深度学习笔记

mIOU

1. Mean Intersection over Union(MIoU，均交并比)：为语义分割的标准度量。其计算两个集合的交集和并集之比，在语义分割的问题中，这两个集合为真实值（ground truth）和预测值（predicted segmentation）。这个比例可以变形为正真数（intersection）比上真正、假负、假正（并集）之和。在每个类上计算IoU，之后平均。

1. Batch Normalization

1. BN怎么做

输入： 批处理（mini-batch）输入 $\boldsymbol{x}$：$\mathcal{B} = \{x_{1,\dots,m}\}$

输出： 规范化后的网络响应 $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

1: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // 计算批处理数据均值

2: $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ // 计算批处理数据方差

3: $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // 规范化

4: $y_i \leftarrow \gamma\hat{x}_i + \beta = \mathrm{BN}_{\gamma,\beta}(x_i)$ // 尺度变换和偏移

5: **return** 学习的参数 γ 和 β.

- 如上图所示，BN步骤主要分为4步：
- 求每一个训练批次数据的均值
- 求每一个训练批次数据的方差
- 使用求得的均值和方差对该批次的训练数据做归一化，获得0-1分布。其中εε是为了避免除数为0时所使用的微小正数。
- 尺度变换和偏移：将xixi乘以γγ调整数值大小，再加上ββ增加偏移后得到yiyi，这里的γγ是尺度因子，ββ是平移因子。这一步是BN的精髓，由于归一化后的xixi基本会被限制在正态分布下，使得网络的表达能力下降。为解决该问题，我们引入两个新的参数：γγ,ββ。 γγ和ββ是在训练时网络自己学习得到的

---

1. **BN到底解决了什么？**

- 那么为什么要有第4步，不是仅使用减均值除方差操作就能获得目的效果吗？我们思考一个问题，减均值除方差得到的分布是正态分布，我们能否认为正态分布就是最好或最能体现我们训练样本的特征分布呢？不能，比如数据本身就很不对称，或者激活函数未必是对方差为1的数据最好的效果，比如Sigmoid激活函数，在-1~1之间的梯度变化不大，那么非线性变换的作用就不能很好的体现，换言之就是，减均值除方差操作后可能会削弱网络的性能！针对该情况，在前面三步之后加入第4步完成真正的batch normalization。
- BN的本质就是利用优化变一下方差大小和均值位置，使得新的分布更切合数据的真实分布，保证模型的非线性表达能力。BN的极端的情况就是这两个参数等于mini-batch的均值和方差，那么经过batch normalization之后的数据和输入完全一样，当然一般的情况是不同的。

---

1. BN的使用位置

- 在CNN中一般应作用与非线性激活函数之前
- BN在深层神经网络的作用非常明显：若神经网络训练时遇到收敛速度较慢，或者“梯度爆炸”等无法训练的情况发生时都可以尝试用BN来解决。同时，常规使用情况下同样可以加入BN来加速模型训练，甚至提升模型精度

```
mean, variance = tf.nn.moments(x, axes, name=None, keep_dims=False)
# 计算统计矩，mean 是一阶矩即均值，variance 则是二阶中心矩即方差，axes=[0]表示按列计算；
tf.nn.batch_normalization(x, mean, variance, offset, scale, variance_epsilon, name=None)
tf.nn.batch_norm_with_global_normalization(x, mean, variance, beta, gamma, variance_epsilon, scale_after_normalization, name=None);
tf.nn.moments 计算返回的 mean 和 variance 作为 tf.nn.batch_normalization 参数调用；
```

2.GAN

1. 工作原理
    - 我们通常使用两个优化算法来训练GANs。判别器是一个普通的神经网络分类器，训练的过程中，我们使用辨别器 (discriminator) 学习引导生成器。

2. 判别器：
    - 在训练的过程中，我们向辨别器discriminator输入的数据一半来自于真实的训练数据，另一半来自于生成器生成的假图像。在训练的过程中，对于真实数据，判别器尝试向其分配一个接近1的概率（为更好泛化，一般会使用smooth参数将labels设为略小于1的值，如0.9）；而对于生成器生成的‘赝品’，判别器尝试向其分配一个接近0的概率。 也就是说，对于真实数据，我们使用label=1计算代价函数来训练判别器，其代价函数的计算方法为：

```
    d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_lo
gits(logits=d_logits_real, labels=tf.ones_like(d_logits_real) * (1
- smooth)))
```

```
-    对于生成器，我们使用label=0计算代价函数来训练判别器，其代价函数的计算方法为：
```

```
     d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_wit
h_logits(logits=d_logits_fake, labels=tf.zeros_like(d_logits_fak
e)))
所以判别器的代价函数为：
d_loss = d_loss_real + d_loss_fake
```

3. 生成器：
    - 与此同时，生成器尝试做相反的事情，它经训练尝试输出能使辨别器分配接近概率1的样本。生成器的代价函数为

```
  g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(lo
gits=d_logits_fake, labels=tf.ones_like(d_logits_fake)))
```

```
-    随着以上训练的进行，判别器‘被迫’增强自身的判别能力，而生成器‘被迫’生成越来越逼真的输出，以欺骗判别器。理论上，最终生成器和判别器会达到一种均衡“纳什均衡”。
```

Discriminator和Generator损失计算

GANs和很多其他模型不同，GANs在训练时需要同时运行两个优化算法，我们需要为discriminator和generator分别定义一个优化器，一个用来来最小化discriminator的损失，另一个用来最小化generator的损失。即**loss = d_loss + g_loss**

d_loss计算方法：

对于辨别器discriminator，其损失等于真实图片和生成图片的损失之和，即 **d_loss = d_loss_real + d_loss_fake** , losses 均由交叉熵计算而得。在 tensorflow 中可使用以下函数：

tf.nn.sigmoid_cross_entropy_with_logits(logits=logits, labels=labels)

在计算真实数据产生的损失d_loss_real时，我们希望辨别器discriminator输出1；而在计算生成器生成的‘假’数据所产生的损失d_loss_fake时，我们希望discriminator输出0.

因此，对于真实数据，在计算其损失时，将上式中的labels全部都设为1，因为它们都是真实的。为了是增强辨别器discriminator的泛化能力，可以将labels设为0.9，而不是1.0。

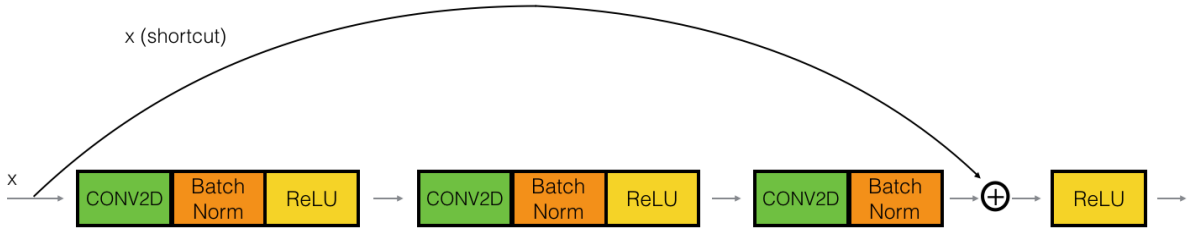
对于生成器生成的‘假’数据，在计算其损失d_loss_fake时，将上式中的labels全部设为0。

g_loss计算方法：

最后，生成器generator的损失用 '假' 数据的logits（即d_logits_fake），但是，现在所有的labels全部设为1（即我们希望生成器generator输出1）。这样，通过训练，生成器generator试图‘骗过’辨别器discriminator。

ResNet50

1. Indentity Block



First component of main path:

- The first CONV2D has F_1 filters of shape (1,1) and a stride of (1,1). Its padding is "valid" and its name should be `conv_name_base + '2a'`. Use 0 as the seed for the random initialization.
- The first BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2a'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Second component of main path:

- The second CONV2D has F_2 filters of shape (f,f) and a stride of (1,1). Its padding is "same" and its name should be `conv_name_base + '2b'`. Use 0 as the seed for the random initialization.
- The second BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2b'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

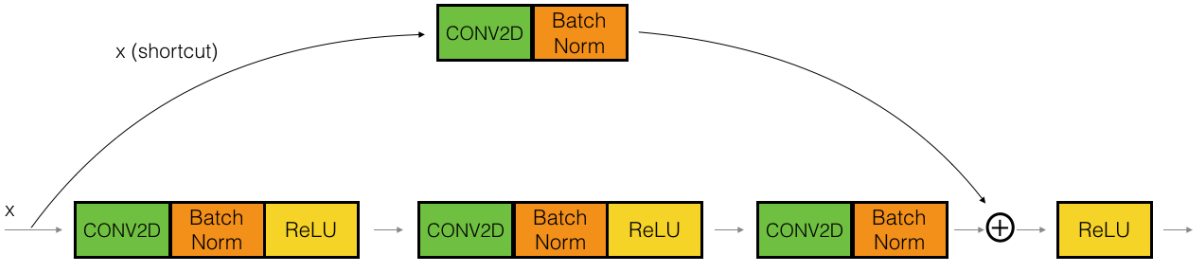
Third component of main path:

- The third CONV2D has F_3 filters of shape (1,1) and a stride of (1,1). Its padding is "valid" and its name should be `conv_name_base + '2c'`. Use 0 as the seed for the random initialization.
- The third BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2c'`. Note that there is no ReLU activation function in this component.

Final step:

- The shortcut and the input are added together.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

2. convolutional block



First component of main path:

- The first CONV2D has F_1 filters of shape (1,1) and a stride of (s,s). Its padding is "valid" and its name should be `conv_name_base + '2a'`.
- The first BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2a'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Second component of main path:

- The second CONV2D has F_2 filters of (f,f) and a stride of (1,1). Its padding is "same" and it's name should be `conv_name_base + '2b'`.
- The second BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2b'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Third component of main path:

- The third CONV2D has F_3 filters of (1,1) and a stride of (1,1). Its padding is "valid" and it's name should be `conv_name_base + '2c'`.
- The third BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2c'`. Note that there is no ReLU activation function in this component.

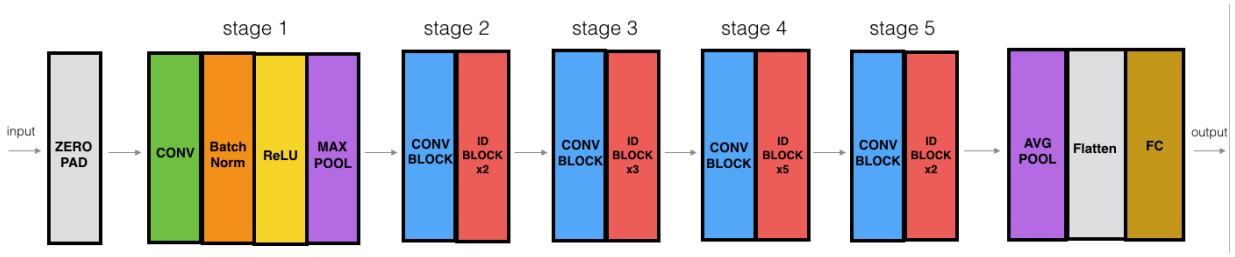
Final step:

- The shortcut and the main path values are added together.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Shortcut path:

- The CONV2D has F_3 filters of shape (1,1) and a stride of (s,s). Its padding is "valid" and its name should be `conv_name_base + '1'`.
- The BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '1'`.

3. ResNet50



- Zero-padding pads the input with a pad of (3,3)
- Stage 1:
  - The 2D Convolution has 64 filters of shape (7,7) and uses a stride of (2,2). Its name is "conv1".
  - BatchNorm is applied to the channels axis of the input.
  - MaxPooling uses a (3,3) window and a (2,2) stride.
  - MaxPooling uses a (3,3) window and a (2,2) stride.
- Stage 2:
  - The convolutional block uses three set of filters of size [64,64,256], "f" is 3, "s" is 1 and the block is "a".
  - The 2 identity blocks use three set of filters of size [64,64,256], "f" is 3 and the blocks are "b" and "c".
- Stage 3:
  - The convolutional block uses three set of filters of size [128,128,512], "f" is 3, "s" is 2 and the block is "a".
  - The 3 identity blocks use three set of filters of size [128,128,512], "f" is 3 and the blocks are "b", "c" and "d".
  - are "b", "c" and "d".
- Stage 4:
  - The convolutional block uses three set of filters of size [256, 256, 1024], "f" is 3, "s" is 2 and the block is "a".
  - The 5 identity blocks use three set of filters of size [256, 256, 1024], "f" is 3 and the blocks are "b", "c", "d", "e" and "f".
- Stage 5:
  - The convolutional block uses three set of filters of size [512, 512, 2048], "f" is 3, "s" is 2 and the block is "a".
  - The 2 identity blocks use three set of filters of size [256, 256, 2048], "f" is 3 and the blocks are "b" and "c".
  - are "b" and "c".
- The 2D Average Pooling uses a window of shape (2,2) and its name is "avg_pool".
- The flatten doesn't have any hyperparameters or name.
- The Fully Connected (Dense) layer reduces its input to the number of classes using a softmax activation. Its name should be `'fc' + str(classes)`.

VAE

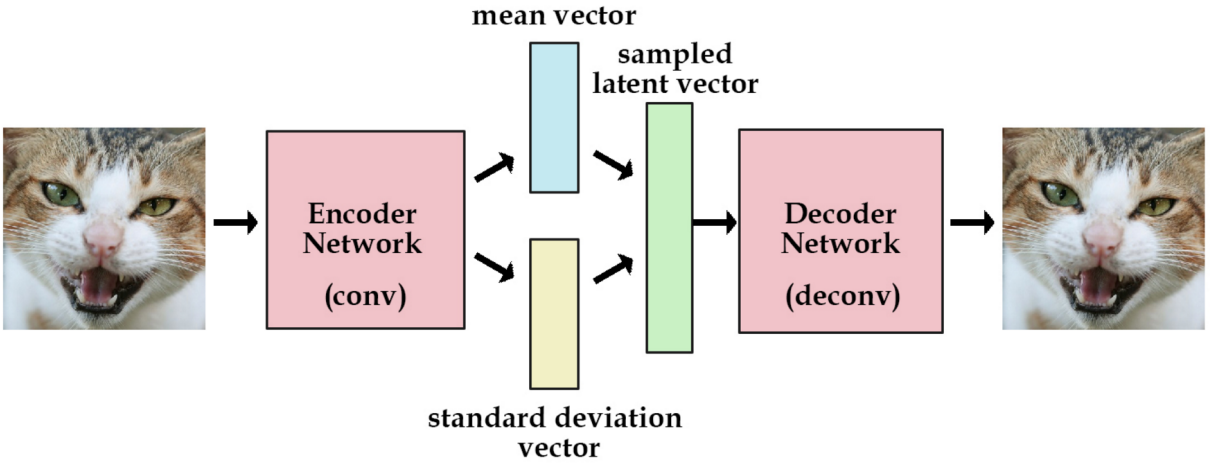
1. 建一个产生式模型，而不是一个只是储存图片的网络。现在我们还不能产生任何未知的东西，因为我们不能随意产生合理的潜在变量。因为合理的潜在变量都是编码器从

原始图片中产生的。这里有个简单的解决办法。我们可以对编码器添加约束，就是强迫它产生服从单位高斯分布的潜在变量。正式这种约束，把VAE和标准自编码器给区分开来了,我们只要从单位高斯分布中进行采样，然后把它传给解码器就可以了。

2. 一方面，是图片的重构误差，我们可以用平均平方误差来度量，另一方面。我们可以用KL散度（KL散度介绍）来度量我们潜在变量的分布和单位高斯分布的差异。

```
generation_loss = mean(square(generated_image - real_image))
latent_loss = KL-Divergence(latent_variable, unit_gaussian)
loss = generation_loss + latent_loss
```

3. 为了优化KL散度，我们需要应用一个简单的参数重构技巧：不像标准自编码器那样产生实数值向量，VAE的编码器会产生两个向量:一个是均值向量，一个是标准差向量



我们可以这样来计算KL散度：

```
# z_mean and z_stddev are two vectors generated by encoder network

latent_loss = 0.5 * tf.reduce_sum(tf.square(z_mean) + tf.square(z_stddev) - tf.log(tf.square(z_stddev))
- 1,1)
```

4. 当我们计算解码器的loss时，我们就可以从标准差向量中采样，然后加到我们的均值向量上，就得到了编码去需要的潜在变量.果给的方差越大，那么这个平均值向量所携带的可用信息就越少。编码越有效，那么标准差向量就越能趋近于标准高斯分布的单位标准差。这种约束迫使编码器更加高效，并能够产生信息丰富的潜在变量。这也提高了产生图片的性能。而且我们的潜变量不仅可以随机产生，也能从未经过训练的图片输入编码器后产生.

```
samples = tf.random_normal([batchsize,n_z],0,1,dtype=tf.float32)
sampled_z = z_mean + (z_stddev * samples)
```

KL散度

1. 相对熵描述两个概率分布P和Q差异的一种方法。它是非对称的，这意味着D(P||Q) ≠ D(Q||P)。特别的，在信息论中，D(P||Q)表示当用概率分布Q来拟合真实分布P时，产生的信息损耗，其中P表示真实分布，Q表示P的拟合分布。

相对熵（relative entropy）又称为KL散度（**Kullback–Leibler divergence**，简称**KLD**），信息散度（information divergence）。

设 $P(x)$ 和 $Q(x)$ 是 X 取值的两个离散概率分布，则 P 对 Q 的相对熵为：

$$D(P||Q) = \sum(P(x)log(P(x)/Q(x)))$$

对于连续的随机变量，定义为： $D(P||Q) = \int P(x)log(P(x)/Q(x))d(x)$

相对熵是两个概率分布 P 和 Q 差别的**非对称性**的度量。 [1]

信息熵，是随机变量或整个系统的不确定性。熵越大，随机变量或系统的不确定性就越大。

相对熵，用来衡量两个取值为正的函数或概率分布之间的差异。

交叉熵，用来衡量在给定的真实分布下，使用非真实分布所指定的策略消除系统的不确定性所需要付出的努力的大小。

相对熵=交叉熵-信息熵： $D(P||Q) = -H(P) + H(P,Q)$

$$KL(p||q) = \sum p(x)log\frac{p(x)}{q(x)}$$

2. KL散度

KL散度的实战——1维高斯分布

我们先来一个相对简单的例子。假设我们有两个随机变量x1,x2，各自服从一个高斯分布

$$N_1(\mu_1, \sigma_1^2), N_2(\mu_2, \sigma_2^2)$$

，那么这两个分布的KL散度该怎么计算呢？

我们知道

$$N(\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

那么KL(p1,p2)就等于

$$\int p_1(x) log \frac{p_1(x)}{p_2(x)} dx$$

$$= \int p_1(x)(log p_1(x) dx - log p_2(x)) dx$$

$$= \int p_1(x) * (log \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{\frac{(x-\mu_1)^2}{2\sigma_1^2}} - log \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{\frac{(x-\mu_2)^2}{2\sigma_2^2}}) dx$$

$$= \int p_1(x) * (-\frac{1}{2} log 2\pi - log\sigma_1 - \frac{(x-\mu_1)^2}{2\sigma_1^2} + \frac{1}{2} log 2\pi + log\sigma_2 + \frac{(x-\mu_2)^2}{2\sigma_2^2}) dx$$

$$= log \frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2} [\int (x-\mu_1)^2 p_1(x) dx + (\mu_1 - \mu_2)^2] - \frac{1}{2}$$

$$= log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}$$

$$\mu_2 = 0, \sigma_2^2 = 1$$

那么N1长成什么样子能够让KL散度尽可能地小呢？

也就是说

$$KL(\mu_1, \sigma_1) = -log\sigma_1 + \frac{\sigma_1^2 + \mu_1^2}{2} - \frac{1}{2}$$

。

我们用“肉眼”看一下就能猜测到当

$$\mu_1 = 0, \sigma_1 = 1$$

1*1 卷积

1. 1*1卷积过滤器 和正常的过滤器一样，唯一不同的是它的大小是1*1，没有考虑在前一层局部信息之间的关系。最早出现在 Network In Network的论文中 ，使用1*1卷积是想加深加宽网络结构 ，在Inception网络（ Going Deeper with Convolutions ）中用来降维。

由于3*3卷积或者5*5卷积在几百个filter的卷积层上做卷积操作时相当耗时，所以1*1卷积在3*3卷积或者5*5卷积计算之前先降低维度。

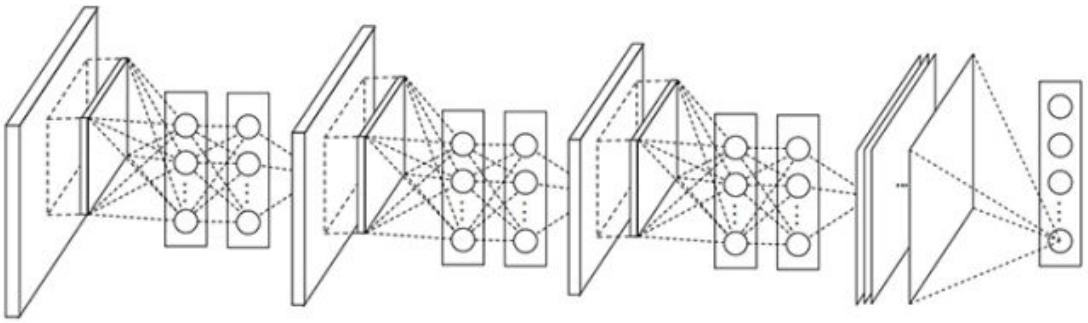
那么，1*1卷积的主要作用有以下几点：

1、**降维（ dimension reductionality ）**。比如，一张500 * 500且厚度depth为100 的图片在20个filter上做1*1的卷积，那么结果的大小为500*500*20。

2、**加入非线性**。卷积层之后经过激励层，1*1的卷积在前一层的学习表示上添加了非线性激励（ non-linear activation ），提升网络的表达能力；

NIN

1. 在卷积后面再跟一个1x1的卷积核对图像进行卷积，这就是Network-in-network的核心思想了。NiN在每次卷积完之后使用，目的是为了在进入下一层的时候合并更多的特征参数。同样NiN层也是违背LeNet的设计原则（浅层网络使用大的卷积核），但却有效地合并卷积特征，减少网络参数、同样的内存可以存储更大的网络
2. 这个“网络的网络”（NIN）能够提高CNN的局部感知区域。例如没有NiN的当前卷积是这样的：3x3 256 [conv] -> [maxpooling]，当增加了NiN之后的卷积是这样的：3x3 256 [conv] -> 1x1 256 [conv] -> [maxpooling]。



MLP在CNN基础上的进步在于：

1）将feature map由多通道的线性组合变为非线性组合，提高特征抽象能力；

2）通过1x1卷积核及Avg-pooling代替fully connected layers实现减小参数。

事实上，CNN里的卷积基本上是多通道的feature map（通道数与其Filter个数相同）和多通道的卷积核做操作，如果使用1x1的卷积核，得到的值就与周边的像素点无关了，则这个操作实现的就是多个feature map的线性组合，实现feature map在通道个数上的变化。多个1x1的卷积核级联就可以实现对多通道的feature map做非线性的组合，再配合激活函数，就可以实现MLP结构，同时通过1x1的卷积核操作还可以实现卷积核通道数的降维和升维，实现参数的减小化。

通过MLP微结构，实现了不同filter得到的不同feature map之间的整合，可以使网络学习到复杂和有用的跨特征图特征。MLP中的每一层相当于一个卷积核为1x1的卷积层。

（2）Global Average Pooling

利用全局均值池化来替代原来的全连接层（fully connected layers），即对每个特征图一整张图片进行全局均值池化，则每张特征图得到一个输出。例如CIFAR-100分类任务，直接将最后一层MLPconv输出通道设为100，对每个feature map（共100个）进行平局池化得到100维的输出向量，对应100种图像分类。这样采用均值池化，去除了构建全连接层的大量参数，大大减小网络规模，有效避免过拟合。

最后，总结下NIN的主要优点：

1）更好的局部抽象能力；

2）更小的参数空间；

3）更小的全局Over-fitting。

RCNN

1. RCNN与 Fast RCNN
    - 原来的方法：许多候选框（比如两千个）–>CNN–>得到每个候选框的特征–>分类+回归
      现在的方法：一张完整图片–>CNN–>得到每张候选框的特征–>分类+回归
        - 所以容易看见，Fast RCNN相对于RCNN的提速原因就在于：不过不像RCNN把每个候选区域给深度网络提特征，而是整张图提一次特征，再把候选框映射到conv5上，而SPP只需要计算一次特征，剩下的只需要在conv5层上操作就可以了。
2. 各大算法的步骤

RCNN

1. 在图像中确定约1000-2000个候选框 (使用选择性搜索)
2. 每个候选框内图像块缩放至相同大小，并输入到CNN内进行特征提取
3. 对候选框中提取出的特征，使用分类器判别是否属于一个特定类
4. 对于属于某一特征的候选框，用回归器进一步调整其位置

Fast RCNN

1. 在图像中确定约1000-2000个候选框 (使用选择性搜索)
2. 对整张图片输进CNN，得到feature map
3. 找到每个候选框在feature map上的映射patch，将此patch作为每个候选框的卷积特征输入到SPP layer和之后的层
4. 对候选框中提取出的特征，使用分类器判别是否属于一个特定类
5. 对于属于某一特征的候选框，用回归器进一步调整其位置

Faster RCNN

1. 对整张图片输进CNN，得到feature map
2. 卷积特征输入到RPN，得到候选框的特征信息
3. 对候选框中提取出的特征，使用分类器判别是否属于一个特定类
4. 对于属于某一特征的候选框，用回归器进一步调整其位置