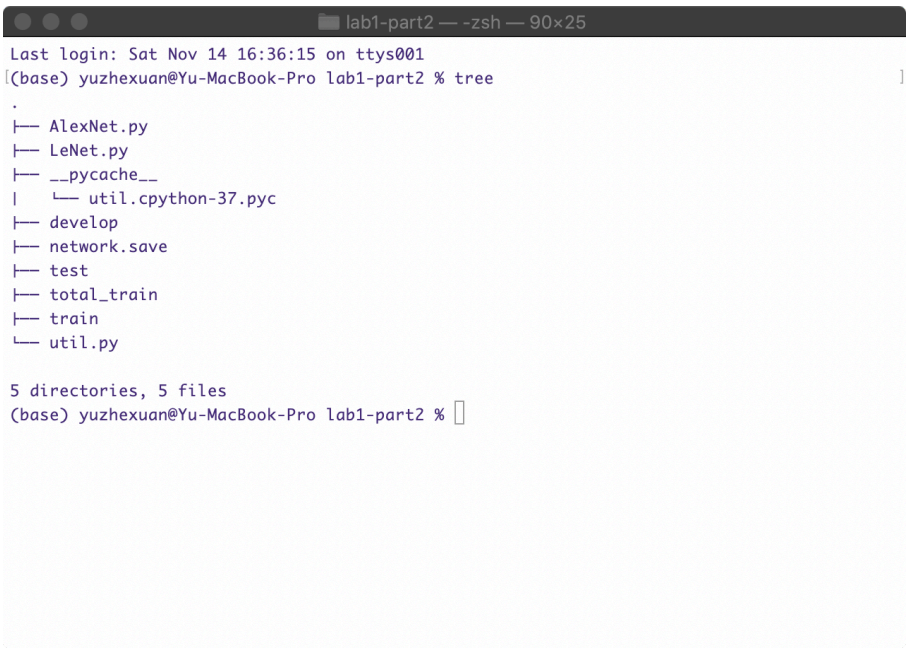
智能系统原理与开发 Lab1-part2

18302010018 俞哲轩

代码基本架构

程序结构

- **File:** `AlexNet.py`和`LeNet.py`为两个卷积神经网络
- **File:** `util.py`为工具类，包含加载数据等方法
- **File:** `network.save`为保存的神经网络，可直接加载进行测试
- **Directory:** `develop`、`test`、`train`和`total_train`为用于训练和检验的数数据集



LeNet

最传统的卷积神经网络，即**LeNet**，具有七层：`卷积层1` — `池化层1` — `卷积层2` — `池化层2` — `全连接层1` — `全连接层2` — `全连接层3` 。

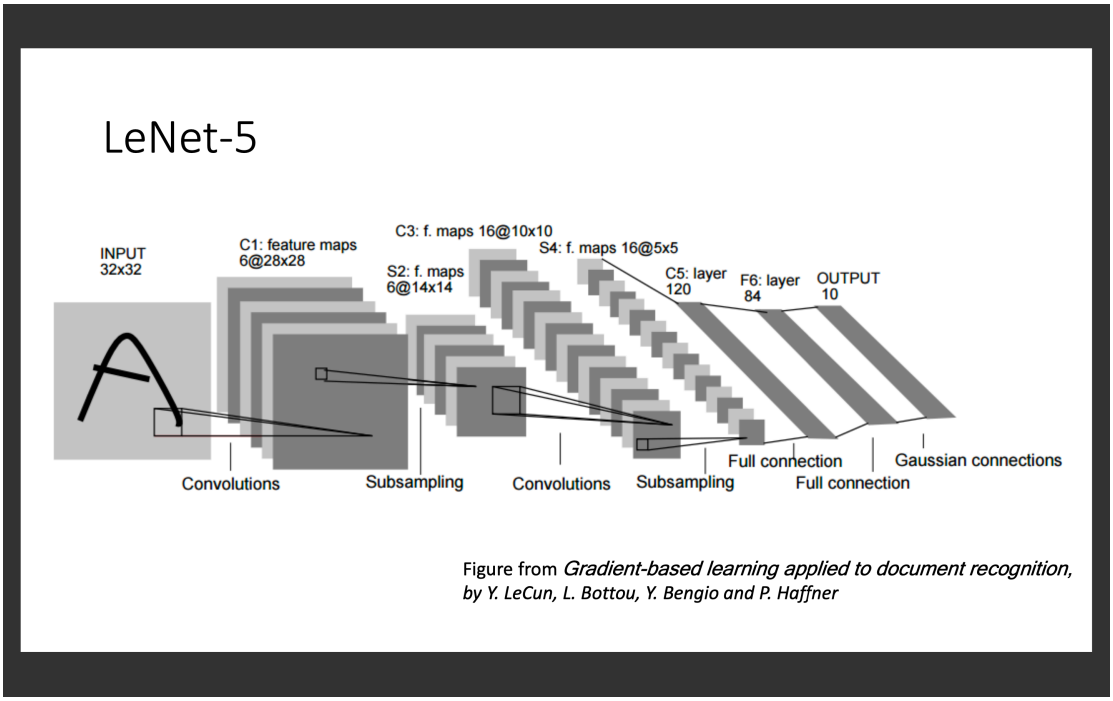


Figure from *Gradient-based learning applied to document recognition, by Y. LeCun, L. Bottou, Y. Bengio and P. Haffner*

经过对网络的精简和改进，我采用了五层的设计：`卷积层1` — `池化层1` — `卷积层2` — `池化层2` — `全连接层` 。

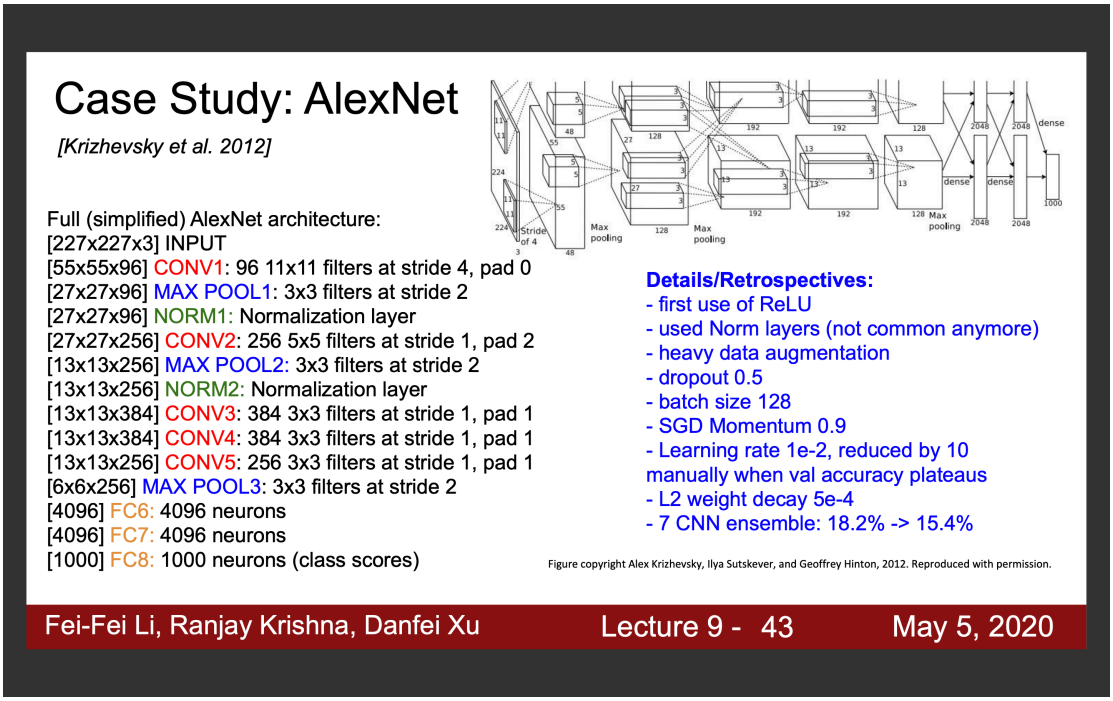
```
class LeNet(nn.Module, ABC):
  def __init__(self):
    super(LeNet, self).__init__()

    self.conv1 = nn.Sequential(
      nn.Conv2d(
        in_channels=1,
        out_channels=16,
        kernel_size=5,
        stride=1,
        padding=2,
      ), # Convolutional Layers
      nn.BatchNorm2d(16), # Batch Normalization
      nn.ReLU(), # Activation Function
      nn.MaxPool2d(kernel_size=2), # Pooling Layer
    )

    self.conv2 = nn.Sequential(
      nn.Conv2d(
        in_channels=16,
        out_channels=32,
        kernel_size=5,
        stride=1,
```

```
        padding=2
      ), # Convolutional Layers
      nn.BatchNorm2d(32), # Batch Normalization
      nn.ReLU(), # Activation Function
      nn.MaxPool2d(kernel_size=2), # Pooling Layer
    )

    self.output = nn.Sequential(
      nn.Dropout(p=0.5), # Dropout
      nn.Linear(in_features=32 * 7 * 7, out_features=12), # Fully-Connected
Layer
    )


  def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = x.view(x.size(0), -1)
    output = self.output(x)
    return output
```

AlexNet

AlexNet是在**LeNet-5**的基础上，形成的更深的神经网络，同时采用了Local **Response Normalization**、**Dropout**和**数据增强**，以避免模型过拟合。

1. **Local Response Normalization**：对局部神经元的活动创建竞争机制，使得其中响应比较大的值变得相对更大，并抑制其他反馈较小的神经元，增强了模型的泛化能力。
2. **Dropout**：随机忽略一部分神经元，在最后几个全连接层使用Dropout，以避免模型过拟合。
3. **数据增强**：随机从256*256的原始图像中截取224*224大小的区域（以及水平翻转的镜像），相当于增加了$2*(256-224)^2$=2048倍的数据量；进行预测时，则是取图片的四个角加中间共5个位置，并进行左右翻转，一共获得10张图片，对他们进行预测并对10次结果求均值。



考虑到实际的任务的要求（12分类问题），相比较于ImageNet的1000分类问题较小，故对AlexNet的模型和训练进行了精简。

```
class AlexNet(nn.Module, ABC):
  def __init__(self):
    super(AlexNet, self).__init__()

    self.features = nn.Sequential(
      nn.Conv2d(1, 16, 5, 1, 2),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2),
      nn.LocalResponseNorm(16),
      nn.Conv2d(16, 32, 5, 1, 2),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2),
      nn.LocalResponseNorm(32),
      nn.Conv2d(32, 96, 3, 1, 1),
      nn.ReLU(inplace=True),
      nn.Conv2d(96, 64, 3, 1, 1),
      nn.ReLU(inplace=True),
      nn.Conv2d(64, 64, 3, 1, 1),
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=1),
    )

    self.classifier = nn.Sequential(
      nn.Dropout(p=0.5),
      nn.Linear(64 * 7 * 7, 2048),
      nn.ReLU(inplace=True),
      nn.Dropout(p=0.5),
      nn.Linear(2048, 2048),
```

```
      nn.ReLU(inplace=True),
      nn.Linear(2048, 12),
    )


  def forward(self, x):
    x = self.features(x)
    x = x.view(x.size(0), -1)
    output = self.classifier(x)
    return output
```

Train and Test

```
def train():
  optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate,
weight_decay=weight_decay) # set optimizer
  loss_function = nn.CrossEntropyLoss() # set loss function

  train_loader = data.DataLoader(dataset=train_data, batch_size=batch_size,
shuffle=True, num_workers=4)
  for epoch in range(epochs):
    for step, (batch_x, batch_y) in enumerate(train_loader):
      # feed forward & back propagation
      output = cnn(batch_x)
      loss = loss_function(output, batch_y)
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()


def test(dataset):
  data_loader = data.DataLoader(dataset=dataset)
  correct_num = 0
  for step, (test_x, test_y) in enumerate(data_loader):
    test_output = cnn(test_x)
    pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
    if pred_y == test_y.item():
      correct_num += 1
```

用GPU进行训练，需要将network、inputs和labels都从CPU放入GPU中；用CPU进行预测结果的比较，需要将labels从GPU放入CPU中。

```
# use GPU to train
cnn = LeNet()
cnn.cuda()

for epoch in range(epochs):
    for step, (train_x, train_y) in enumerate(train_loader):
      train_x = train_x.cuda()
      train_y = train_y.cuda()

# use CPU to predict
pred_y = torch.max(test_output, 1)[1].data.cpu().numpy().squeeze()
```

改进网络

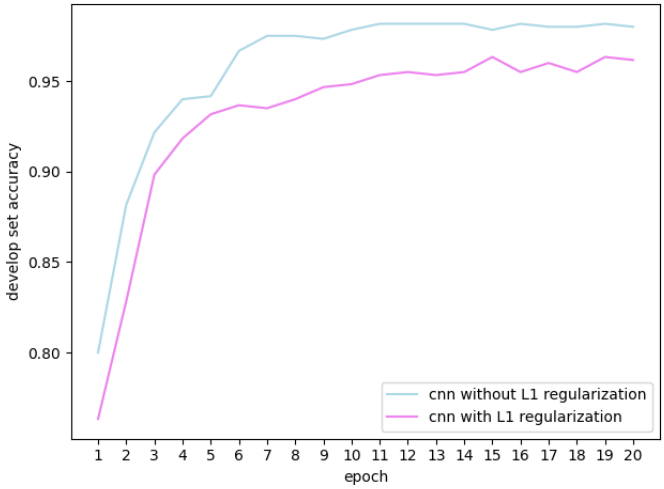
Regularization(L1 & L2 Regularization)

$$l1:\Omega(w)=||w||_1=\sum_i|w_i|$$

代码实现：

```
def train_with_L1Reg(network):
  regularization_loss = 0
  for param in network.parameters():
    regularization_loss += torch.sum(abs(param))

  for epoch in range(epochs):
    for step, (batch_x, batch_y) in enumerate(train_loader):
      output = network(batch_x)
      classify_loss = loss_function(output, batch_y)
      loss = classify_loss + 0.001 * regularization_loss # L1 Regularization
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
```

使用了L1 Regularization和未使用L1 Regularization的对比：

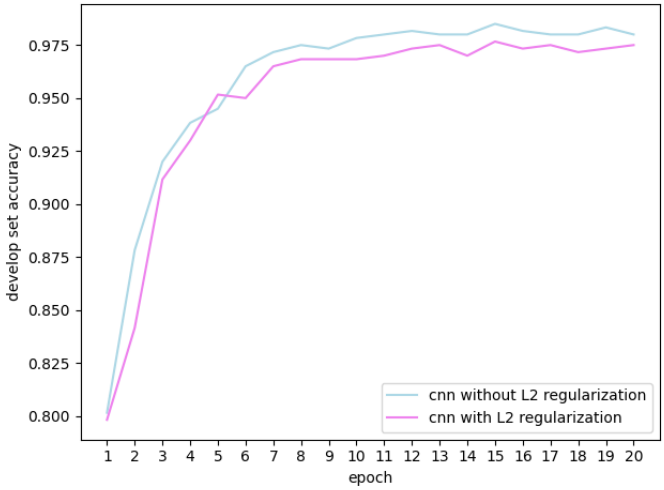


$$l2:\Omega(w)=||w||_2^2=\sum_i w_i^2$$

代码实现：

```
def train_with_L2Reg(network):
  optimizer = torch.optim.Adam(network.parameters(), lr=learning_rate,
weight_decay=0.01) # L2 Regularization
```

使用了L2 Regularization和未使用L2 Regularization的对比：

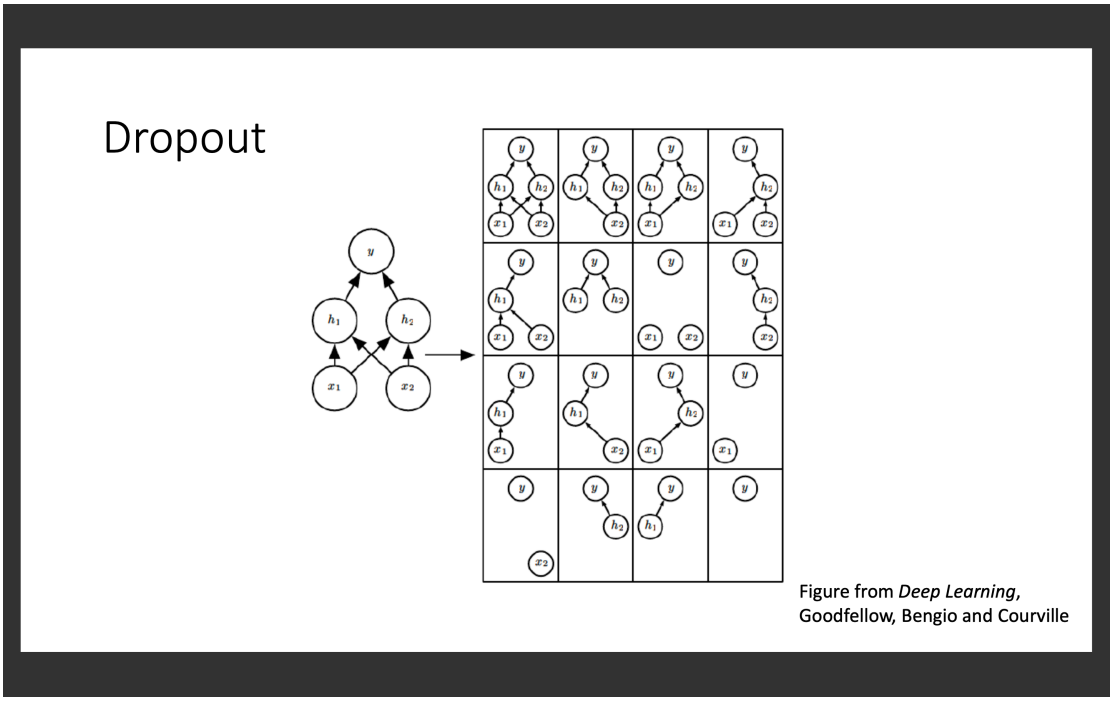


我们可以看到，使用了Regularization之后的模型，明显正确率低于未使用Regularization的模型，其原因可能是：

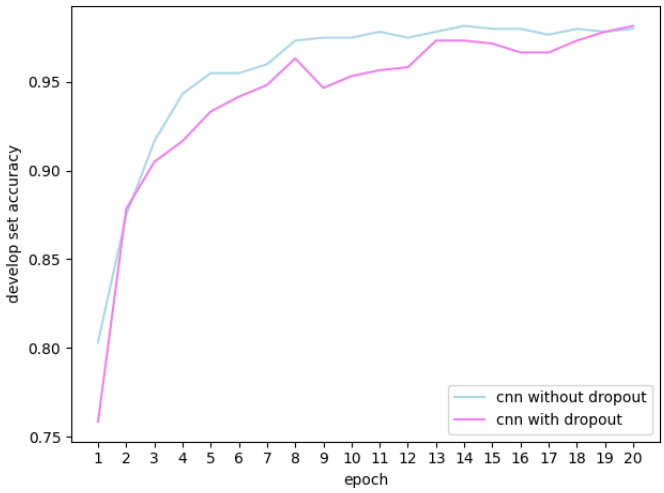
1. 任务较为简单，并未出现明显的过拟合现象，Regularization的效果并不明显
2. L2 Regularization表现相较于L1 Regularization更好，可能是因为更加适合本任务

Dropout

Dropout: 在前向传播的时候，让某个神经元的激活值以一定的概率p停止工作，这样可以使模型泛化性更强，因为它不会太依赖某些局部的特征。

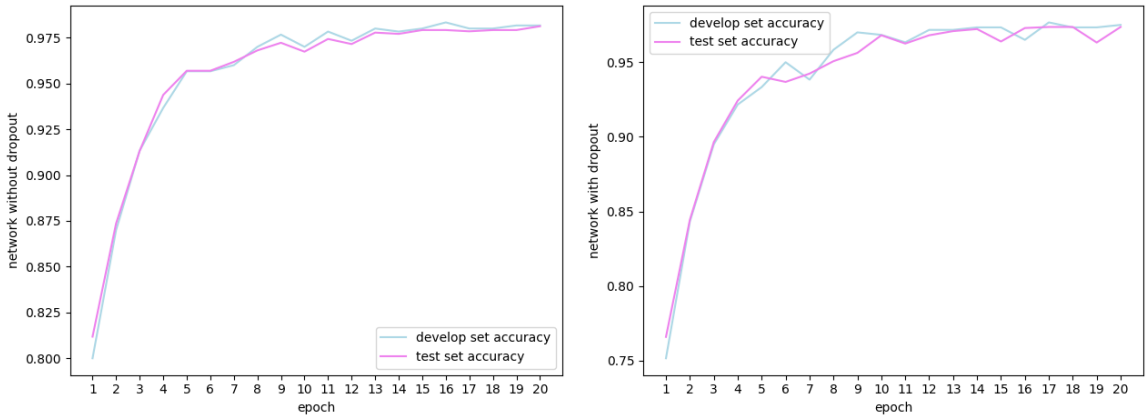


使用了dropout和未使用dropout的对比：



我们可以看到，使用了dropout之后，模型收敛的速度较慢，且正确率略低于未使用dropout的模型。

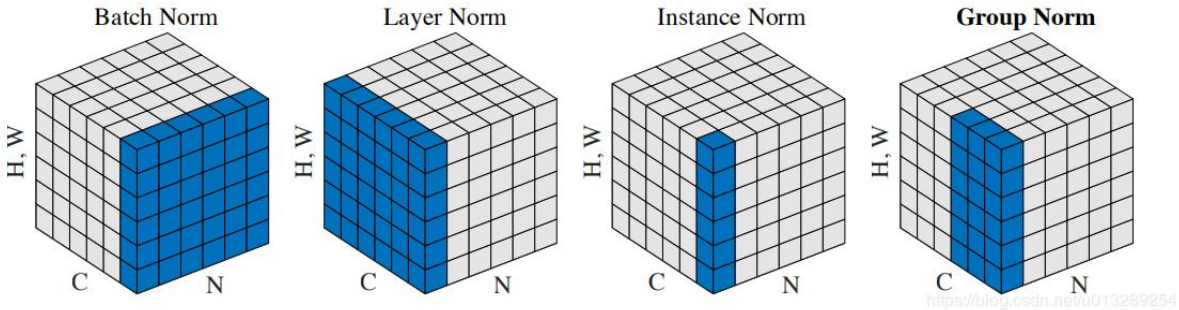
左图：未使用dropout；右图：使用dropout



通过实际效果，我们发现使用了dropout之后，训练的收敛速度较慢，且模型在test set上测试的正确率有所下降，我们认为可能原因有：

1. 任务较为简单，dropout使得模型的输出降低，从而使得每次epoch的改变量降低，“学习”速度减慢
2. 任务较为简单，并**未出现**明显的过拟合现象，dropout的效果并不明显
3. 任务较为简单，模型抽取特征的能力太强，即未使用dropout的模型**已经**过拟合了**（2，3两点将会在之后验证猜想）**
4. 在训练网络的时候加入dropout层，在测试的时候并未移除

Normalization(Local Response Normalization & Batch Normalization)



LRN: 在AlexNet中使用到了LRN层，LRN采用了横向抑制，即兴奋的神经细胞抑制周围神经细胞的能力。应用到深度神经网络中，这种横向抑制的目的是进行局部对比度增强，以便将局部特征在下一层得到表达。

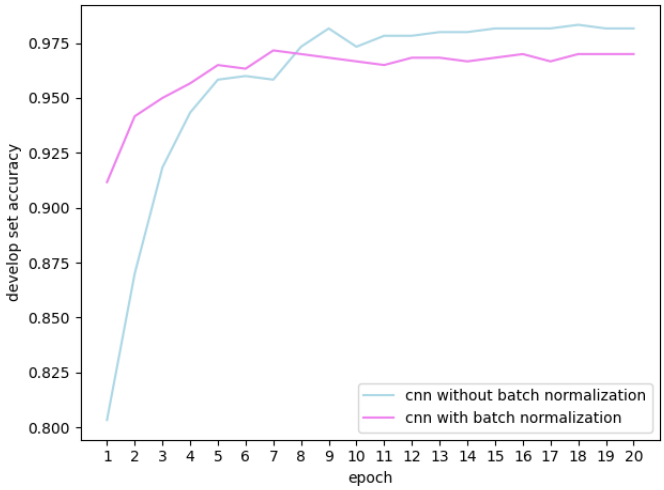
公式如下：

$$b^i_{x,y} = a^i_{x,y}/(k + \alpha \sum^{min(N-1,(i+n)/2)}_{j=max(0,(i-n)/2} (a^j_{x,y})^2)^\beta$$

Pytorch 源码中对实现了LRN的操作，与AlexNet中的方法略有不同，α所乘的项由所有数值的平方和变成了所有数值的均值，但核心都起到了横向抑制的作用。

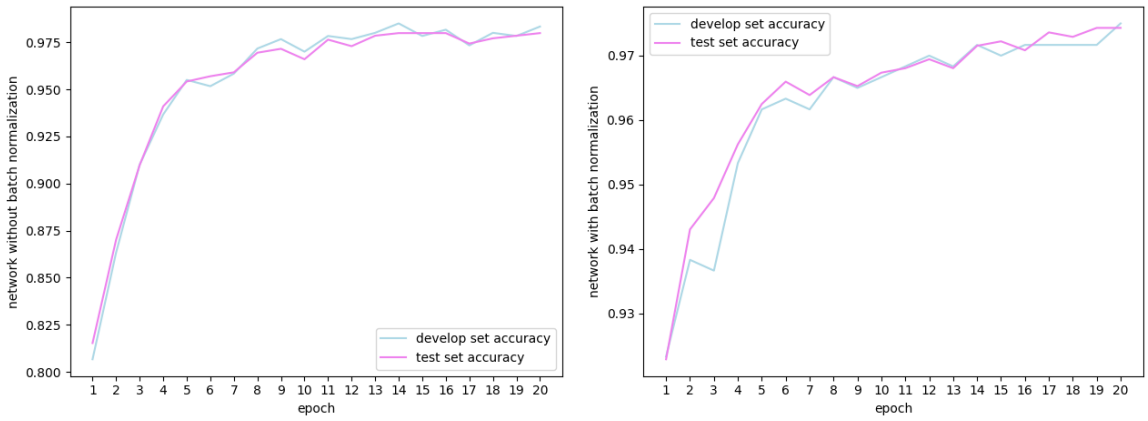
BN: 对于每个隐层神经元，把逐渐向非线性函数映射后向取值区间极限饱和区靠拢的输入分布强制拉回到均值为0方差为1的比较标准的正态分布，使得非线性变换函数的输入值落入对输入比较敏感的区域，以此避免梯度消失问题。

使用了bn和未使用bn的对比：



我们可以看到，使用了batch normalization之后，模型收敛的较为平缓，且随着epoch的次数增加，正确率逐渐被未使用batch normalization的模型超过。

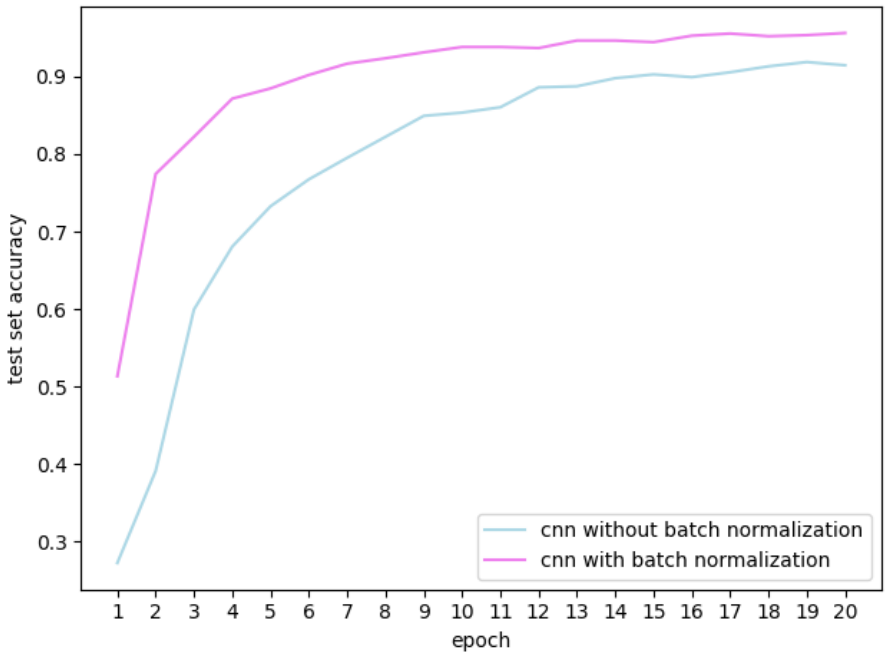
左图：未使用bn；右图：使用bn



通过实际效果，我们发现使用了batch normalization之后，模型在test set上测试的正确率随着epoch的增长，逐渐低于未使用batch normalization的模型，我们认为可能原因有：

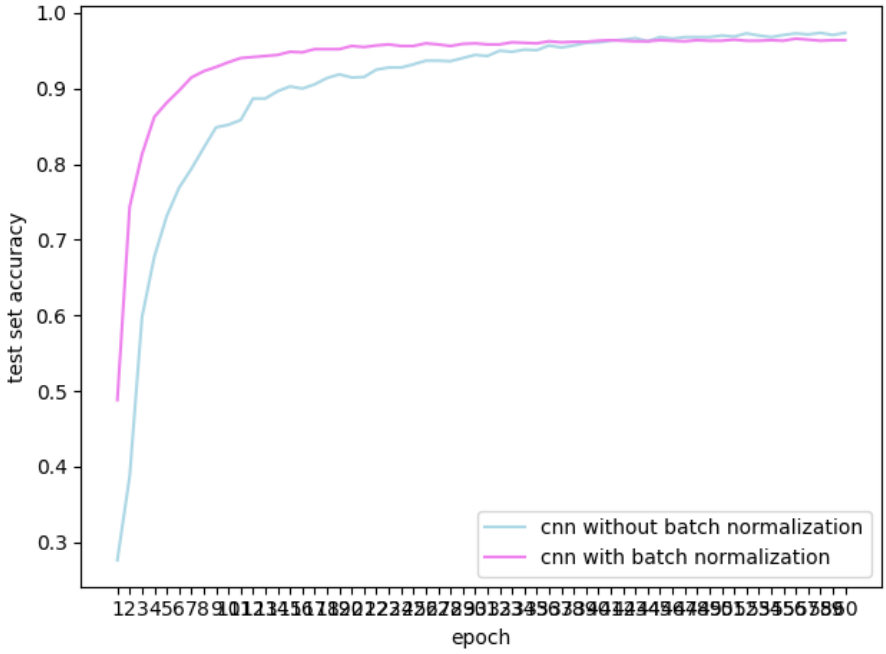
1. 任务较为简单，并未出现明显的过拟合现象，batch normalization的效果并不明显
2. 任务较为简单，模型抽取特征的能力太强，即未使用batch normalization的模型**已经**过拟合了**（1，2两点将会在之后验证猜想）**
3. 为了对比实验，batch的值、learning rate的值设置的较小，在较大的情况下可能batch normalization的表现更好

验证猜想，将batch从64提升至1024：



我们可以看到，此时使用batch normalization的模型**明显优于**未使用batch normalization的模型；但是这也有可能是因为batch size**过大**，导致未使用batch normalization的模型**并未很好收敛**，需要更多的epoch来证明。

将epoch从20提升至60：



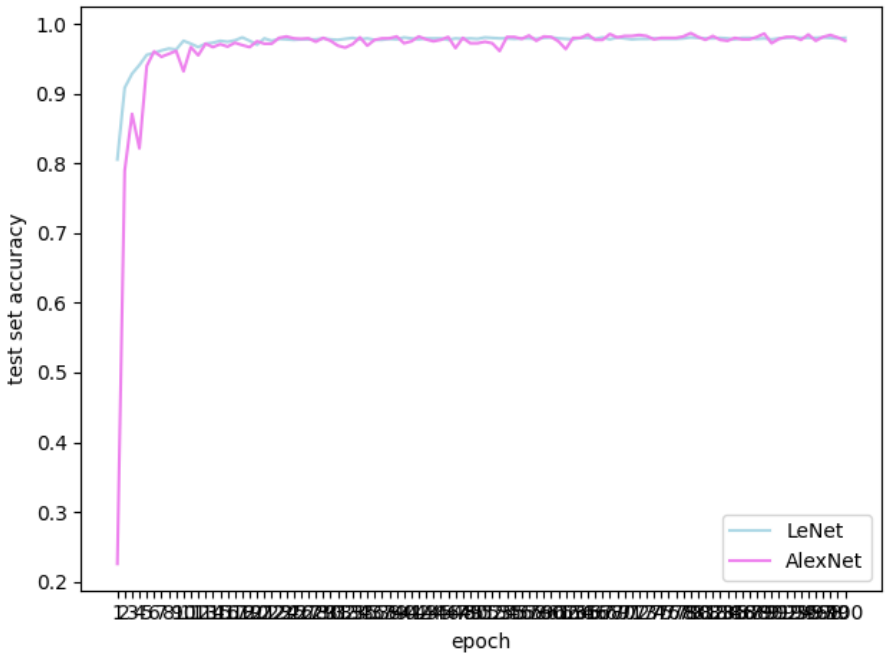
随着epoch的增加，我们可以验证之前的猜想：确实是因为epoch数量较低，未使用batch normalization的网络**并未很好收敛**。

同时我们可以观察到，使用了batch normalization的网络收敛之后，正确率**始终较为平稳**；而未使用batch normalization的网络正确率的趋势为**逐步提升**，即使用了batch normalization之后，网络进一步抽取特征的能力被**“抑制”**了，这既可以**有效防止过拟合**，也会**部分阻止网络性能的提升**。

可能在**较大的分类问题**中，batch normalization可以表现出很好的性能，但是在本任务中，**最普通的cnn**已经具有足够强大的性能了。

LeNet vs AlexNet

两种网络的对比：



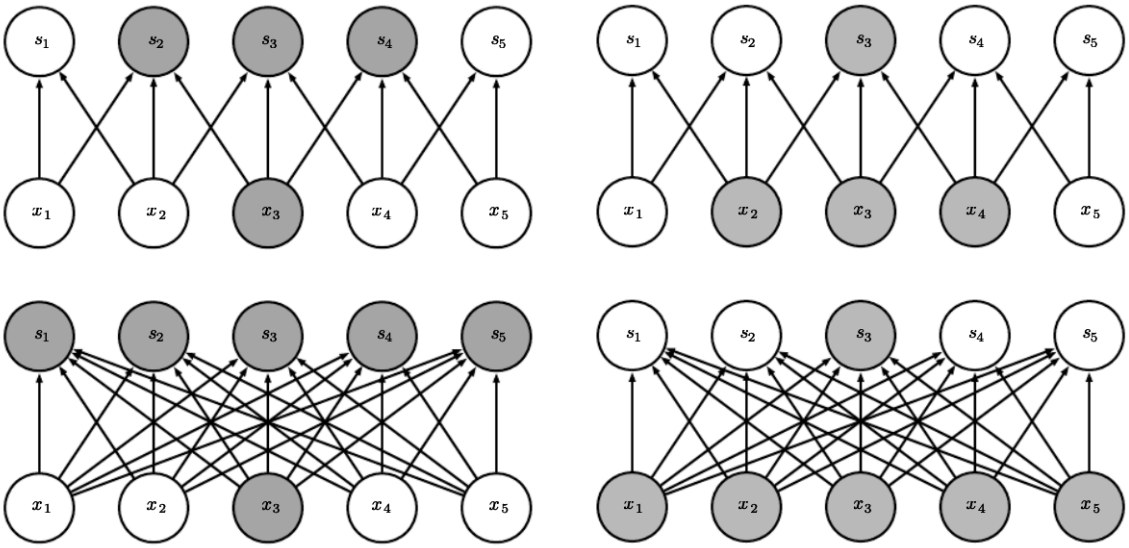
通过数据我们可以看出，在训练初期，由于LeNet的网络深度较低、且参数较少，训练速度明显快于AlexNet；在训练中期，AlexNet网络的波动幅度，也明显大于LeNet，可能的原因是：LeNet已经接近收敛，但是AlexNet由于深度较深且参数较多，仍未很好收敛；但是随着训练次数的增加，更深的网络的优势逐步体现：AlexNet的正确率会略高于LeNet，但是波动也会较大。

通常来说，对于以LeNet为基础发展的卷积神经网络，**网络层数越多，效果越好**。对于本任务来说，可能因为**分类较少**，较深层次的网络的优势**并未完全体现**。

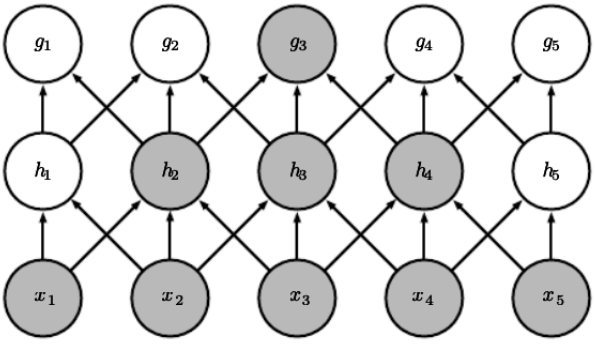
对网络设计的理解

卷积和池化

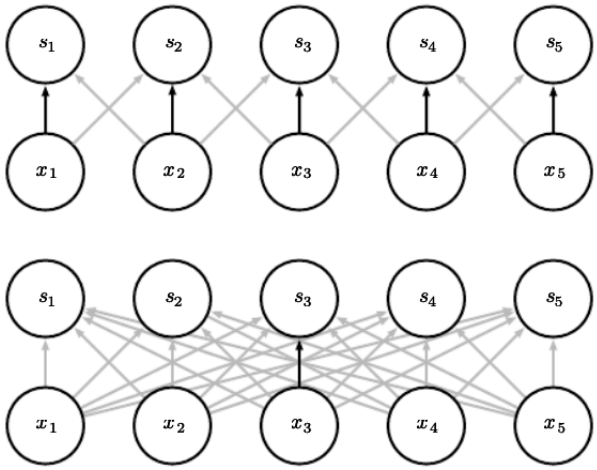
> 参考《Deep Learning》以及《Computer Vision: A Modern Approach》



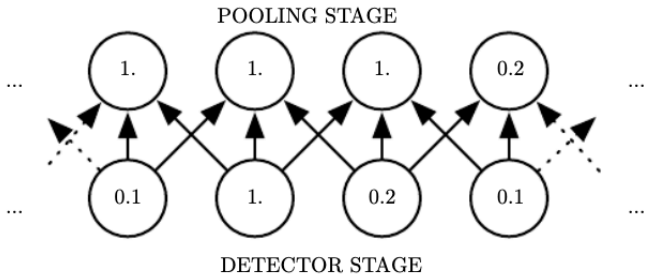
卷积神经网络的**稀疏连接**：每一个输出单元影响k（卷积核大小）个下一层单元；每一个输入单元也只被k个上一层单元影响。

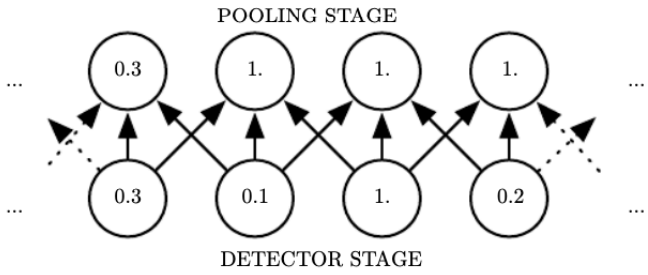


处于卷积神经网络更深的层中的单元，它们的接受域要比处在浅层的单元的接受域更大。如果网络还包含类似步幅卷积或者池化之类的结构特征，这种效应会加强。这意味着在卷积神经网络中尽管直接连接都是很稀疏的，但处在更深的层中的单元可以间接地连接到全部或者大部分输入图像。

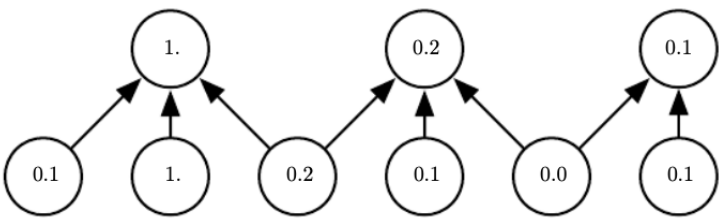


卷积神经网络的**参数共享**：卷积核的参数会被每一个位置使用，即参数被共享。





当输入作出少量平移时，池化能够帮助输入的表示近似**不变**，即当我们对输入进行**少量平移**时，经过池化函数后的大多数输出并**不会发生改变**。

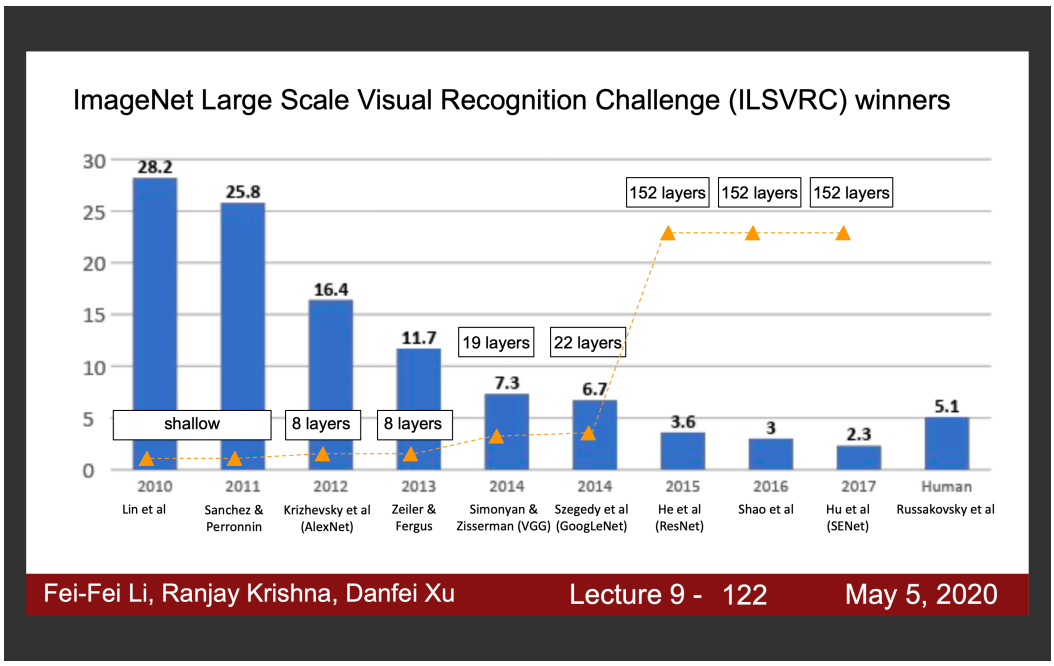


池化**综合**了全部邻居的反馈，这使得池化单元少于探测单元成为了可能——当池化区域为k个像素的时候，下一层就会少k倍的输入，提高了网络的**计算效率**。

综合来看，卷积和池化可以理解为一个具有**无限强的先验**的**全连接网络**：**卷积**，即为这个这个网络的权重必须和它邻居的权重相同，但可以在空间上移动；**池化**，即为每一个单元都具有对少量平移的不变性——然而，卷积神经网络比这样一个具有无限强先验的全连接网络，**计算效率更高**。

卷积神经网络

- 大体上来看，**越深的网络性能越好**，而现在用于分类的网络已经越来越深
- 由于越深的网络**参数越多**，训练**越耗费时间**，所以除了搭建合理的层次，也越来越关注**训练效率**、**网络结构**等问题
- 对于具体的任务，选择构建**合适**的网络，并非越深的网络性能一定越好，比如本次任务量较小的**12分类问题**，使用**1000分类问题**的网络，就不一定能取得满意的效果



总结来说，一定要根据具体的任务选择合适的网络、选择合适的改进方式、选择合适的优化器——并非最新的才是最好的，**最契合**任务的才是**最好**的。