第七讲 训练神经网络二

2019年4月30日 10:54

1.更好的优化

1.1权重初始化

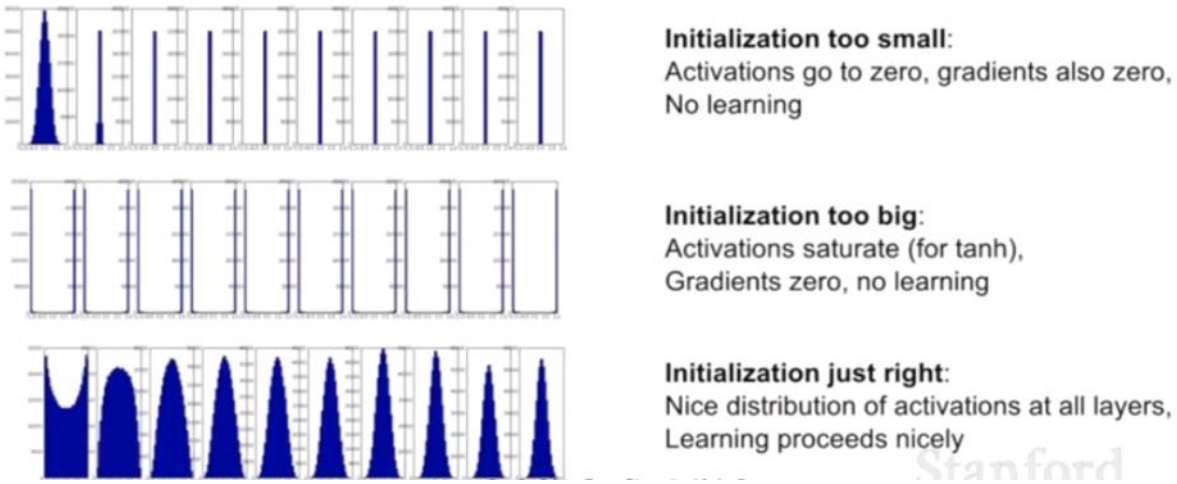


图7.1.1 参数矩阵的初始化选择

在权重初始化时，初始化参数过小激活值会消失，（激活函数每次都乘以一个很小的值），初始化参数过大则会导致激活函数饱和，（越往前走数字越大！）。

1.2预处理

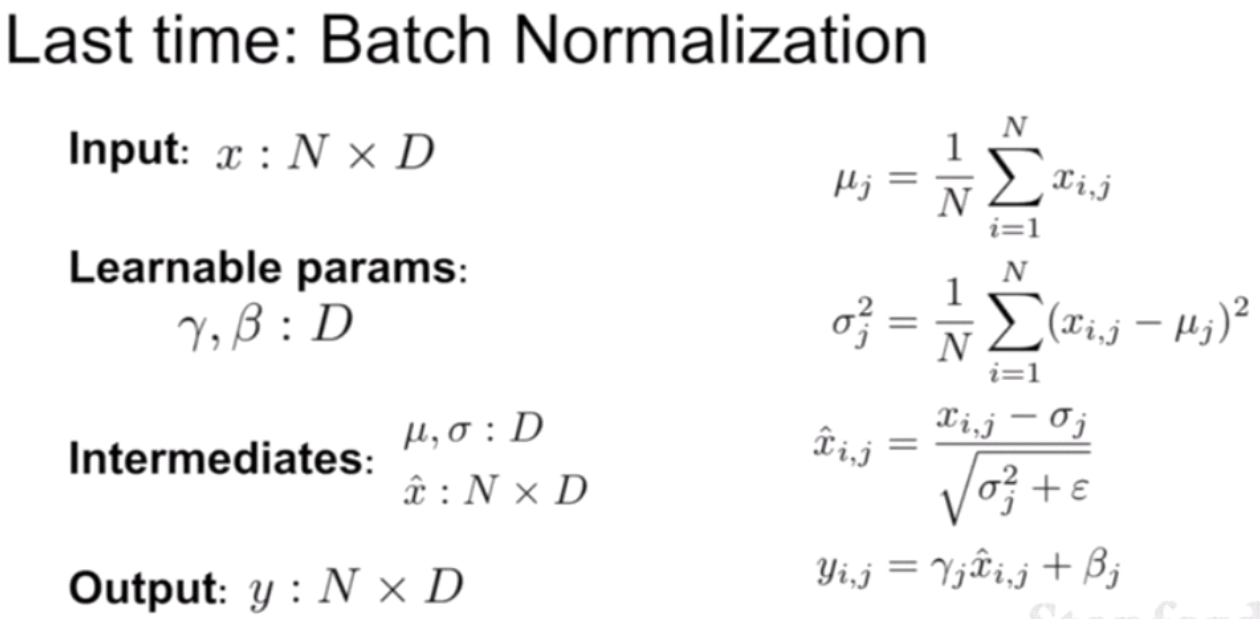


图7.1.2 批量归一化总结

选取一个很小的batch计算其均值和方差，使用这些数据对输入数据进行归一化操作。

1.3 优化小结

1.选择激活函数
2.权重初始化
3.数据预处理
4.批量归一化
5.跟踪学习(监督学习过程中的动态) 图7.1.3
6.超参数(优化)搜索

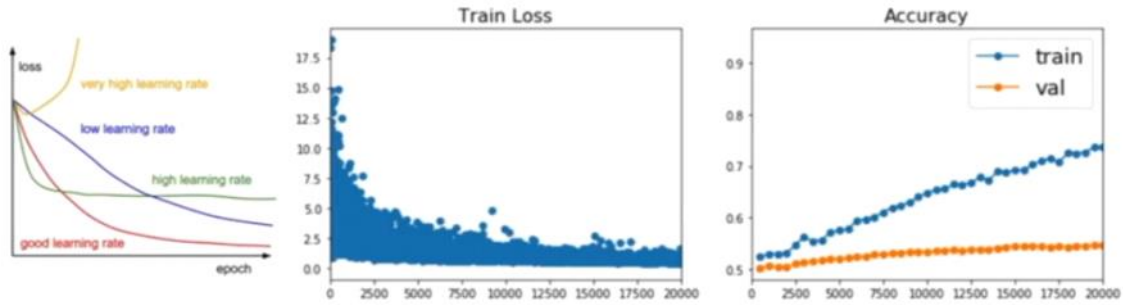
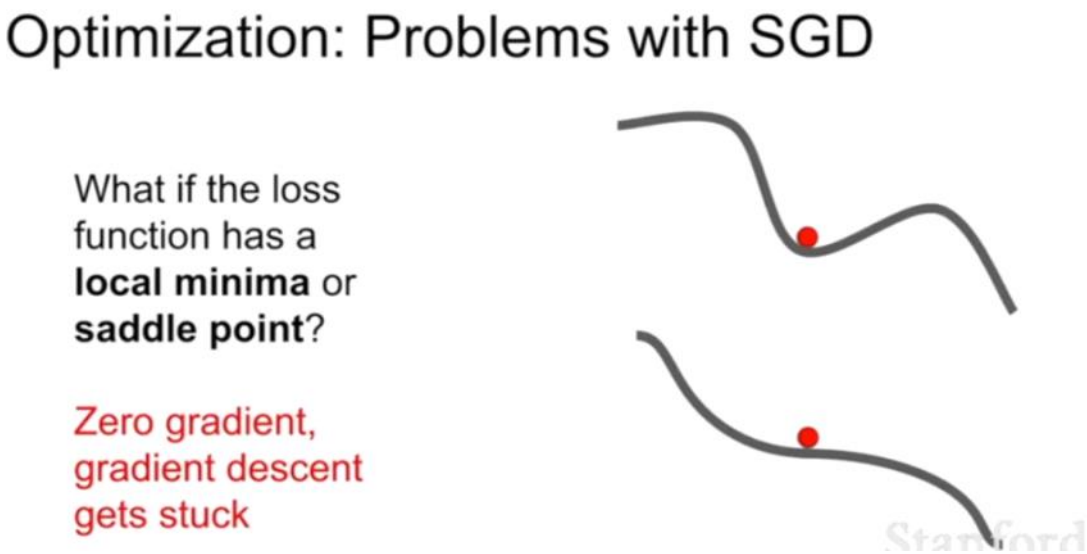


图7.1.3 跟踪学习中的动态变化

例如最右侧曲线：随着训练集的增加训练集的准确率上升而测试集却不怎么变化，这个说明了模型过拟合了，要加入正则化项。

1.4 优化器算法

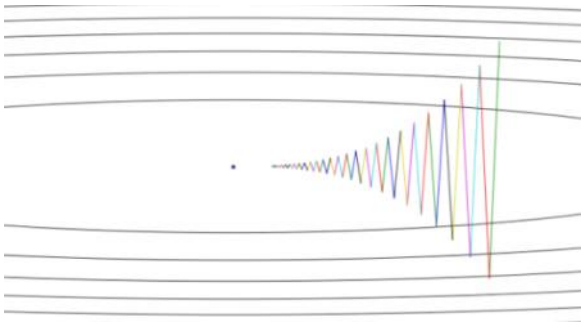
1.梯度下降优化



梯度下降过程中遇到局部最小值(鞍点问题)，使用随机梯度下降可能跳出这个局部最小值。

参数维度越高，遇到鞍点问题越少。

动量SGD(SGD with momentum)：学习率根据梯度动态调整，这样会使损失函数快速迭代。



上图的第一部分是梯度下降的动态变化，中心点是损失函数的最优点，而在使用梯度下降的时候明显X轴和Y轴在使用同一学习率的时候对X轴来说偏小，而对Y轴来说偏大(从向量角度看，是学习率的方向不对)，所以我们要想办法加大X轴方向的步长(使梯度方向更偏向X)，从而出现了动量SGD。

动量SGD计算过程：

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

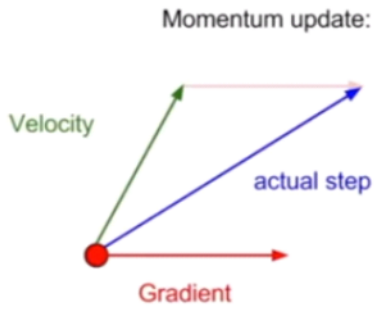
rho：值越大之前的梯度对现在的方向影响越大，一般取0.5 、0.9 、0.99

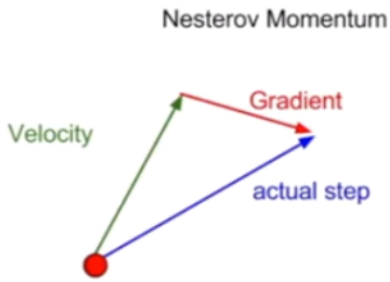
Nesterov momentum：

Nesterov momentum计算过程：

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

Nesterov Momentum

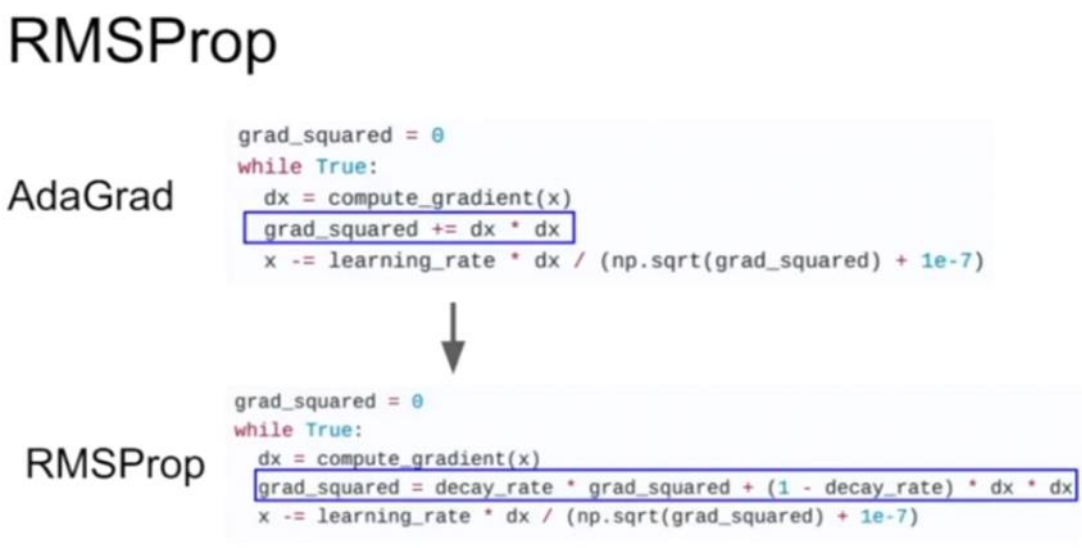




Nesterov momentum: 与一般的动量SGD不同，可以理解为nesterov动量在标准动量方法中添加了一个校正因子。？ 提前一步知道梯度的下次迭代值，计算V。

$$
\begin{aligned}
VdW &= \beta VdW - \alpha dW \\
Vdb &= \beta Vdb - \alpha db \\
W &= W + \beta VdW - \alpha dW \\
b &= b + \beta Vdb - \alpha db
\end{aligned}
$$

RMSProp & AdaGrad:



AdaGrad与动量SGD：

动量SGD通过叠加梯度获得速度这一概念，然后梯度下降沿着速度方向走，而在Adagrad和RMSProp中使用的是另外一种思想，先求梯度平方的估计值然后除以梯度平方的实际值。

结合AdaGrad与动量SGD ==>> Adam算法：

```
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))
```

adam存在的问题：

第二部分动量在初始化时可能会得到一个很小的值，此时的步长为first_moment/(second_moment平方),这会导致我们刚开始迭代的时候会有一个很大的步长，很可能会使我们的算法迭代失败。所以adam算法在后面加上偏置项。

```
first_moment = 0
second_moment = 0
for t in range(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

随着时间降低学习率，在梯度下降中，损失函数在最优值附近徘徊时，减小学习率会进一步优化损失值。

1.5 优化综述

《An overview of gradient descent optimization algorithms》

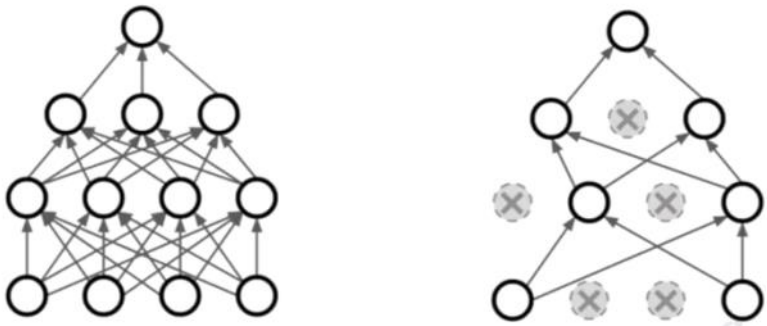
论文：https://arxiv.org/abs/1609.04747

2.正则化

1.dropout

Regularization: Dropout

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



在前向传播的时候随机置零一些神经元，激活函数置零，每层的置零神经元是个超参数，一般选取一半。

```
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)
```

3.迁移学习

训练一个CNN网络需要大量的数据，当我们的数据集比较小时 -->迁移学习

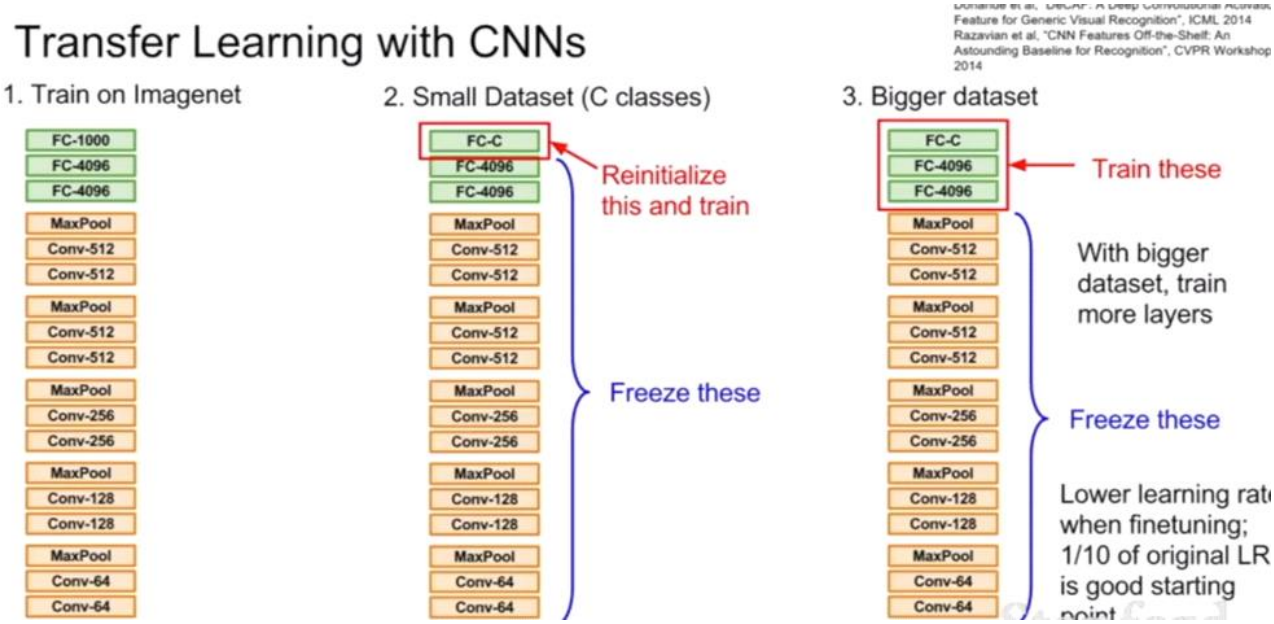
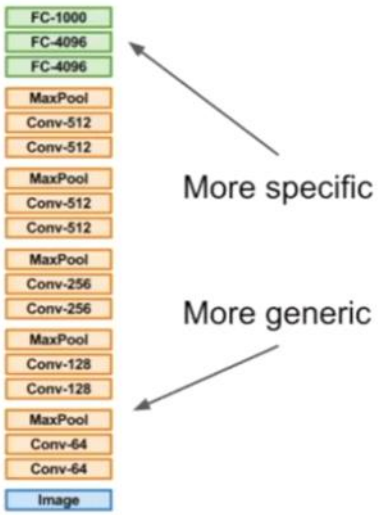


图 7.3.1 迁移学习

迁移学习使用场景：当我们有一些小的数据集比如给十种常见的图片分类，我们可以使用之前在Imagenet上训练好的模型，保持模型的图像特征提取部分权重不变，仅仅训练全连接层。

如果我们的图像数据比较特殊，例如CT，或其他领域的一些专用图片，那么迁移学习就不能用了。

迁移学习保留的是整个模型的特征提取部分，重新训练线性分类部分。



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

图 7.3.2 适用范围

数据集相似的话特征提取部分也会相似，所以只需要训练全连接层的线性分类部分。

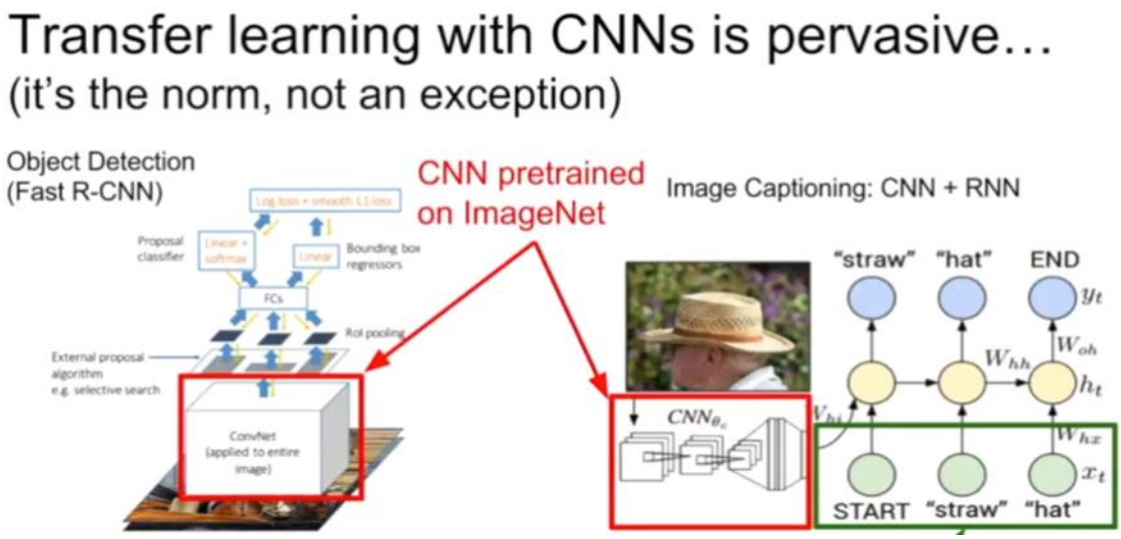


图 7.3.3 论文中类似这种结构就是使用预训练的CNN