首都师范大学

为学为师 求实求新

深度学习应用与工程实践
7. 训练-2
7. Training-Part 2

李冰

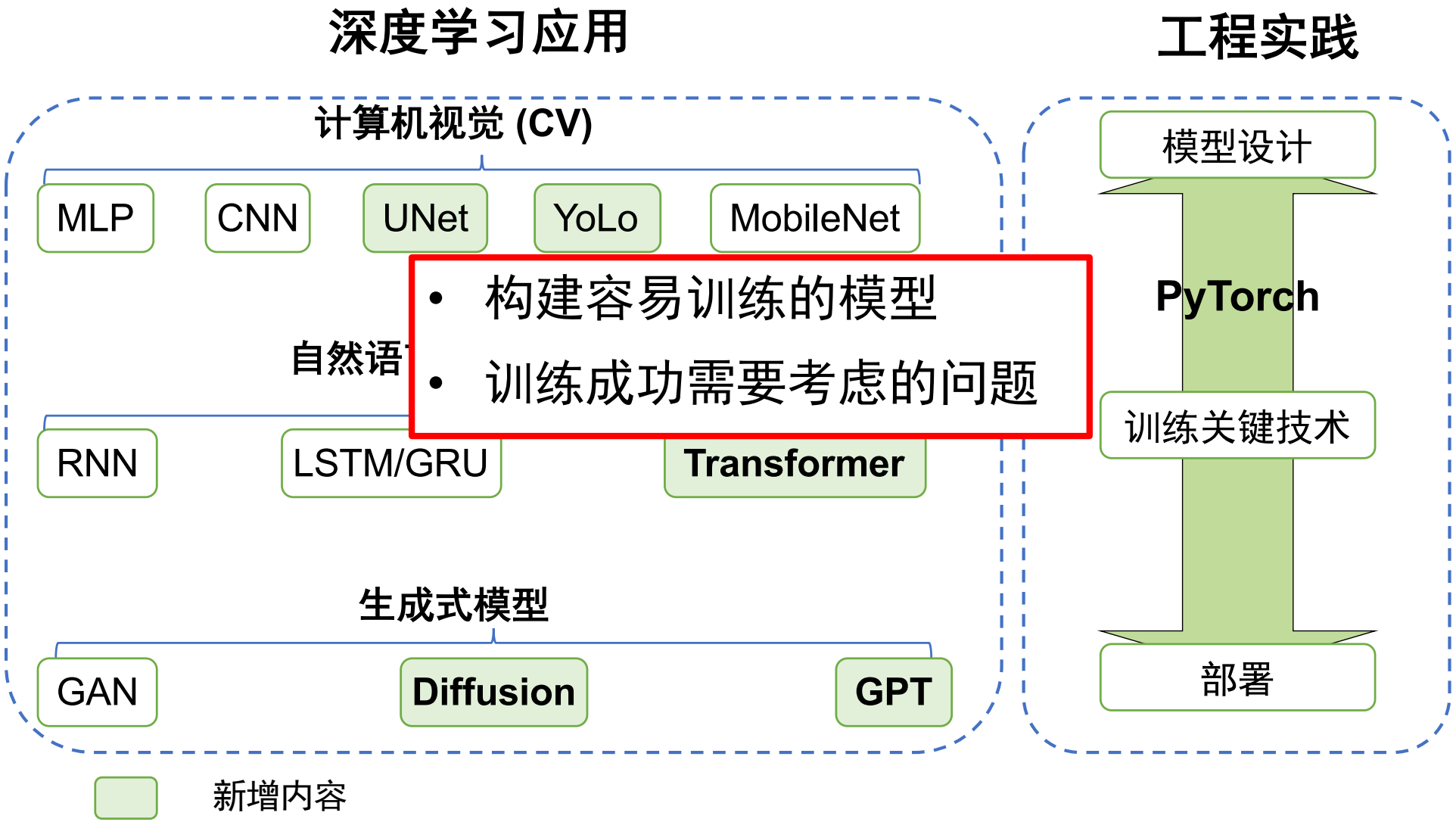
Bing Li

Tenure-track Associate Professor

Academy of Multidisciplinary Studies

Capital Normal University

这节课


深度学习应用
工程实践
计算机视觉 (CV)
MLP
CNN
UNet
YoLo
MobileNet
自然语
RNN
LSTM/GRU
Transformer
生成式模型
GAN
Diffusion
GPT
• 构建容易训练的模型
• 训练成功需要考虑的问题
模型设计
PyTorch
训练关键技术
部署
新增内容

优化策略 – Cont'd

- 激活函数
- 神经网络架构
- 损失函数 & 优化
- 权重初始化
- 正则化
- 数据预处理
- 超参数优化

优化策略 – Cont'd

- 正则化
  - 范数
  - Dropout
  - 批归一化
  - Label Smoothing
- 数据预处理
- 超参数优化

正则化 -- 过拟合

- 神经网络模型在训练集上表现很好，但对新数据预测时效果不好。
  - **比较好的作法是将数据集划分为三个部分——训练集、开发集（也称为交叉验证集）和测试集。**
  - 按照60/20/20的比例拆分
  - 按照98/1/1的比例划分

如果模型在训练集上表现良好，但是在验证集上验证错误率时，错误率会显著增加，这可能意味着模型存在**过拟合**。

最直接的方法：获取更多数据

防止过拟合

- L-norm 正则化
- Dropout
- Drop connect
- 批标准化Batch normalization
- Label smoothing

L-norm 正则化

L-norm 正则化用来解决**过拟合**的问题.

$$L(W) = \underbrace{CE(X, Y; W)}_{\text{交叉熵误差}} + \underbrace{\lambda R(W)}_{\text{正则化误差}}$$

- **L1 (Lasso) 正则化**

$$R(W) = \sum_{i,j} |W_{ij}| = \|W\|_1$$

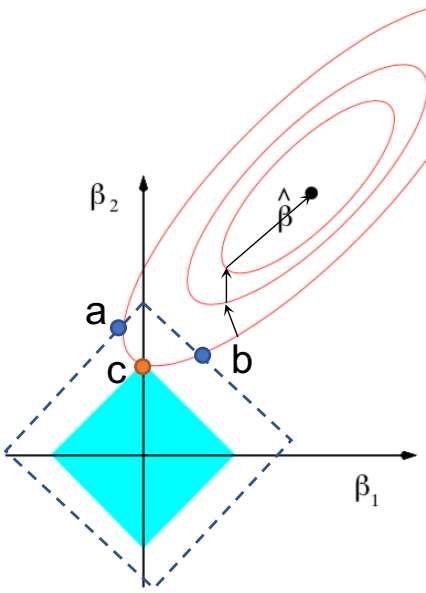
- **L2 (Ridge) 正则化 (权重衰减)**

$$R(W) = \sum_{i,j} W_{ij}^2 = \|W\|_2^2$$

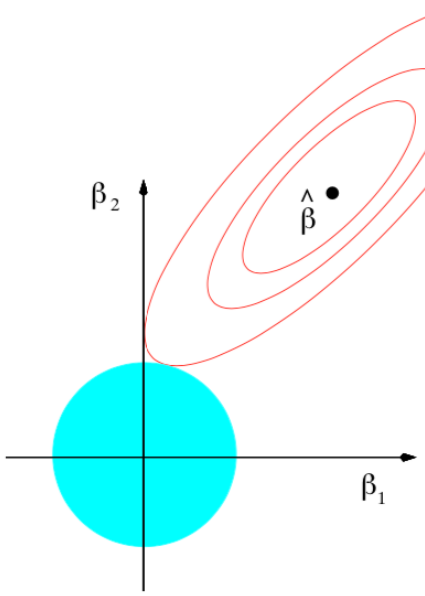
L-norm 正则化

L-norm的可视化

假设 $W = \{\beta_1, \beta_2\}$.



L1 (Lasso)
正则化



L2 (Ridge)
正则化

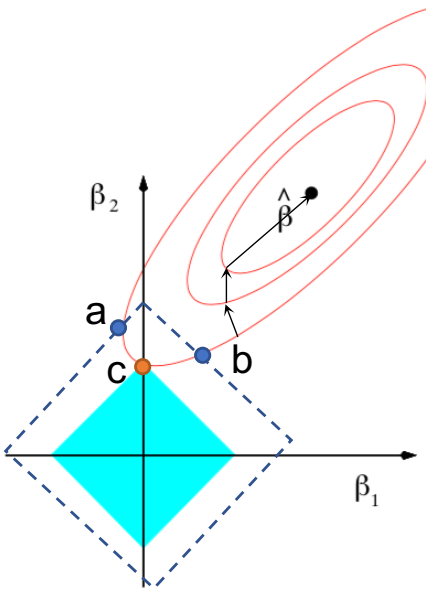
- 只有在目标函数方向上显著减小的参数会保存完好。
- 由于L1 （Lasso）正则化的损失函数表面在零值有顶点。
- 因此在训练期间应用L1 （Lasso）正则化更可能导致权重稀疏。
- 而L2（Ridge）正则化倾向于产生较小的权重。

具体参考花书7.1

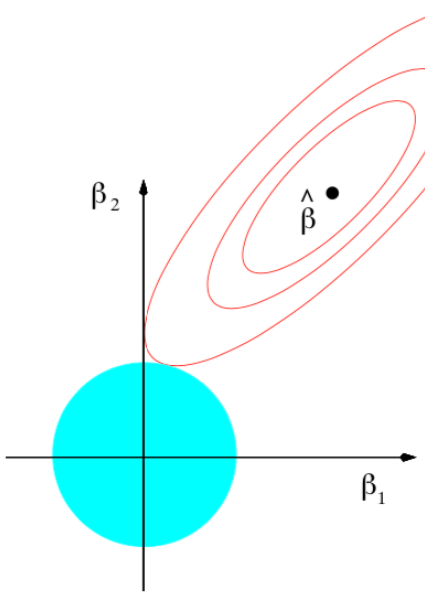
L-norm 正则化

L-norm的可视化

假设 $W = \{\beta_1, \beta_2\}$.



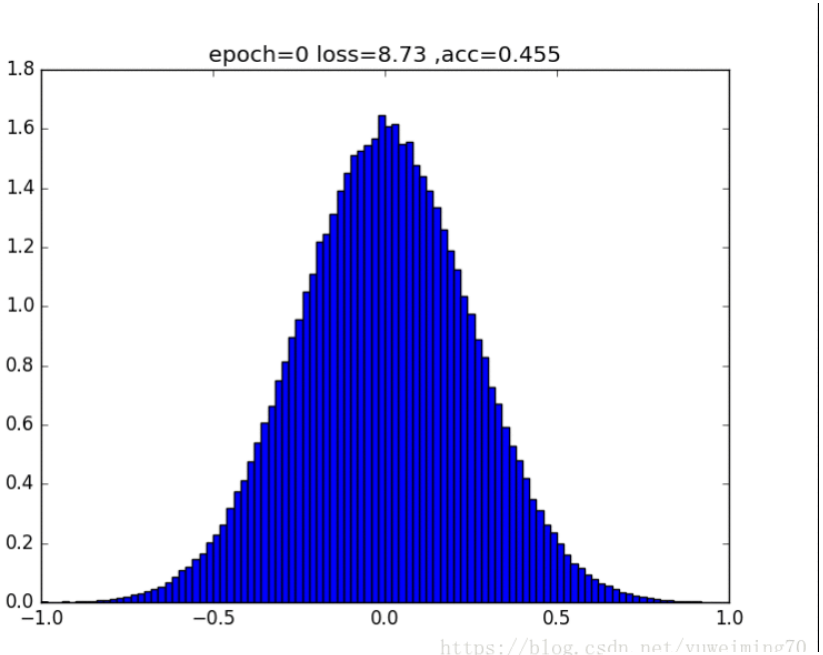
L1 (Lasso)
正则化

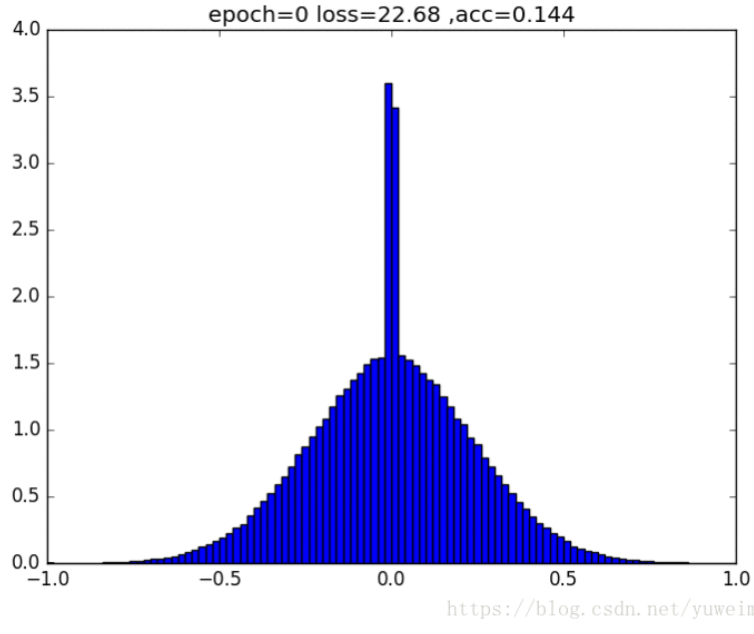


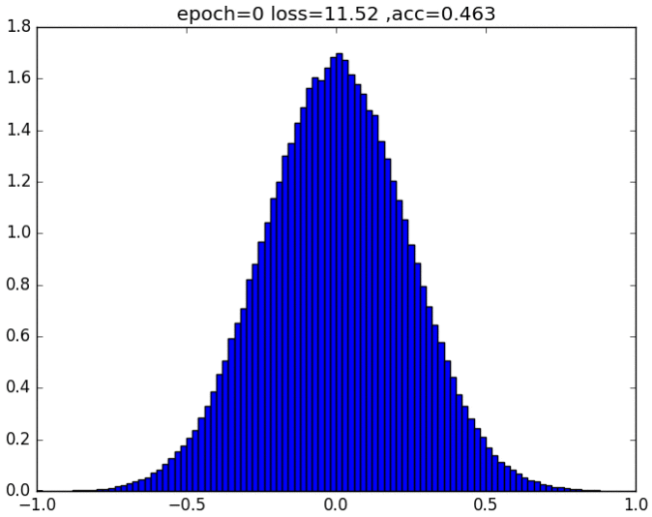
L2 (Ridge)
正则化

- 只有在目标函数方向上显著减小的参数会保存完好。
- 由于L1 （Lasso）正则化的损失函数表面在零值有顶点。
- 因此在训练期间应用L1 （Lasso）正则化更可能导致权重稀疏。
- 而L2（Ridge）正则化倾向于产生较小的权重。

具体参考花书7.1


epoch=0 loss=8.73 ,acc=0.455
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0


epoch=0 loss=22.68 ,acc=0.144
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0


epoch=0 loss=11.52 ,acc=0.463
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0

Pytorch实现

- torch.optim优化器实现L2正则化

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,
dampening=0, weight_decay=0)
```

- **参数：weight_decay**：L2 正则化中λ 的值。

- L1正则化

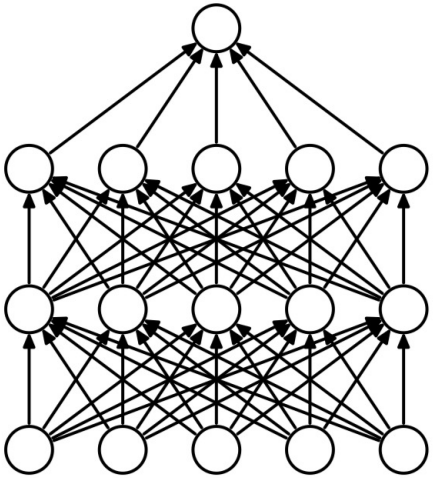
```
#L1Norm
class L1Norm(nn.Module):
    def __init__(self):
        super().__init__()
        self.regular_loss = nn.CrossEntropyLoss()
        self.penalty = nn.L1Loss()

    def forward(self, in_data, target,l1_lambda):
        regular_loss = self.regular_loss(in_data, target)
        l1_loss = self.penalty(in_data, target)
        output = regular_loss+l1_lambda*l1_loss
        return output
```

```
l1_lambda = 0.01
in_1 = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)
loss = L1Norm()
output = loss(in_1, target, l1_lambda)
output.backward()
```

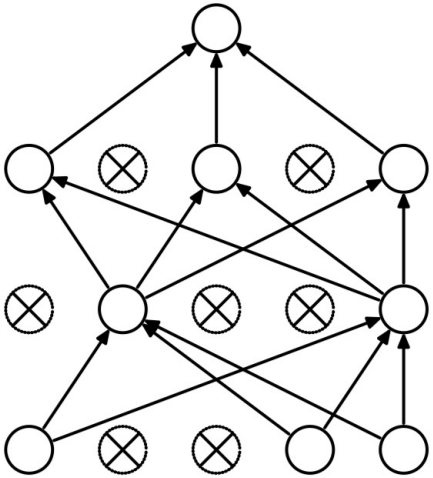
Dropout

随机地“删除”一些的隐层单元和它们的连接。

- 将一些单元的输出乘零，就能有效地删除这些单元



(a) Standard Neural Net



(b) After applying dropout.

优点:

+ 与L-norm正则化相比，配置更容易，泛化效果更好
+ 创建**子网络集合**以解决过拟合问题.

建议:

1.在实验开始时**分别**使用dropout和正则化，以确定它们各自的效果和作用。

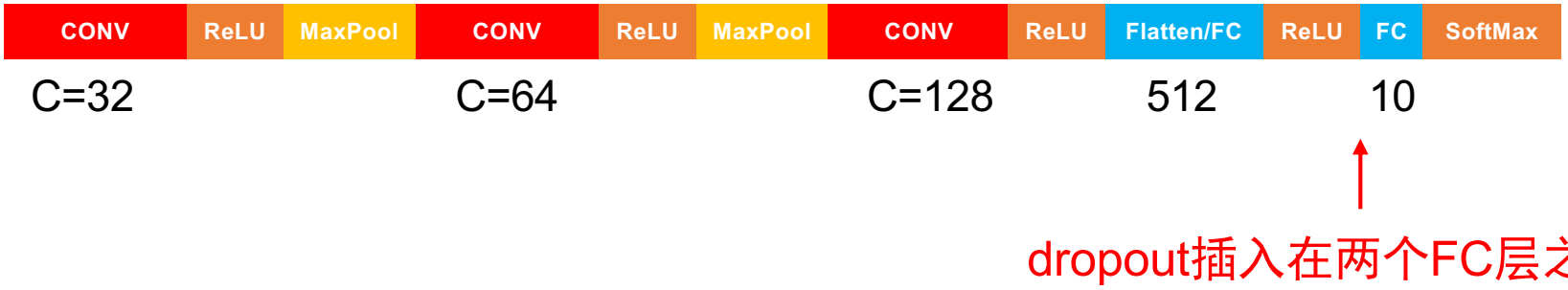
2.再进一步确定联合使用后的效果。

Dropout

- Dropout 概率p 是丢掉某个神经元和它的连接的概率.
- Dropout 在**训练**过程中使用。
  - 随机产生应用与输出单元的二值/实值掩码。网络中的单元乘以相应的掩码。
- **测试**阶段, 不采用dropout.
- 大部分情况下, 一层最多丢弃一半的神经元.

Dropout: 位置

- 一般情况下, dropout 只用在全连接层之间，p=0.5。
- 具体来说，在上一个FC层的ReLU激活之后和下一个FC层之前插入dropout.
  - 有些工作提到在卷积层使用dropout，但是p要取更小的值.
  - 不在最后一层使用dropout. 因为最后一层dropout会导致预测的logit信息丢失。



Dropout: 训练 vs. 测试

- 在训练里神经元以概率 p 执行dropout.

输出 输入 W: 权重.

$$Y = f(X, Z; W)$$

Z: Dropout 阶段用的随机掩码

- 测试阶段, 计算随机掩码的期望值以获得稳定的输出.

$$Y = f(X, Z; W) = E_Z[f(X, Z; W)] = \int_Z p(Z) f(X, Z; W)$$

由于每个神经元具有相同的dropout概率p，因此上述积分近似为：

$$Y = f(X, Z; W) = (1 - p) f(X, Z; W)$$

因此，需要在测试过程中通过对激活值输出进行 $(1 - p)$ 的缩放。

Dropout: 训练 vs. 测试

Dropout工作流程:

输入: 当前层的激活值 α; dropout 概率 p

训练:

Step 1: 以概率p 随机选择要dropout的激活值.

Step 2: dropout的激活值设为 0.

Step 3: 将激活值（许多激活值已变为0）传递到下一个DNN层。

Step 4: 执行后向传播 (许多激活值已变为0).

测试:

Step 1: 按（1-p）缩放激活值α

Step 2: 缩放后的激活值α 传给下一层.

在测试过程中进行缩放会带来额外的计算成本，从而导致测试速度降低。

翻转 dropout: 训练 vs. 测试

工作流程: 翻转 dropout

输入:当前层的激活值 α; dropout 概率 p

训练:

Step 1:以概率p 随机选择要dropout的激活值..

Step 2: dropout的激活值设为 0.

Step 3: 以1 / (1-p)缩放激活值.

Step 4:将激活值（许多激活值已变为0）传递到下一个DNN层。

Step 5:执行后向传播 (许多激活值已变为0).

测试:

Step 1缩放后的激活值α 传给下一层.

通过将比例缩放部分移至训练阶段，翻转dropout减少了测试阶段额外的计算量。

Pytorch实现

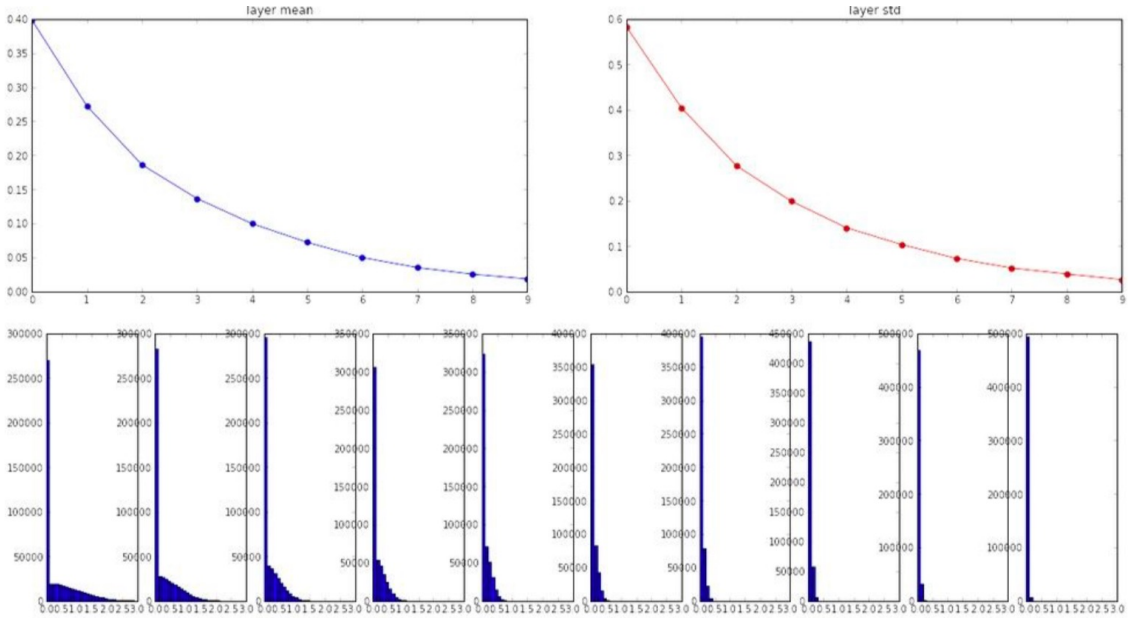
- torch.nn.Dropout(*p=0.5*,...)
- 输出乘以**1/(1-p)**因子
- 参数：p: 元素置为0的概率，默认为0.5

```
import torch.nn as nn

m = nn.Dropout(p=0.2)
input = torch.randn(20, 16)
output = m(input)
```

批归一化Batch normalization

- 训练DNN十分复杂，在训练过程中，随着先前各层的激活值的变化，各层输入的分布也会发生变化。
- 期望保持激活分布在各层之间的相似性。
- 理想情况下，我们希望各层输入是零均值和单位方差的高斯分布。

10层网络激活值的分布



各层输入是零均值和单位方差的高斯分布时，神经网络效果最好。

但是，在这个10层网络中，激活是没有被缩放的，并且具有完全不同的分布。 这使得DNN很难训练。

批标准化Batch normalization: 训练

目标：使激活在每一层中的均值和单位方差为零.

训练阶段, 我们累计每个神经元的移动平均值和方差；

通过训练得到BN层里的缩放比例和偏差参数

优点:

+ DNNs 训练可以用更大的学习率
+ 某种形式的正则化
+ 不在对权重初始化敏感.

缺点:

- 额外的计算.
- Dropout 会破坏BN产生的分布.

Input: values of x over a mini-batch.

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

$$\mu_{\mathrm{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathrm{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathrm{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathrm{B}}}{\sqrt{\sigma_{\mathrm{B}}^2 + \epsilon}}$$

$$y_i = \gamma\hat{x}_i + \beta$$

μ_{B}: 当前批次的平均值

σ_{B}^2: 当前批次的方差

γ: 可训练的scale参数

β: 可训练的偏差值.

ϵ: 很小的值避免零除. 小数据集设为1e-5 ，大数据集是1e-3

批归一化Batch normalization:推断

在训练过程中累积的移动平均值（μ）和移动方差（σ）会在测试和推断阶段使用。以下是在训练过程中移动平均值和移动方差的计算：

$$\mu^{t+1} = \rho\mu^{t} + (1-\rho)\mu_{\mathrm{B}}^{t+1}$$

$$\sigma^{t+1} = \rho\sigma^{t} + (1-\rho)\sigma_{\mathrm{B}}^{t+1}$$

- 对于批量128，ρ最好设置为0.9。
- 批标准化在训练和推断阶段具有不同的行为。 在PyTorch中使用时要注意。

批归一化Batch normalization:位置

- Batch normalization (BN) 层加在非线性激活层之前

CONV-RELU-CONV-RELU

CONV-BN-RELU-CONV-BN-RELU

- Dropout 加在非线性激活层之后.

CONV-BN-RELU-DROPOUT-CONV-BN-RELU-DROPOUT

经验上来看, batch normalization会取得2%左右的性能提升.

Pytorch实现

torch.nn.BatchNorm1d(*num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None*)

torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None)

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

```
import torch.nn as nn
nn.Linear(784, 48),
nn.BatchNorm1d(48) #48 对应输入的特征图的数量.
import torch.nn as nn
nn.Conv2d(3, 6, 3)
nn.BatchNorm2d(6)                    #输入特征图的通道数
```

Label smoothing

Label smoothing: 原始的标签

$$y_{label} = [0,1,0,0,0, \dots]$$

替换为

$$y_{label} = \left[\frac{\epsilon}{K-1}, 1-\epsilon, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1}, \dots\right]$$

K: 类别数
ϵ: 标签平滑因子.

One-hot 标签在训练过程中倾向做出极端的预测，导致过拟合.

通常, ϵ 设置为 0.1.

Label smoothing

我们有一个one-hot编码的分类目标

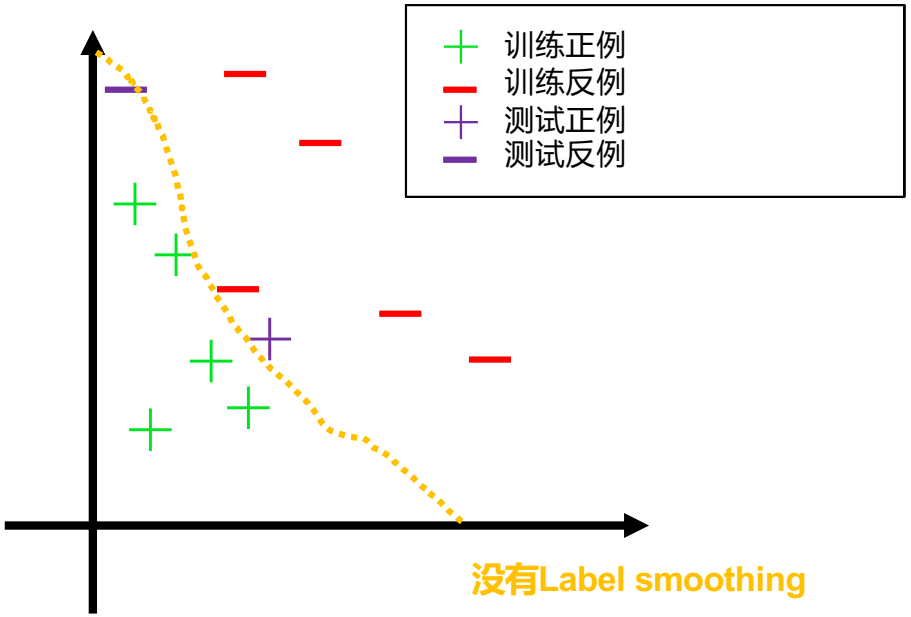
$$y_{label} = [0,1,0,0,0, \dots]$$

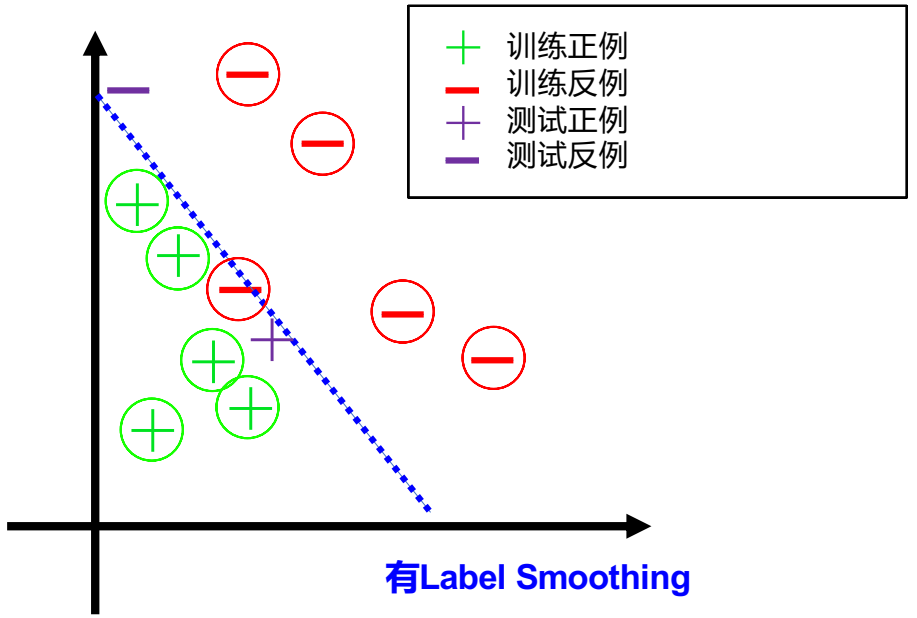
SoftMax 的预测结果如下：

$$y_{predict} = [0.02,0.9,0.04,0.001, \dots]$$

因为神经网络不会预测0/1，所以训练过程会使得权重越来越大来逼近正确的预测值，很容易就导致过拟合.

Label smoothing





没有 label smoothing, DNN会做出极端的预测，因此决策边界很难适应每个训练示例。 这会导致严重的过度拟合。 新的测试示例未正确分类。

有 label smoothing, 决策边界 稍微跨越类别. 但是，此决策边界对新的测试示例进行了正确分类。

Pytorch实现

```
def smooth_one_hot(true_labels: torch.Tensor, classes: int, smoothing=0.0):
    assert 0 <= smoothing < 1
    confidence = 1.0 - smoothing
    label_shape = torch.Size((true_labels.size(0), classes))
    with torch.no_grad():
        true_dist = torch.empty(size=label_shape, device=true_labels.device)
        true_dist.fill_(smoothing / (classes - 1)) true_dist.scatter_(1,
        true_labels.data.unsqueeze(1), confidence)
        return true_dist
```

https://github.com/pytorch/pytorch/issues/7455#issuecomment-513735962

优化策略 – Cont'd

- 正则化
  - 范数
  - Dropout
  - 批归一化
  - Label Smoothing
- 数据预处理
- 超参数优化

数据预处理

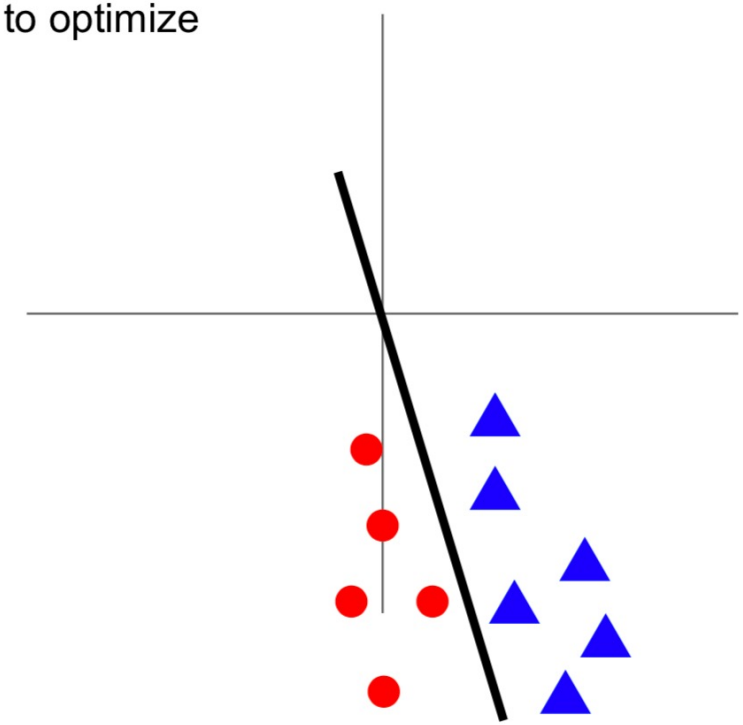
常用的数据预处理方法有:

- 归一化
- 数据扩充
- 随机擦除

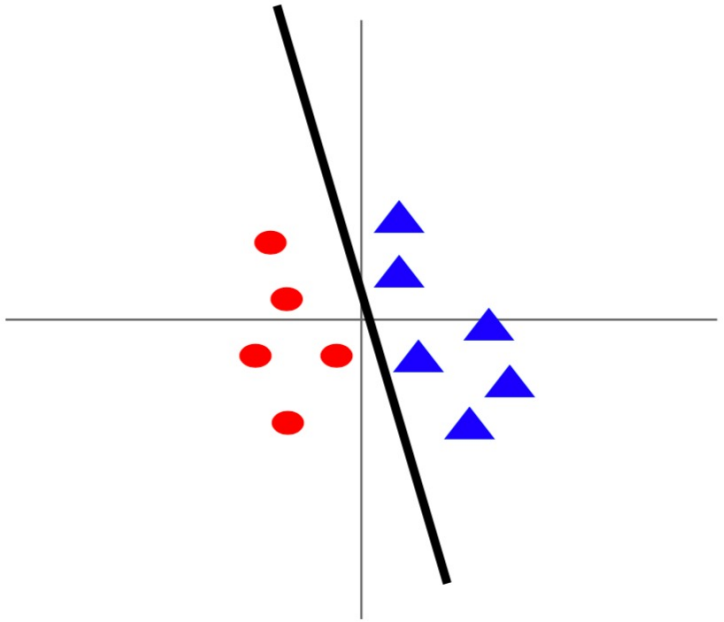
数据归一化

- 归一化输入对DNN学习是非常有好处. 通常, 归一化之后， 输入数据会有零平均值和单位方差。

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

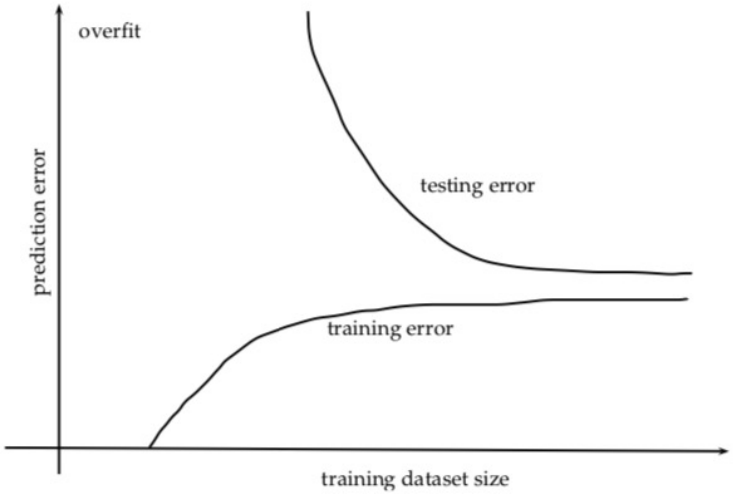


After normalization: less sensitive to small changes in weights; easier to optimize



数据扩充

- 增加数据量能够提高模型的泛化能力.
- 是某种形式上的‘正则化’来避免模型在训练数据上表现过好.



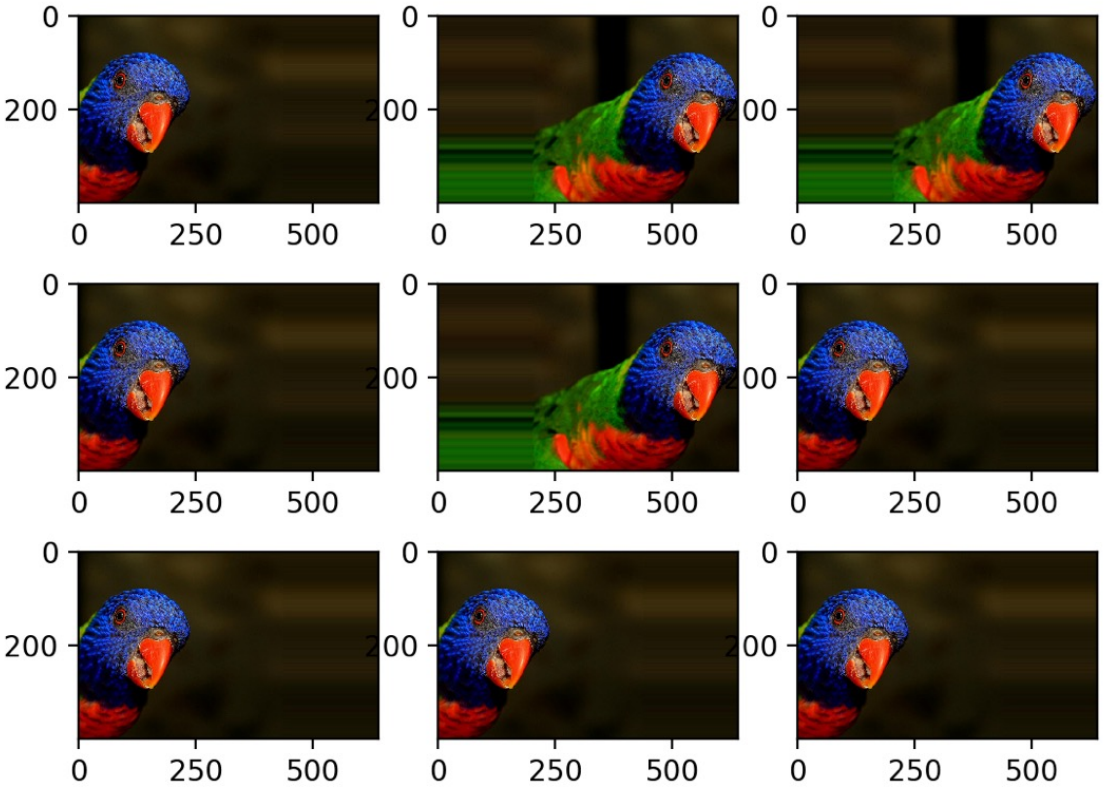
Note: 数据扩充只是在训练过程。在验证和测试集上不推荐这么做。保证增加的数据是合理的

通常图片类型数据增加的一些做法：

- 随机的水平/垂直移动
- 随机水平/垂直翻转
- 随机的亮度变化

数据扩充

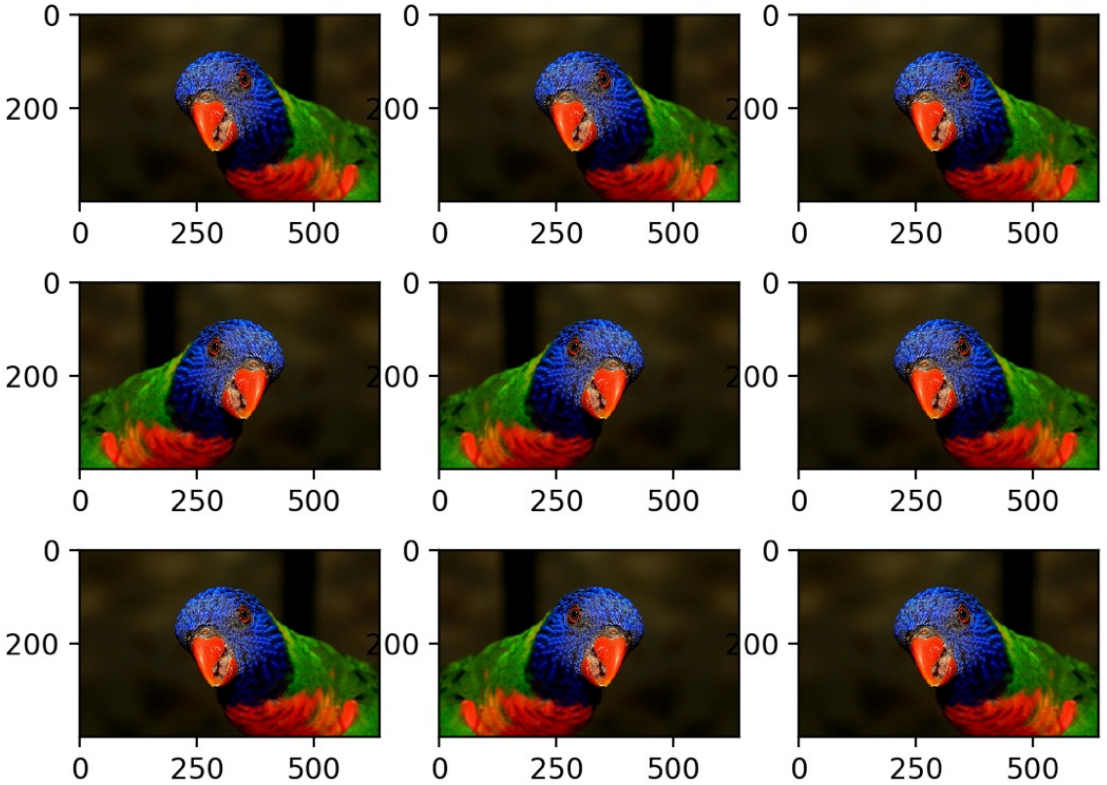
- 随机水平移动



https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/

数据扩充

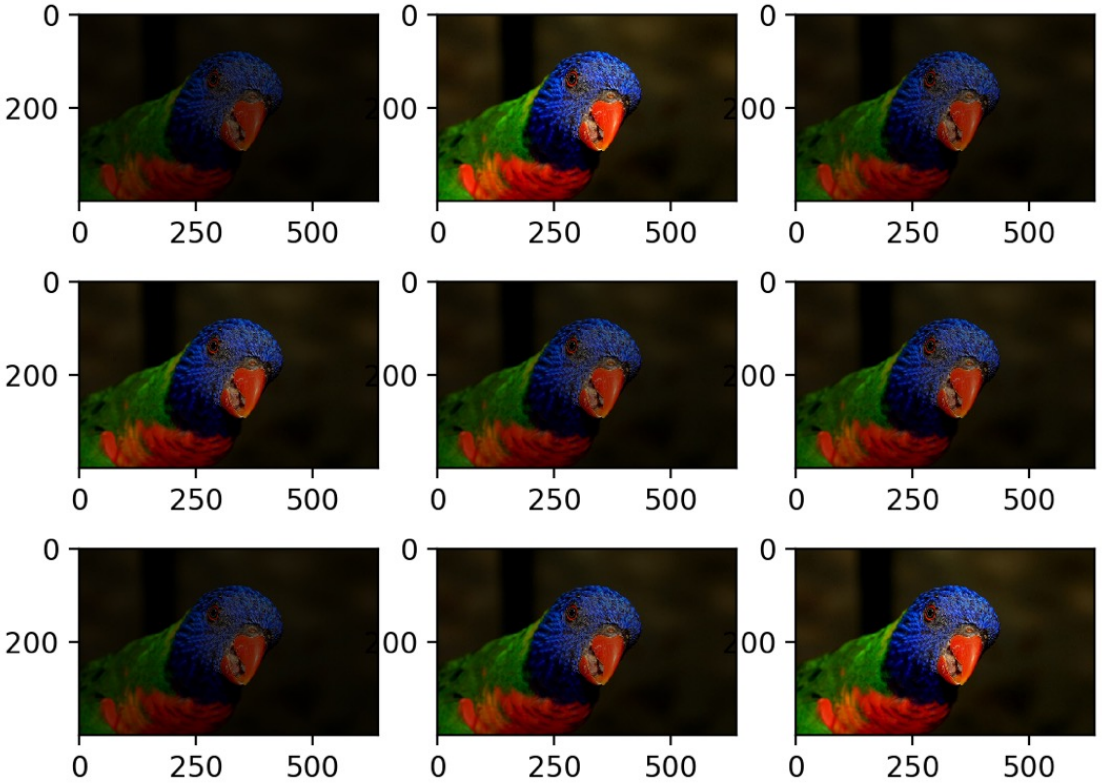
- 随机水平翻转



https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/

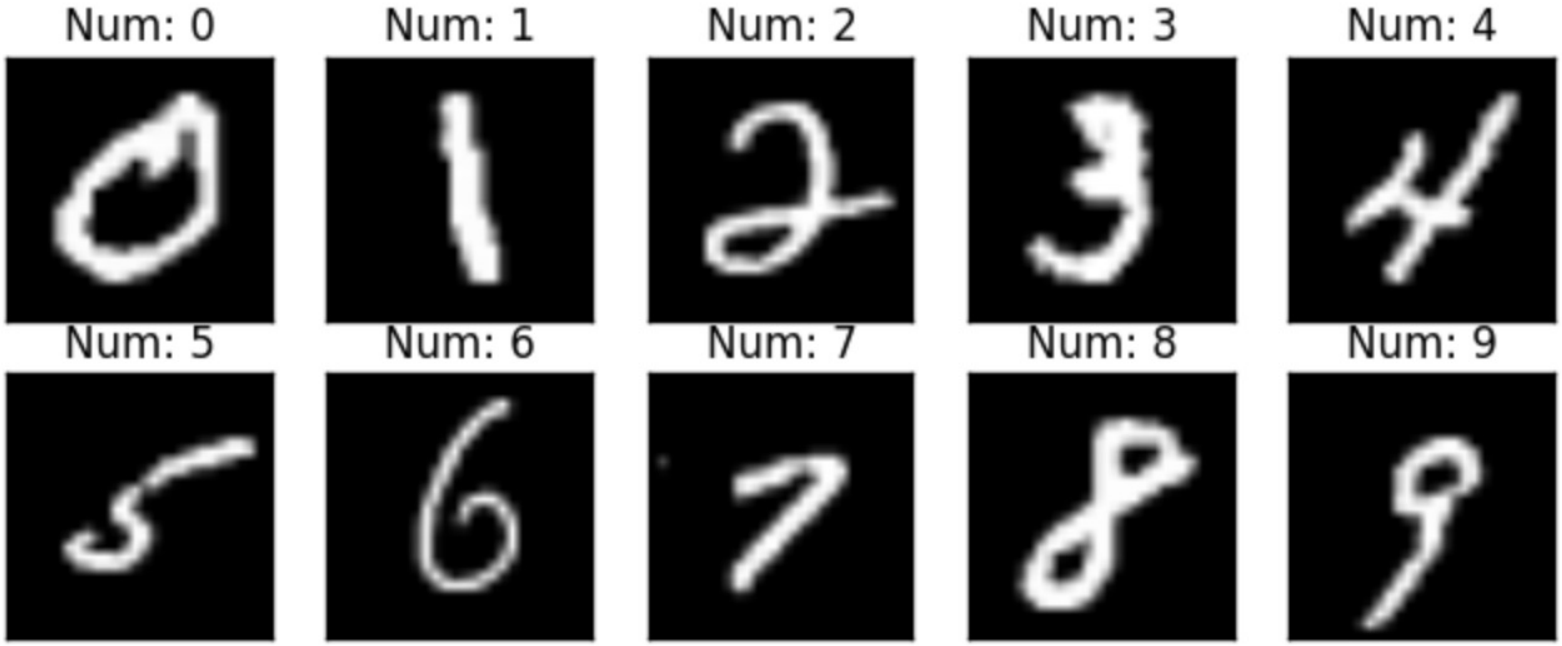
数据扩充

- 随机亮度变化



数据扩充

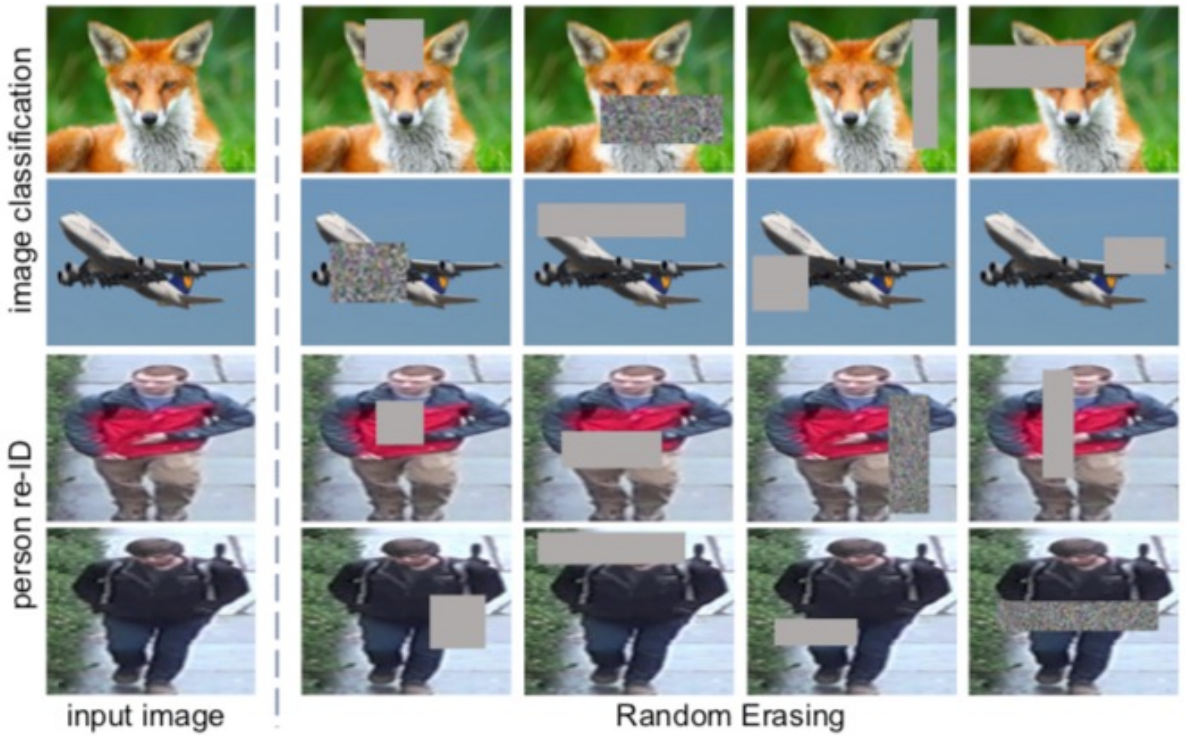
- 确认增加的数据是合理的!



- 比如, 在MNIST训练期间使用随机垂直翻转是不合理的，因为数字6和数字9看起来很相似。

随机擦除

- 随机擦除用随机的模糊来遮挡图像以在训练过程中增加数据。 它是现有数据扩充方法和正则化方法的补充方法。



Zhong, Zhun, et al. "Random erasing data augmentation." (2017)

“撒切尔效应”

- 1980年心理学教授彼得·汤普森（Peter Thompson）的“撒切尔效应”插图

https://en.wikipedia.org/wiki/Thatcher_effect

“撒切尔效应”

- 1980年心理学教授彼得·汤普森（Peter Thompson）的“撒切尔效应”插图
- 当图像上下颠倒时，看起来没有任何毛病。 只有当面部朝上时，我们才识别出的图片，并且图像突然显得怪诞。

https://en.wikipedia.org/wiki/Thatcher_effect

Pytorch实现

```
import torchvision.transforms as transforms
import numpy as np
transform_compose=
transforms.Compose([  # 先归一化再标准化
     transforms.ToTensor()  # 图像归一化，shape 会从(H，W，C)变成(C，H，W)
     transforms.Normalize(
     mean=[0.5, 0.5, 0.5],  # 取决于数据集
     std=[0.5, 0.5, 0.5] ) ,
     transforms.RandomCrop(size, fill=0, padding_mode='symmetric') # 镜像填充
     transforms.RandomHorizontalFlip(p=0.5) # 随机水平翻转
     transforms.RandomVerticalFlip(p=0.5) # 随机垂直翻转

     transforms. ColorJitter(brightness=0.5,  constrast = 0.5, saturation=0.5,
     hue=0,5) # 随机改变图片亮度，对比度，饱和度，色调


])
```

数据预处理

常用的数据预处理方法有:

- 归一化
- 数据扩充
- 随机擦除
- https://pytorch.org/vision/stable/transforms.html

优化策略 – Cont'd

- 激活函数
- 神经网络架构
- 损失函数 & 优化
- 权重初始化
- 正则化
- 数据预处理
- 超参数优化

超参数优化

- Hyper parameter Optimization， HPO
- 超参数是不可训练的参数，需要手动配置以控制DNN的行为和性能。
  - 网络结构，包括神经元之间的连接关系、层数、每层的神经元数量、激活函数的类型。
  - 优化参数，包括优化方法、学习率、小批量的样本数量。
    - 学习率、优化方法
  - 正则化/Dropout率

学习率是最敏感的超参数.

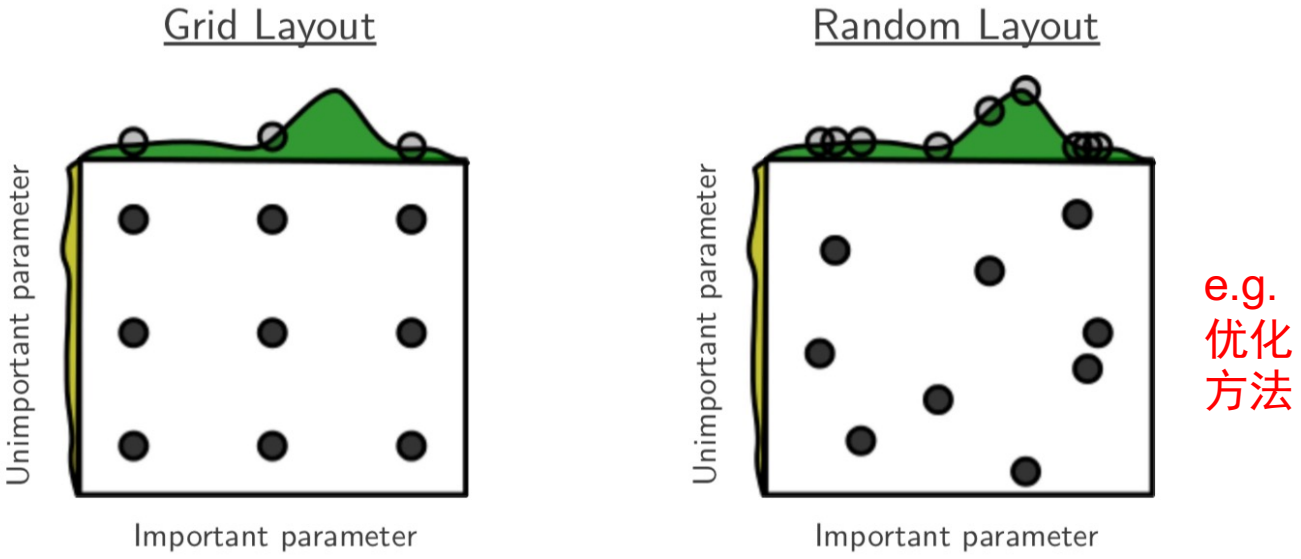
如果只有一个超参数可以调节的时候，调节学习率.

超参数优化

网格布局 vs. 随机布局

网格布局以同等重要性评估每个超参数.

然而，随机布局更加频繁评估重要的超参数从而产生具有有限的调整试验更好的结果.



e.g. 优化方法

e.g. 学习率

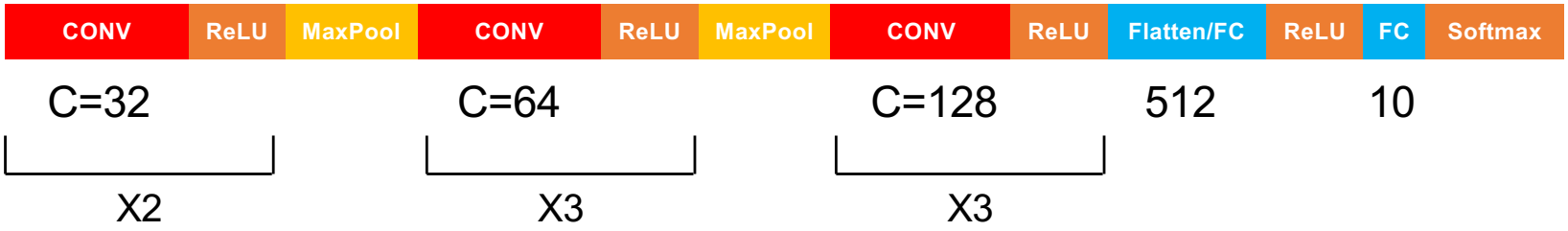
e.g. 优化方法

e.g. 学习率

DNN 架构

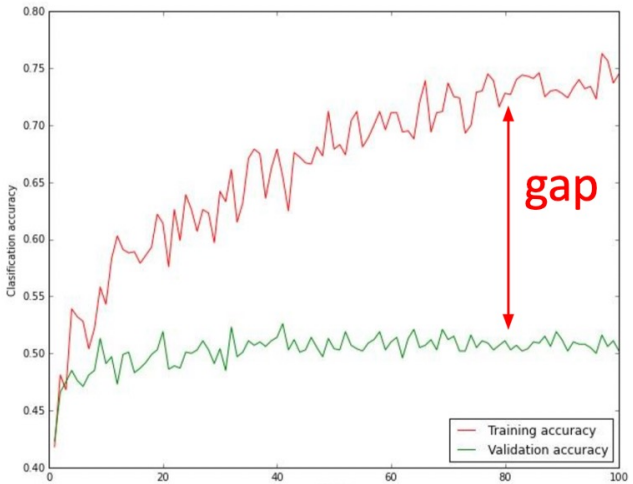
DNN架构是最抽象的一个超参数.

遵循convnet设计的设计准则:



什么时候应该在神经体系结构中添加/删除层/通道?

- 如果泛化差距很大，则通过减少层/通道来减少模型容量.
- 如果训练和验证准确性均较低，则通过增加层/通道来增加模型容量。



- **神经架构搜索(neural architecture search,NAS)**

Google Pathways 模型

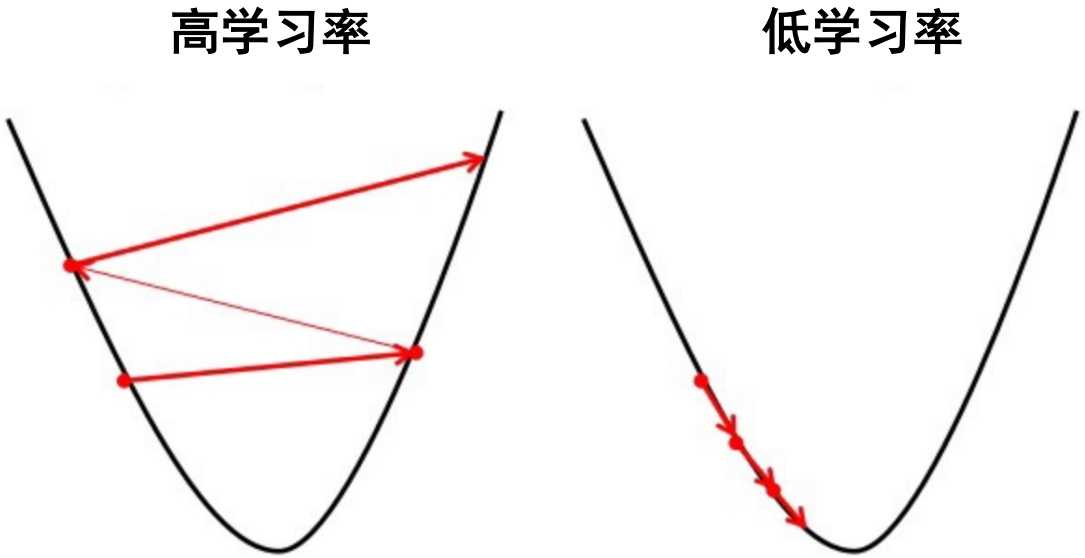
Pathways，称为「下一代AI 架构」——只训练一个模型，就可以处理数以万计的任务类型。

https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/

调节学习率

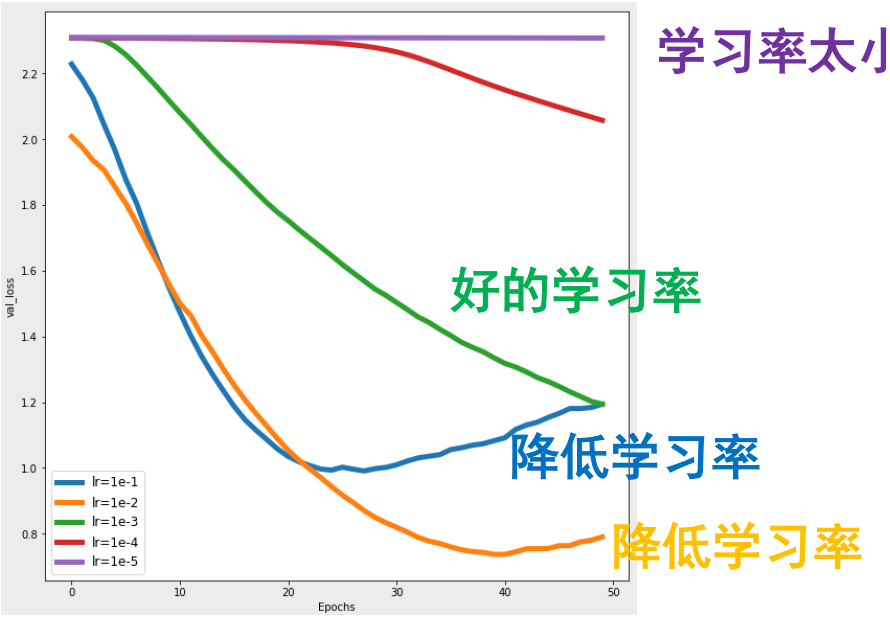
学习率是最重要的超参数配置。

- 高学习率将导致分歧。
- 低学习率会减慢学习过程，并在训练过程中浪费大量时间。.



调节学习率

- 在大多数情况下，带有SGD的学习率最好是0.01.
- 如果损失减小的缓慢，请增大学习率。 或者，如果损失急剧波动甚至上升，则降低学习率。

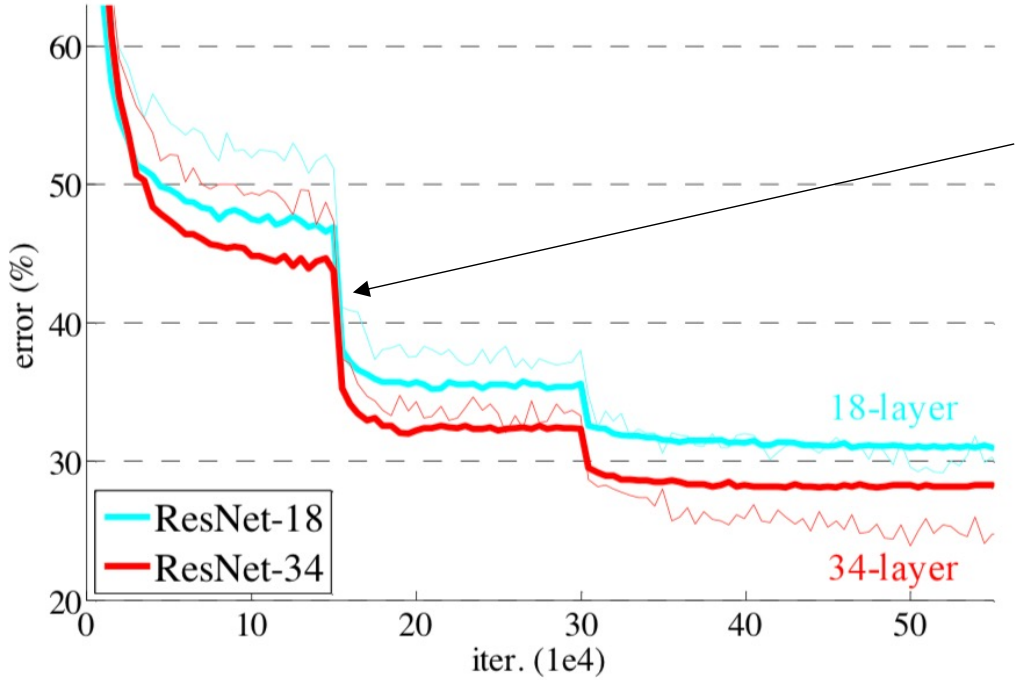


对于其他优化方法，由于不同的优化机制，默认学习率可能会发生变化.

通常，最佳学习率会随着优化算法的复杂性而降低。 例如，使用Adam优化器默认的学习率最好是1e-3.

调节学习率

随着训练进度来调节学习率。 通常，我们在训练过程中每隔几个周期就会降低学习率。



ResNet的学习曲线。 粗线表示验证错误，细线表示训练错误。

这里有学习率衰减

推荐的学习率调度方法:
当验证准确率不变的时候，以0.1被衰减学习率.

Pytorch实现

- torch.optim.lr_scheduler
- 提供了许多调节学习率的方法

- torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=10, threshold=0.0001, threshold_mode='rel', cooldown=0, min_lr=0, eps=1e-08, verbose=False)
- 当某个指标停止更新的时候就更新学习率
- 参数：
  - mode：'min'， 指标停止更新时减少学习率， 'max' ,指标停止时增加学习率
  - *factor*： 控制学习率更新，新学习率=当前学习率*factor
  - patience: 指标停止更新patience个周期后，进行学习率的更新。

```
import torch.optim.lr_scheduler

optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
scheduler = ReduceLROnPlateau(optimizer, 'min')
for epoch in range(10):
    train(...)
    val_loss = validate(...)
    # Note that step should be called after validate()
    scheduler.step(val_loss)
```

正则化的选择

L2 正则化:

- 我们建议使用正则化强度$\lambda = 5e{-}4$作为CNN训练的开始。
- 如果CNN架构较小，建议降低正则化强度。

Dropout:

- 除最终输出层外，每个完全连接的层之后通常使用概率为0.5的Dropout层.
- 在卷积层或池化层之后添加概率为0.05的Dropout层.

优化方法的选择

- Adam 用来检查网络的健全性. 如果Adam不工作, 网络设计/代码一定有问题.
- SGD优化器通常用于获得最新结果。 但是收敛速度慢，可能需要更长的时间才能获得最好的结果。

在这两节课，我们学到了：

- DNN 训练设置
  - 激活函数
  - 神经网络设计
- 优化方法
  - 基于SGD的优化方法
  - Adagrad, RMSprop, Adam
  - 如何选择优化方法
- 权重初始化
  - Xavier 均匀/标准初始化
  - MSRA方法(方差缩放)
  - 如何选择权重初始化方法
- 正则化
  - L-norm 正则化
  - Dropout
  - Batch normalization
  - Label smoothing
- 数据预处理
  - 归一化
  - 数据扩充
  - 随机擦除
- 超参数优化
  - 调节学习率
  - 选择正则化方法和 dropout
  - 选择优化方法

训练日志举例

regularization	weight decay	epochs	lr	pretrained	top-1/5
None	0.0001	90	0.1	None	**76.15, 92.87**
3, 5e-4	0.0001	45	0.01	Yes	~60%
3, 2e-4	0.0001	45	0.01	Yes	66.902, 87.556
3, 1e-4	0.0001	45	0.01	Yes	70.032, 89.696
3, 2e-4	0.0001	70	0.1	Yes	59.772, 82.458
3, 1e-4	0.0001	70	0.1	Yes	crashed@57 epoch
3, 1e-4	0.0001	90	0.1	Yes	**64.210, 85.760**
3, 7e-5	0.0001	90	0.1	Yes	**67.118, 87.756**
3, 5e-5	0.0001	90	0.1	Yes	**69.190, 88.964**
3, 5e-5	0.0001	70	0.1	Yes	69.540, 89.166
3, 4e-5	0.0001	90	0.1	Yes	**70.658, 89.990**
3, 3e-5	0.0001	90	0.1	Yes	**71.668, 90.622**
3, 3e-5	**0**	90	0.1	Yes	74.902, 92.122
3, 2e-5	0.0001	70	0.1	Yes	crashed@32 epoch
3, 2e-5	0.0001	90	0.1	Yes	**73.038, 91.476**
3, 1e-5	0.0001	90	0.1	Yes	**74.962, 92.252**
3, 1e-5	0.0001	90	0.1	**NO**	73.154, 91.352