

Dropout
批归一化
数据增广
DropConnect
分数阶最大值池化
随机深度
Cutout / 随机裁剪
混合

- 对于大的全连接层考虑使用dropout
- 批归一化和数据增广几乎总会是很好的方法
- 尝试cutout和混合，尤其对于小的分类数据集来说

选择超参数

(没有大量的gpu时)

选择超参数

步骤1: 检查初始的损失

关闭权重衰减, 初始化时检查损失的合理性
例子: log(C) 对于有C 类的softmax

选择超参数

步骤1:检查初始损失
步骤2:过拟合小样本

试着在一个小样本的训练数据上达到100%的训练准确度(大约5-10 迷你批);调整网络结构, 学习率, 权重初始化

损失没有减小? 学习率太低, 初始化不好
损失为Inf或者NaN? 学习率太高, 初始化不好

选择超参数

步骤1:检查初始损失
步骤2:过拟合小样本
步骤3:找到使损失减小的学习率

使用上一步中的网络结构, 使用所有的训练数据, 开启小权重衰减, 找出一个学习率, 使损失在约100次迭代内显著下降

可以尝试的好的学习率:1e-1, 1e-2, 1e-3, 1e-4

选择超参数

步骤1:检查初始损失
步骤2:过拟合小样本
步骤3:找到使损失减小的学习率
步骤4:粗糙网格, 训练大约1-5个周期

选择一些在步骤3中有效的学习率和权重衰减的值, 训练一些模型大约1-5个周期。

可以尝试的好的权重衰减: 1e-4, 1e-5, 0

选择超参数

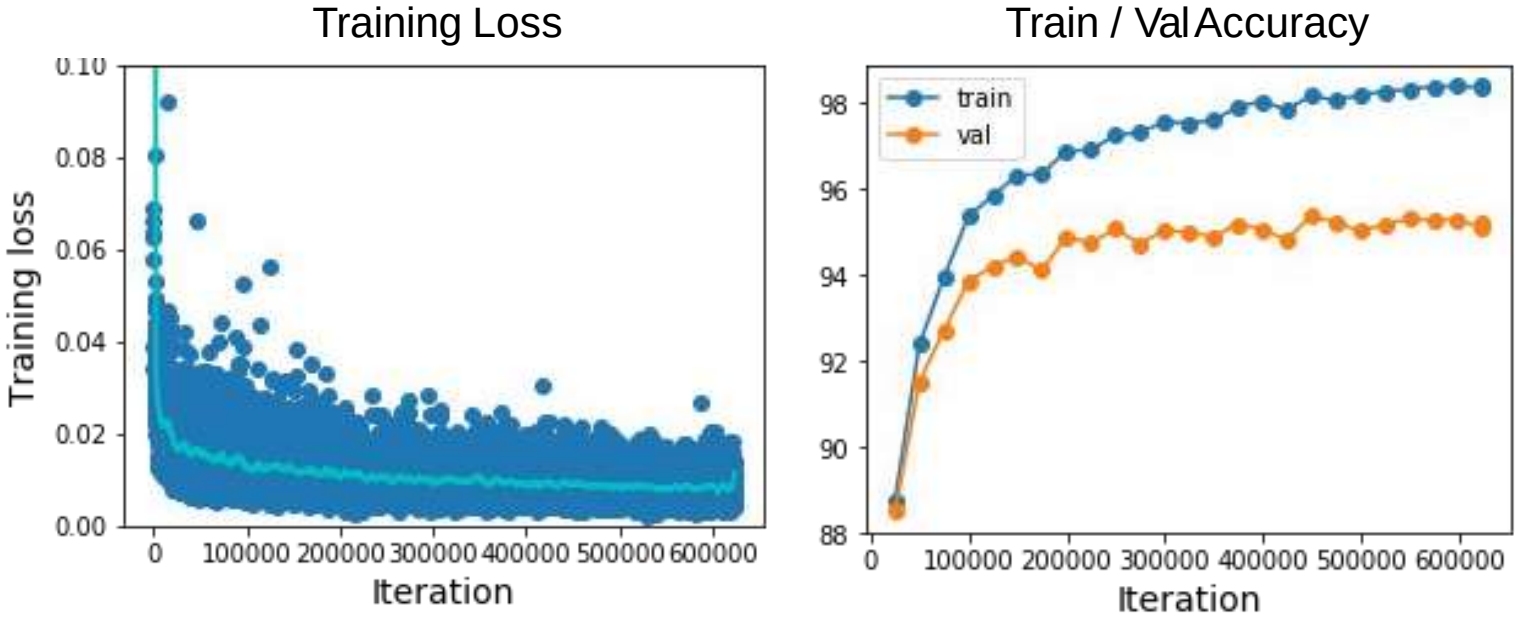
步骤1:检查初始损失
步骤2:过拟合小样本
步骤3:找到使损失减小的学习率
步骤4:粗糙网格, 训练大约1-5个周期
步骤5:精细网格, 训练更长周期

从步骤4中选择最好的模型, 在没有学习率衰减的情况下对它们进行更长时间的训练(大约10-20个周期)

选择超参数

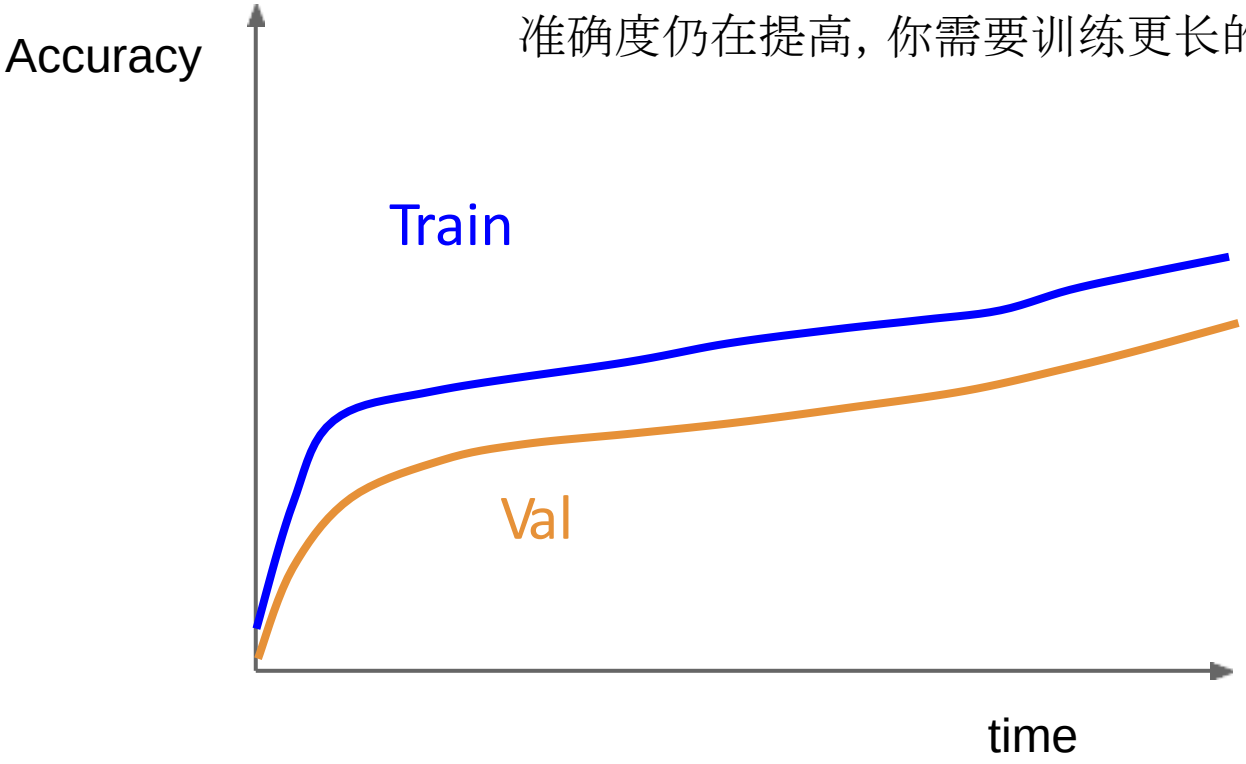
步骤1:检查初始损失
步骤2:过拟合小样本
步骤3:找到使损失减小的学习率
步骤4:粗糙网格, 训练大约1-5个周期
步骤5:精细网格, 训练更长周期
步骤6:观察损失曲线

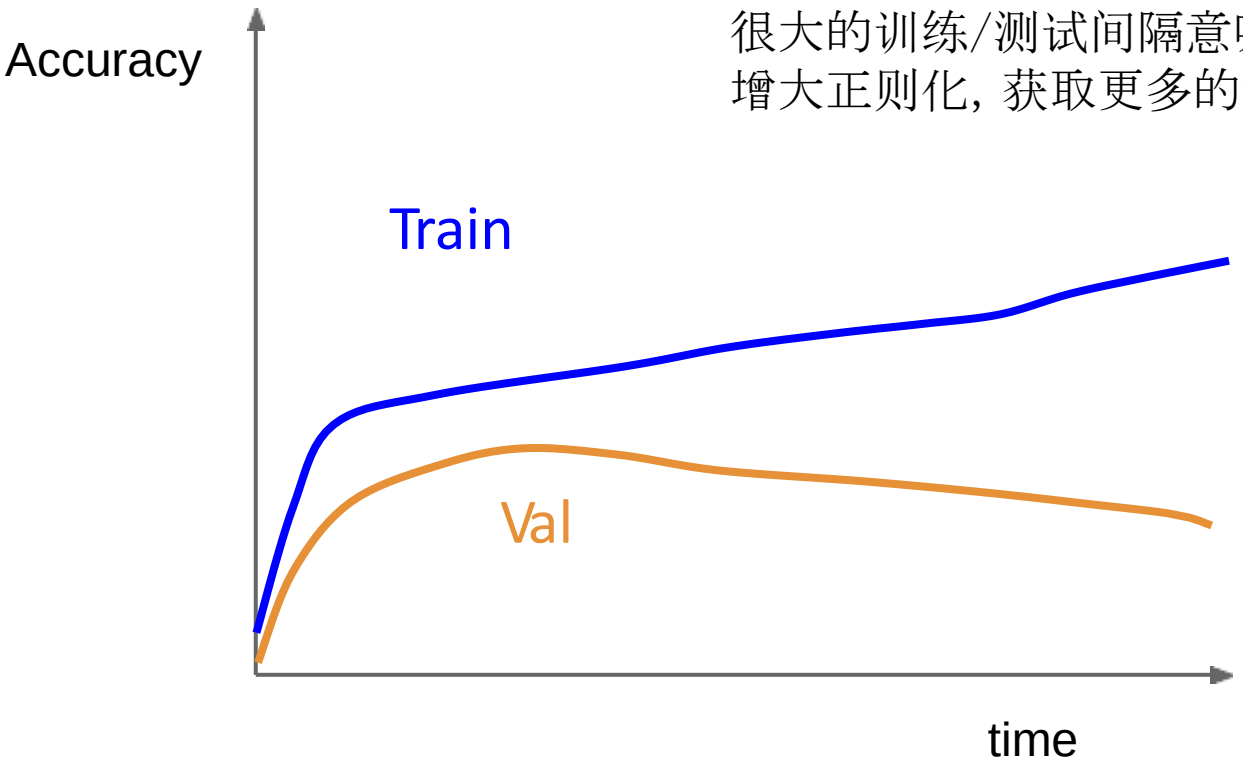
观察损失曲线!



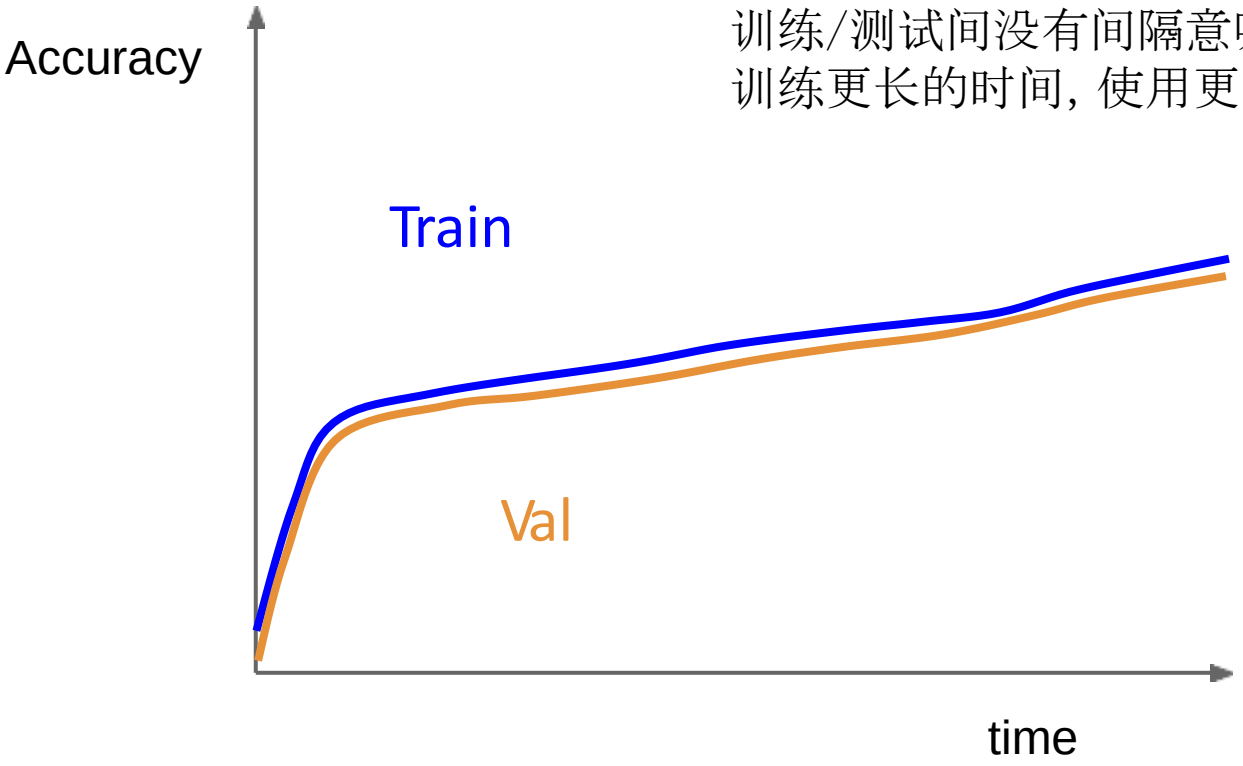
损失可能是有噪声的, 使用散点图,
也可绘制移动平均值曲线, 以更好地观察趋势

准确度仍在提高，你需要训练更长的时间





很大的训练/测试间隔意味着过拟合!
增大正则化，获取更多的训练数据



训练/测试间没有间隔意味着欠拟合:
训练更长的时间, 使用更大的模型

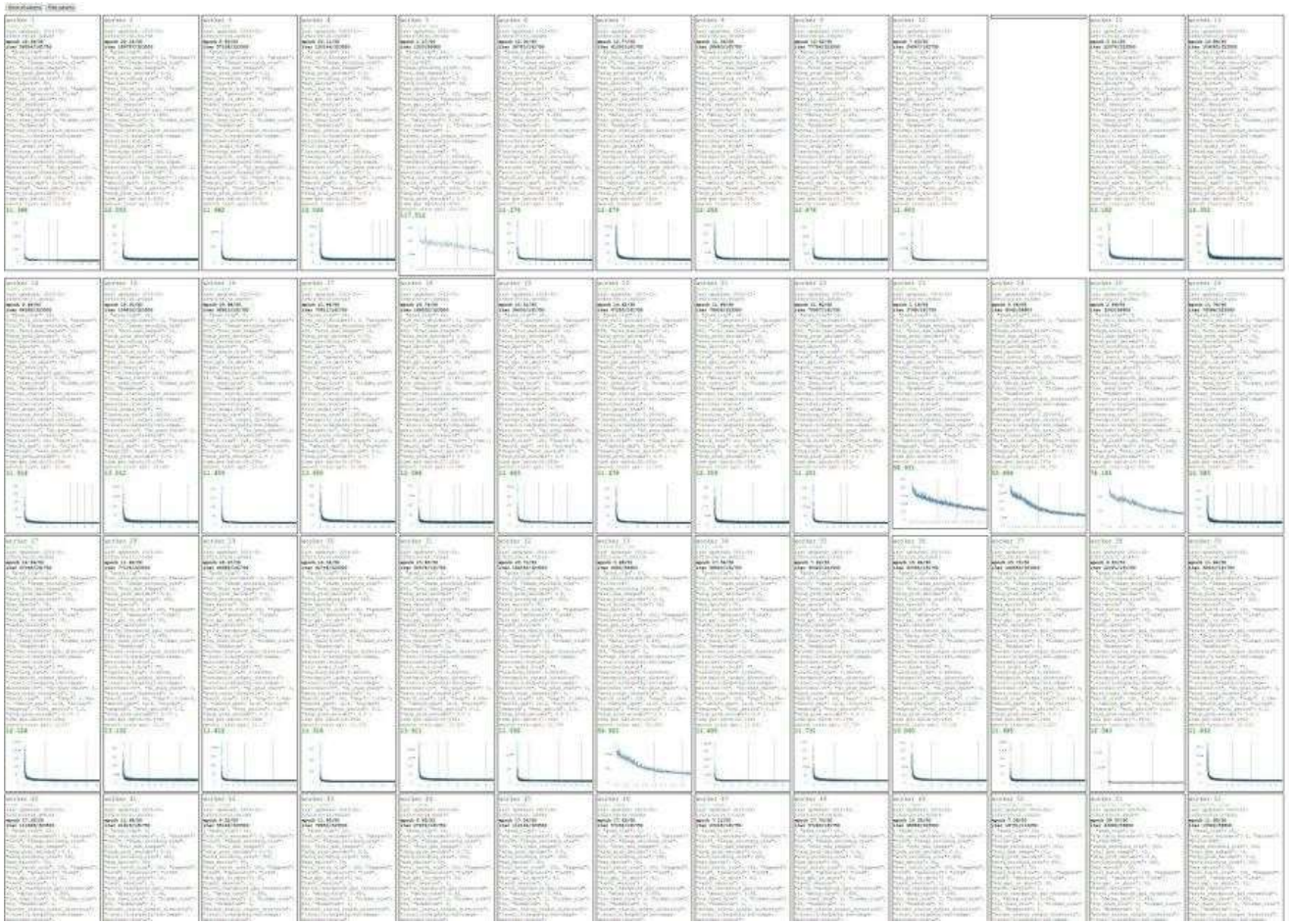
选择超参数

步骤1:检查初始损失
步骤2:过拟合小样本
步骤3:找到使损失减小的学习率
步骤4:粗网格，训练大约1-5个周期
步骤5:精细网格，训练更长周期
步骤6:观察损失曲线
步骤7:回到步骤5

可以调整的超参数:

- 网络结构
- 学习率, 衰减计划, 更新类型
- 正则化 (L2/Dropout strength)

交叉验证
“指挥中心”



随机搜索vs网格搜索

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

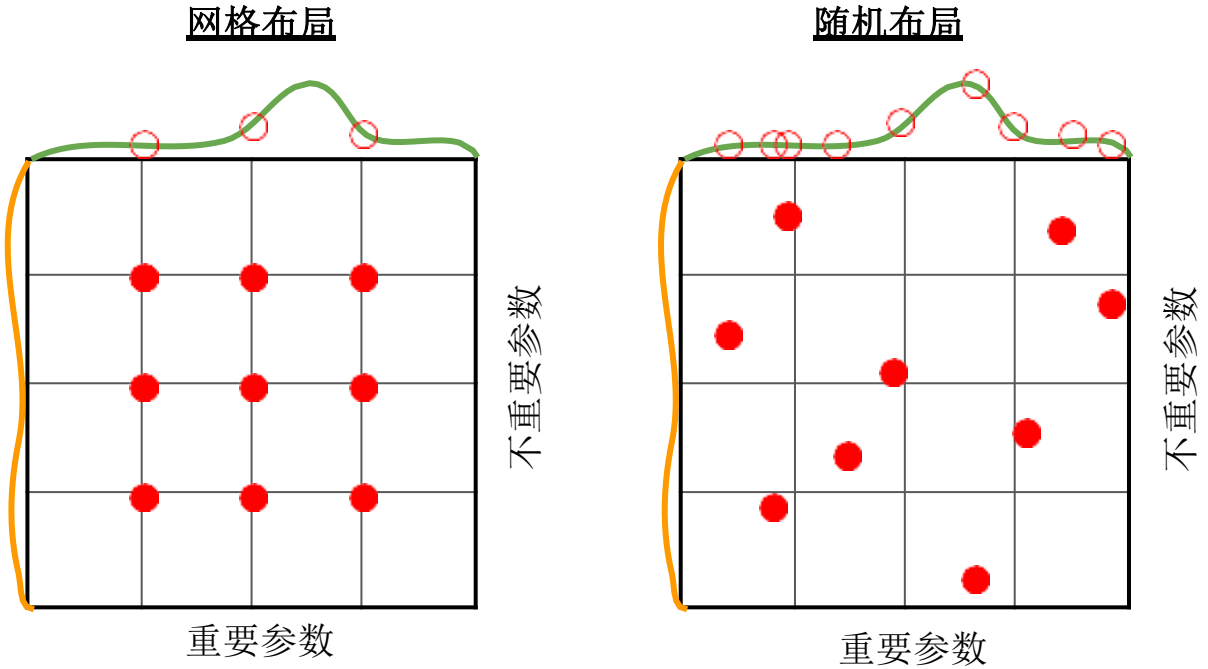


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

总结

- 改进训练误差:
  - 优化器
  - 学习率调整
- 改进测试误差:
  - 正则化
  - 选择超参数

翻译：张宇航