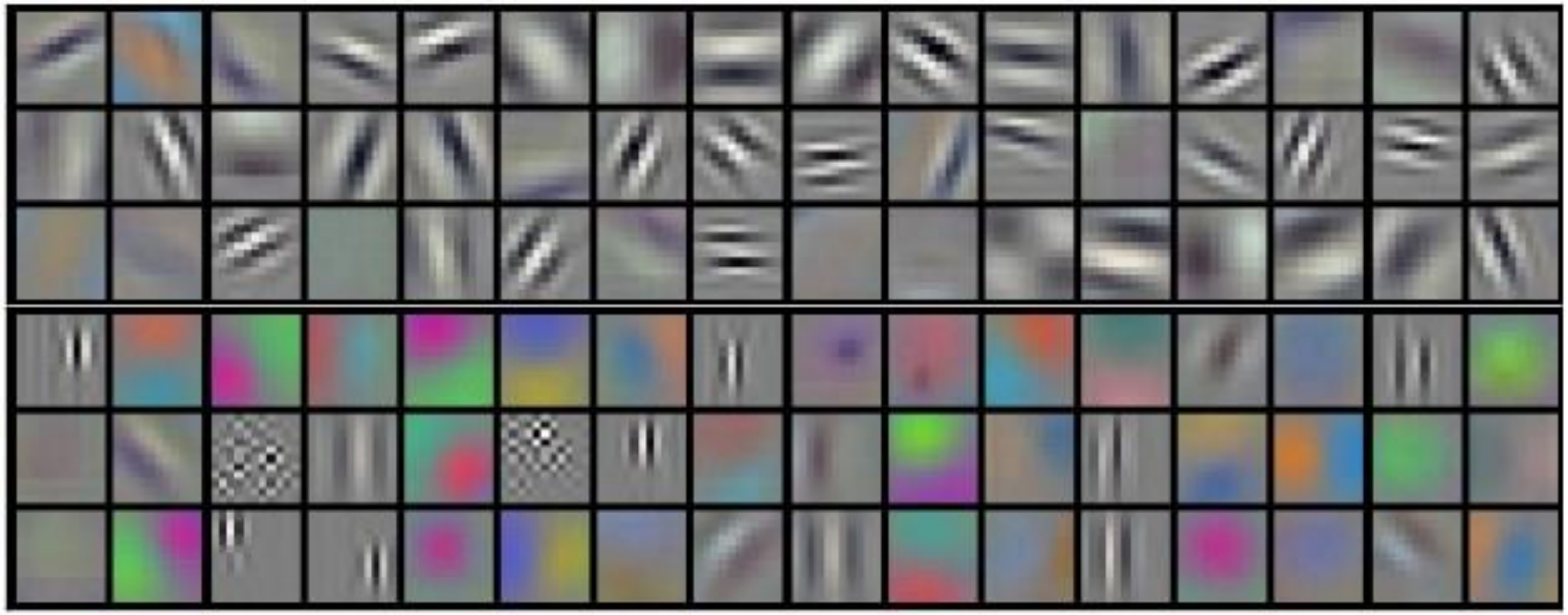
CNN 可视化：滤波器

- AlexNet中的滤波器（96 filters [11x11x3]）

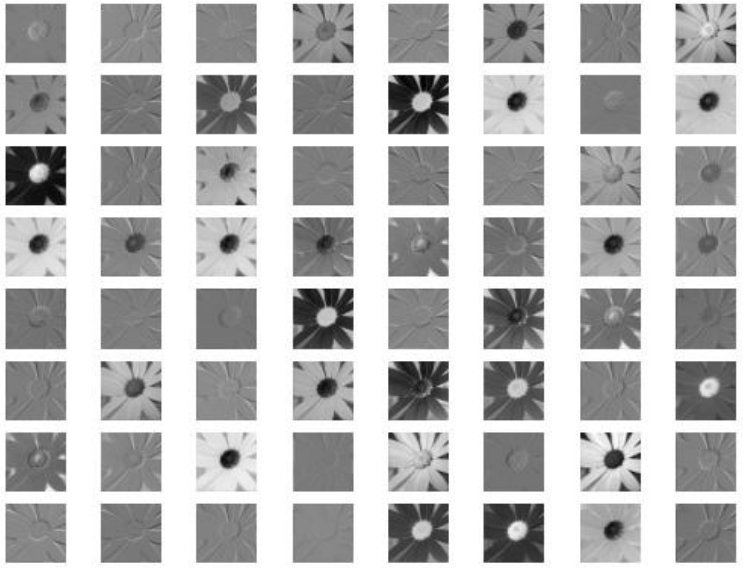


Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

理解和可视化神经网络

这一节将展示如何可视化卷积神经网络的每一层输出的特征图，这里选取的卷积神经网络是pytorch框架提供的VGG16模型。

网络的输入以左图的花为例，在经过VGG16的第一层卷积层之后，就会有64个特征图：



理解和可视化神经网络

首先是获取到VGG16的模型，以及一些路径的初始化。

```
# 使用pytorch官方提供的VGG16模型
# 如果是第一次使用，就将pretrained设置为True，然后就会在当前目录下下载pytorch的VGG16预训练模型的权重
model = models.vgg16(pretrained=False)
print(model)
modelWeightPath = './vgg16.pth'
model.load_state_dict(torch.load(modelWeightPath))
# 设置特征图保存的路径
rootPath = r'./feature_map_save'
```

```
# 对图片进行预处理
def getImageInfo(imageDir):
    imageInfo = Image.open(imageDir).convert('RGB')
    # 数据预处理方法
    image_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
    imageInfo = image_transform(imageInfo)
    imageInfo = imageInfo.unsqueeze(0)
    return imageInfo
```

然后是对输入的图片先进行预处理，使图像能够变成被卷积神经网络所接受的大小。

理解和可视化神经网络

下面是VGG16的整个模型的结构。

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

理解和可视化神经网络

将处理后的图片输入VGG16网络进行前向传播，但是在传播的过程中计算当前的网络层数，直到当前的网络层数等于我们所希望可视化的层数，这样就能获取到我们需要的特征图。

```python
# 得到第k层的特征图
def getKLayerFeatureMap(modelLayer, k, x):
    with torch.no_grad():
        for index, layer in enumerate(modelLayer):
            x = layer(x)
            if k == index:
                return x
```

```python
# 显示特征图
def showFeatureMap(featureMap, k):
    if not os.path.exists(rootPath):
        os.mkdir(rootPath)
    kPath = os.path.join(rootPath, str(k))
    if not os.path.exists(kPath):
        os.mkdir(kPath)
    featureMap = featureMap.squeeze(0)
    featureMapNum = featureMap.shape[0]  # 返回通道数
    row_num = np.ceil(np.sqrt(featureMapNum))  # 将通道数开方取整，尽可能地使行列数相同
    plt.figure()
    for index in range(1, featureMapNum + 1):  # 通过遍历的方式，将每个特征图拿出
        plt.subplot(row_num, row_num, index)
        plt.imshow(featureMap[index - 1], cmap='gray')
        plt.axis('off')
        # 保存特征图到指定的路径下
        scipy.misc.imsave(os.path.join(kPath, str(index) + '.png'), featureMap[index - 1])
    # 将这一层所有的特征图合并显示并保存
    plt.savefig(os.path.join(kPath, 'totImg.png'))
    plt.show()
```

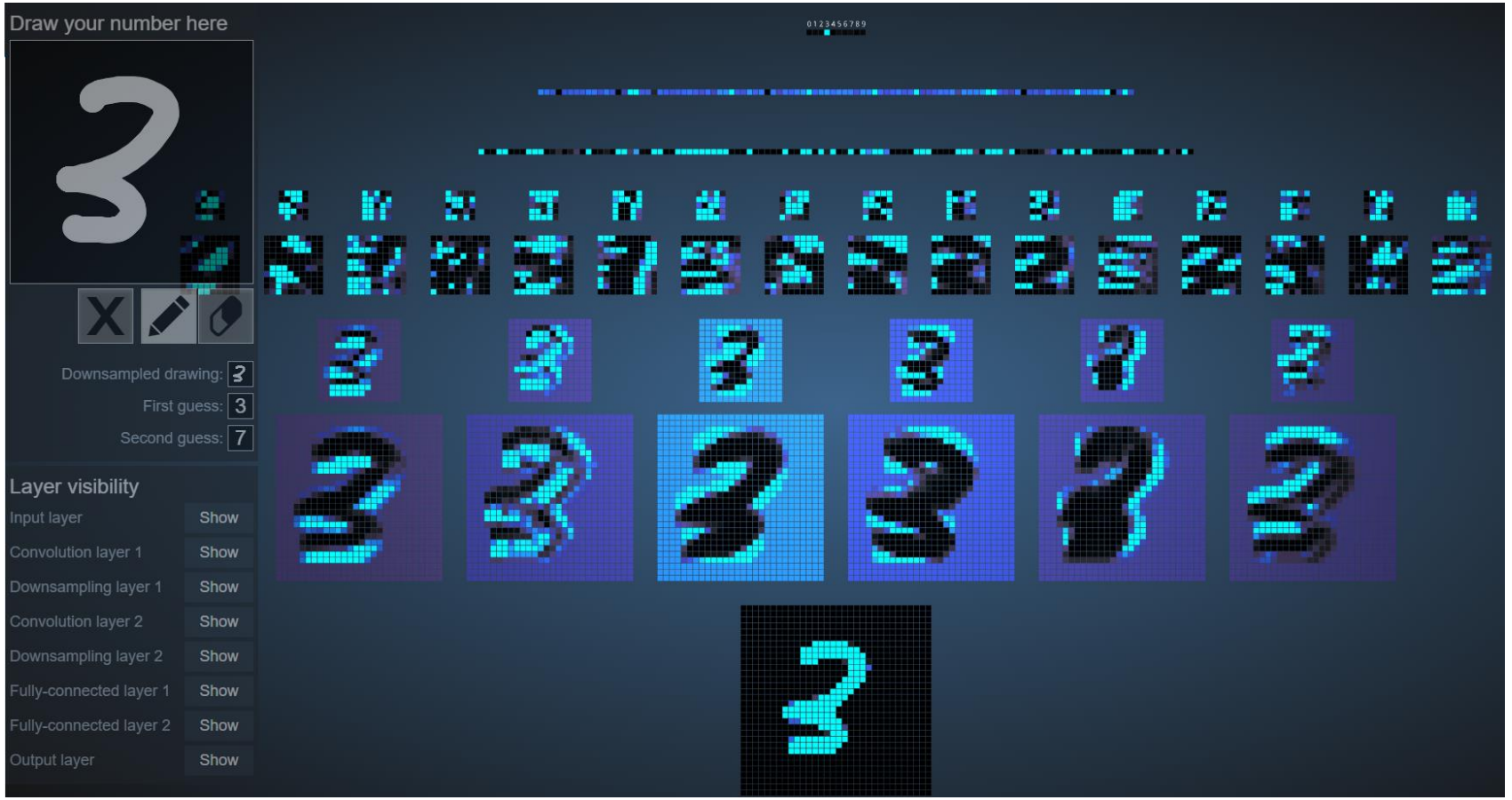
在得到第k层的特征图之后，就要显示特征图，但是由于每一层的特征图可能有多个（第一层卷积层的输出特征图就有64个），所以我们得到某一层的所有特征图后，需要遍历取出。

理解和可视化神经网络

```python
if __name__ == '__main__':

    # 图片的路径
    imageDir = r"E:\code\python\pytorch1_6\data\flower\flower_photos\daisy\43474673_7bb4465a86.jpg"

    # 定义提取第几层的feature map
    k = 0

    # 将图片预处理方便送入网络进行特征提取
    imageInfo = getImageInfo(imageDir)

    # 取出网络的所有层
    modelLayer = list(model.children())
    modelLayer = modelLayer[0]

    # 得到第k层的特征图
    feature_map = getKLayerFeatureMap(modelLayer, k, imageInfo)

    # 将特征图可视化
    showFeatureMap(feature_map, k)
```

如图是所有的流程，可以自定义图片的路径、希望提取的网络层数。

理解和可视化神经网络



这个是LeNet-5识别手写数字体的可视化网站：https://www.cs.ryerson.ca/~aharley/vis/conv/flat.html

神经网络模型的设计与训练

训练预处理

神经网络的主要作用是通过数据训练出一个拟合函数。
神经网络学习到的是**训练数据的分布**。
如果训练数据和测试**数据的分布不同**，那么就很难在测试数据上得到很好的效果。这就是训练预处理的必要性

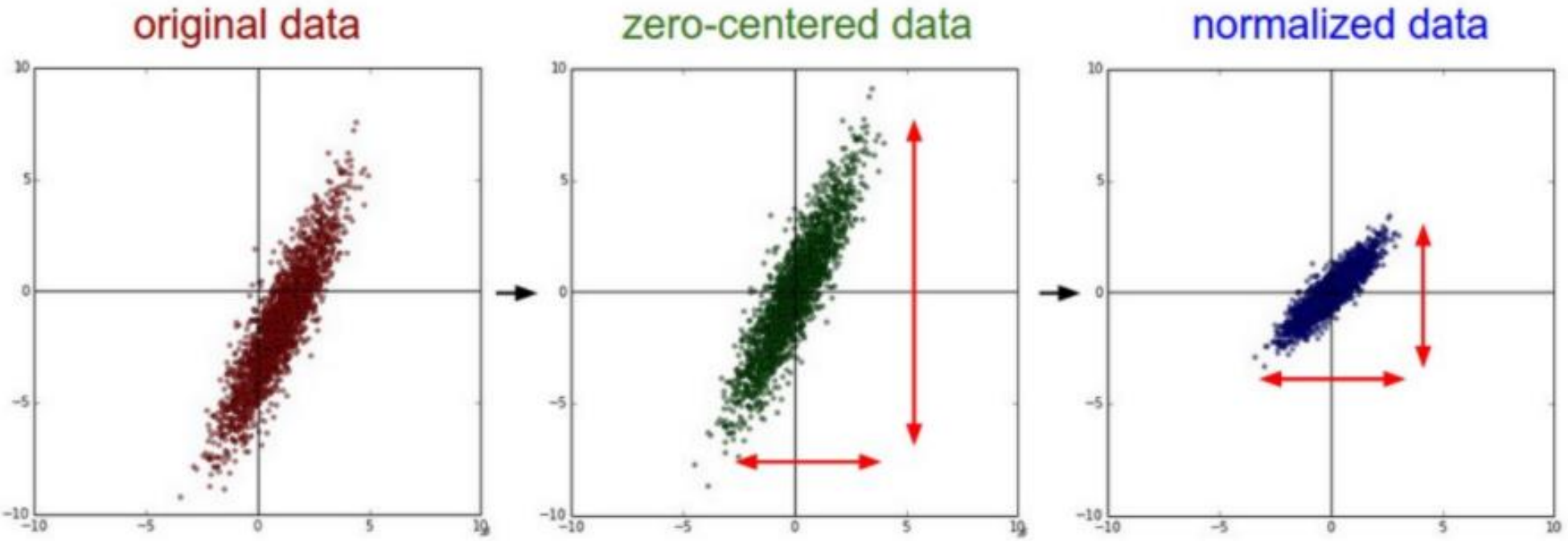
训练预处理

数据预处理的方法通常有三种

0均值 是最常用的预处理方法。就是把数据的每一维减去每一维的均值，这样数据就变成0均值的了

归一化(Normalization) 是指将数据归一化到相同的尺度。通常有两种方法

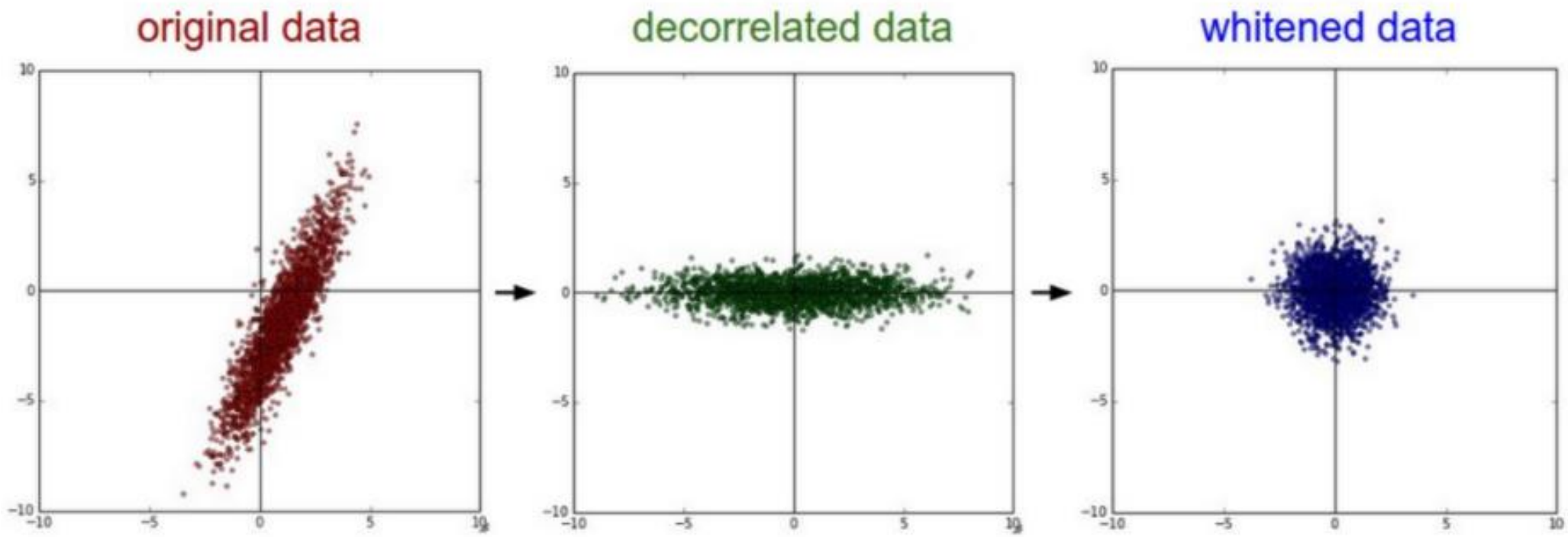
- 0均值后，数据的每一维除每一维的标准差
- 可以让每一维数据除以其绝对值的最大值，令每一维最大值为1,最小值为-1



训练预处理

数据预处理的第三种方法：

PCA和白化 是另一种形式的预处理方法。首先我们将数据变成0均值的，然后计算数据的协方差矩阵来得到数据不同维度之间的相关性



数据增广(Data Augmentation)

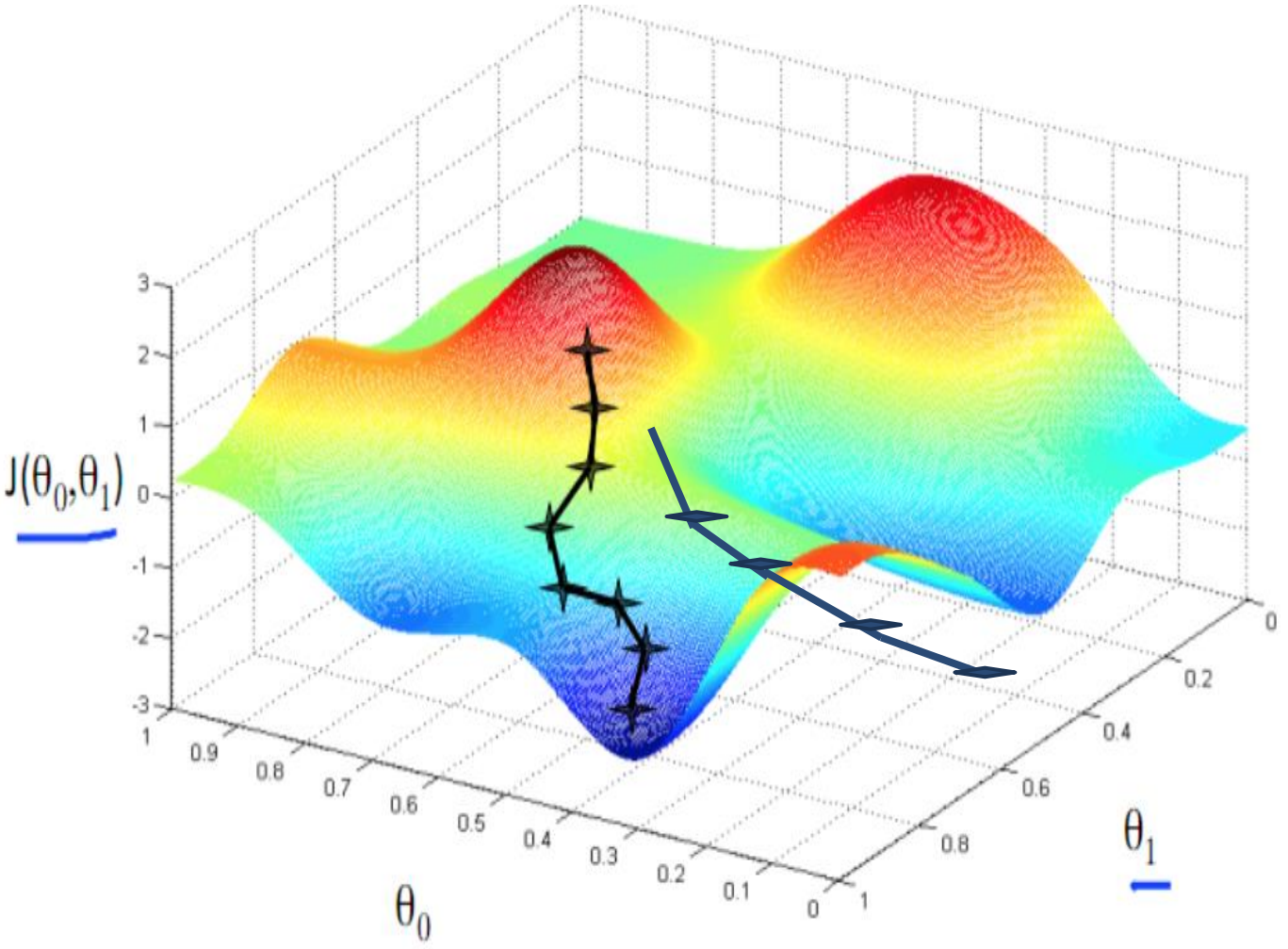
数据增广是深度学习中的一种常见预处理，不同于之前的三种预处理，他的目的是为了增加训练数据集，让数据集尽可能的多样化，使得训练的模型具有更强的泛化能力

常见的数据增广：

- Color Jittering：对颜色的数据增强：图像亮度、饱和度、对比度变化；**彩色变换**
- Random Scale：尺度变换；**随机缩放**
- Random Crop：采用随机图像差值方式，对图像进行裁剪、缩放；尺度和长宽比增强变换；**随机裁剪**
- Horizontal/Vertical Flip：水平/垂直翻转；**翻转**
- Shift：平移变换；
- Rotation/Reflection：旋转/仿射变换；
- Noise：高斯噪声、模糊处理；

权重初始化

- 对收敛的算法适当的初始化能**加快收敛速度**。
- 初始值的选择将影响**模型收敛局部最小值还是全局最小值**，右图，因初始值的不同，导致收敛到不同的极值点
- 初始化也会影响**模型的泛化**
- 初始值过大可能在前向传播或反向传播中产生梯度爆炸
- 如果太小将导致梯度消失



权重初始化

- **零值初始化**

把权重初始化为0，可能出现梯度消失，不常用

- **随机初始化**

将随机数赋值给w，通常倾向于使用很小的随机数，防止落在激活函数的平滑区域导致梯度下降变慢

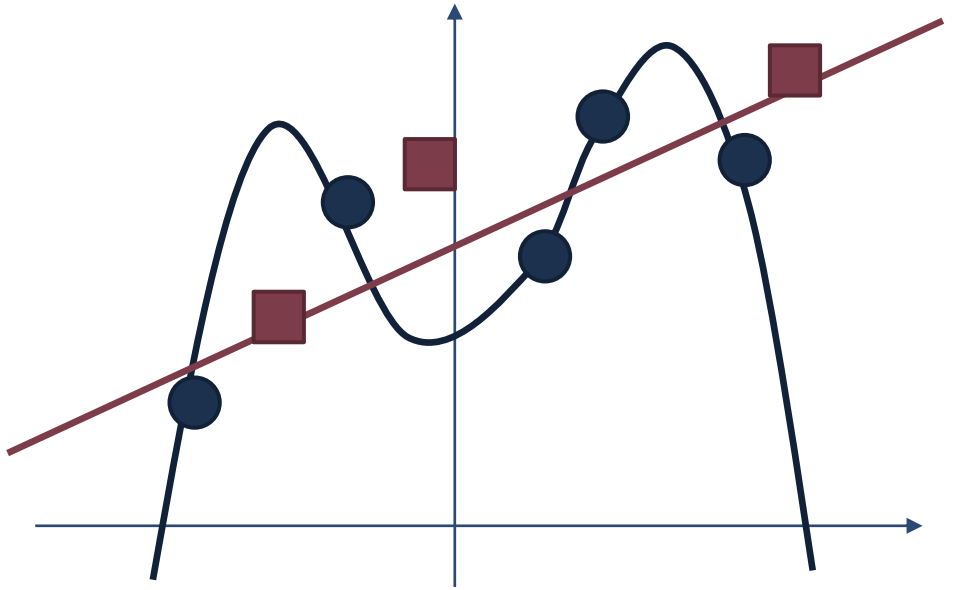
- **均匀分布初始**
- **正态分布初始**
- **正交分布初始**

同样都属于随机初始化，不过随机数分别符合均匀分布，正态分布，正交分布，实践表明以上三种初始化效果更好

正则化-----过拟合和欠拟合

对于训练集，存在多条曲线与有限样本训练集一致

蓝原点：训练集数据
红方点：测试数据
蓝曲线：完美拟合训练集的模型曲线
红直线：较好拟合训练集的模型曲线

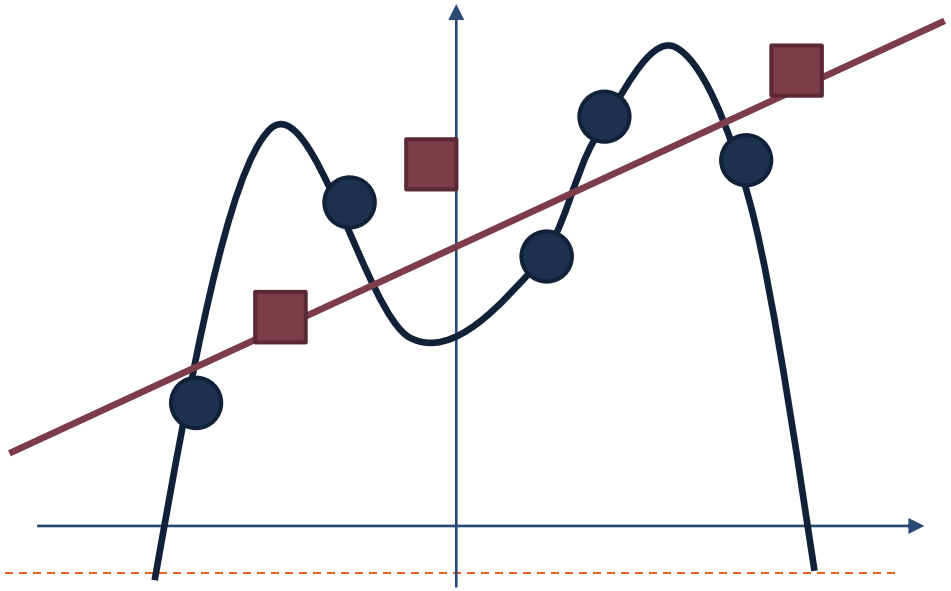


训练是让模型更好的拟合有限样本，但我们更在意的应该是模型对新的测试数据的表现。
对于新的数据点绿方点来说，蓝色曲线所表示的模型**过于贴近训练样本特征**，导致测试样本表现不佳，这就是**过拟合**
相反，未能有效学习训练数据的关系称为**欠拟合**

正则化-----过拟合和欠拟合

- **过拟合时**，通常是因为模型过于复杂，学习器把训练样本学得“太好了”，很可能把一些训练样本自身的特性当成了所有潜在样本的共性了，这样一来模型的泛化性能就下降了。**发生过拟合的根本原因在于训练数据与训练参数比例太小**
- **欠拟合时**，模型又过于简单，学习器没有很好地学到训练样本的一般性质，所以不论在训练数据还是测试数据中表现都很差。

对于欠拟合：增加模型参数
对于过拟合：
- 增加训练数据量
- 减小参数数量
- **正则化**



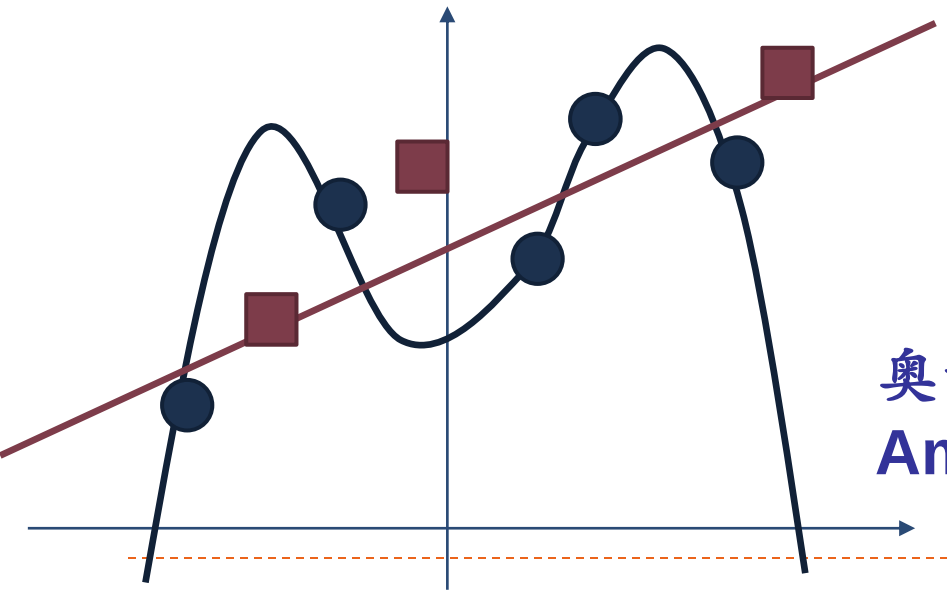
正则化-----过拟合和欠拟合

正则化通常是在目标损失函数中加入正则项，减轻模型复杂度

$$L(w) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)}_{\text{Data Loss}} + \underbrace{\lambda R(w)}_{\text{Regularization}}$$

Data Loss
模型预测值需要和训练样本匹配

Regularization
模型应该尽可能简单，有利于防止过拟合

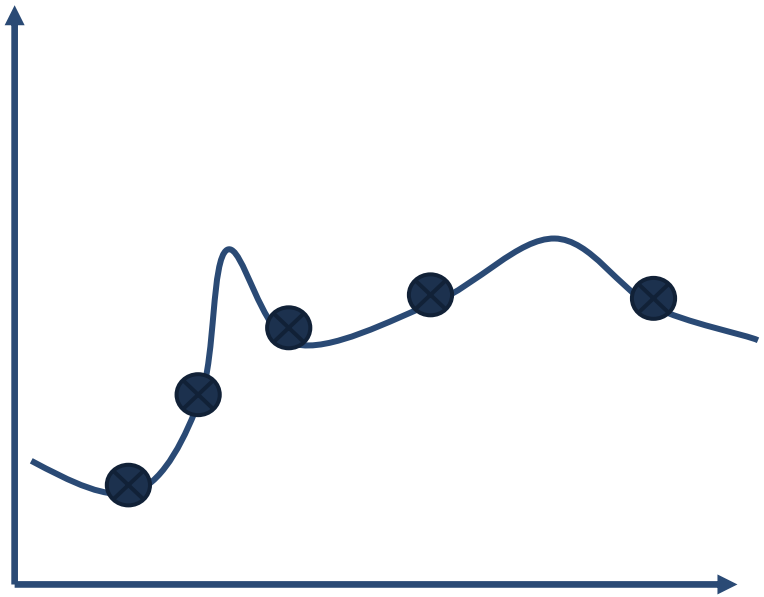


奥卡姆剃刀原理：

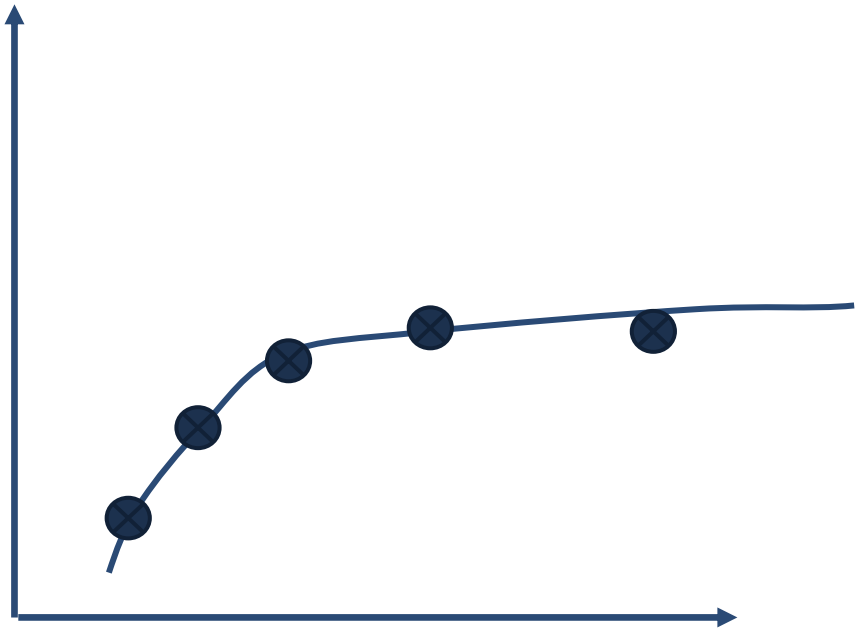
Among competing hypotheses, the simplest is the best

权重正则化

$$L(w) = \frac{1}{4}\sum_{i=1}^{4}(pre_w(x_i) - y_i)^2) \quad L(w) = \frac{1}{4}\sum_{i=1}^{4}(pre_w(x_i) - y_i)^2) + 10000 * w_3^2 + 10000w_4^2$$



$w0 + w1x + w2x^2 + w3x^3 + w4x^4$



$w0 + w1x + w2x^2$

正则化函数

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

最为常见的三个正则化函数如下：

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

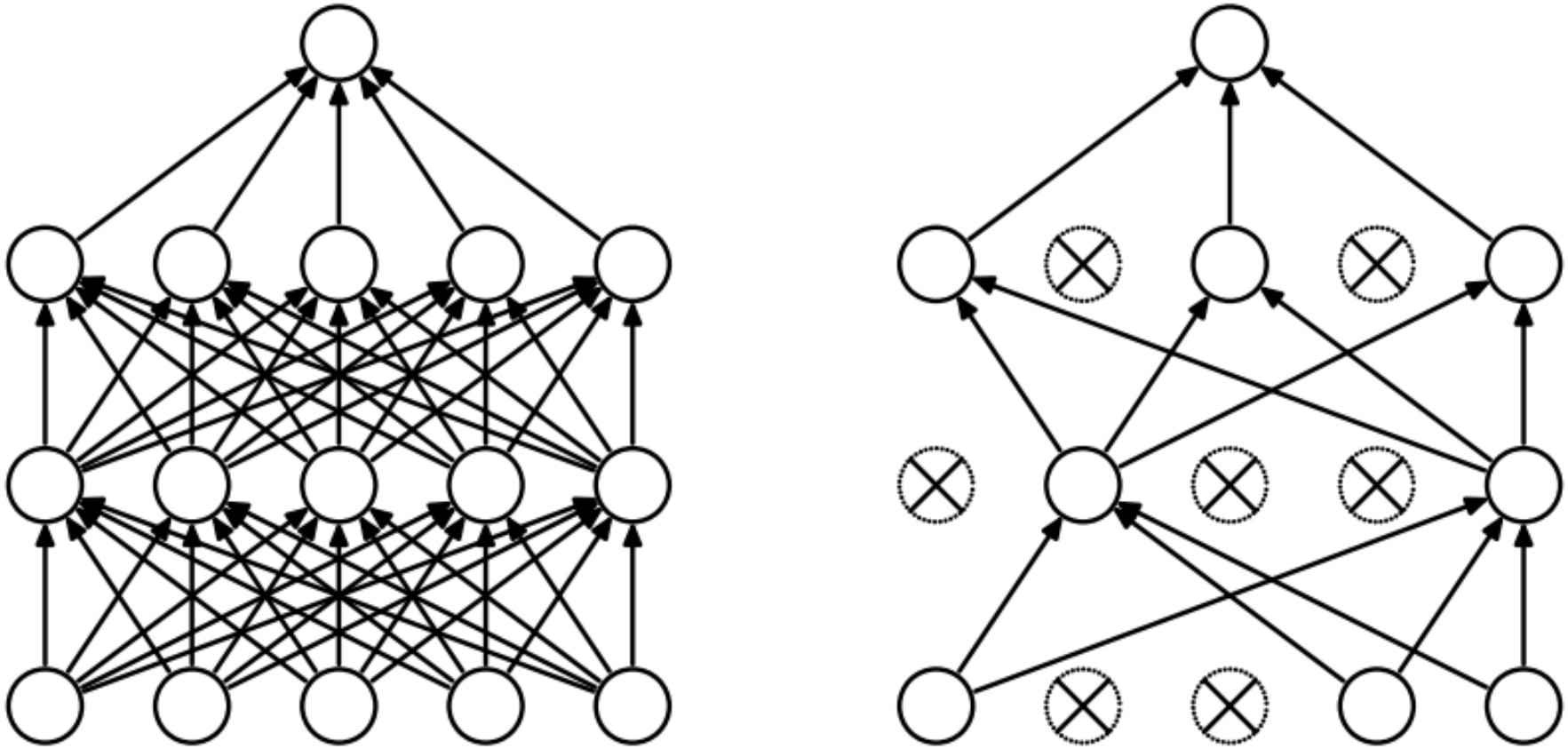
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

- L1正则化可以产生稀疏权值矩阵，即产生一个稀疏模型，可以用于特征选择
- L2正则化可以防止模型过拟合（overfitting）；一定程度上，L1也可以防止过拟合

正则化-----Dropout正则化

Srivastava等人在2014年发表的一篇论文中，提出了一种针对神经网络模型的正则化方法Dropout(A Simple Way to Prevent Neural Networks from Overfitting)

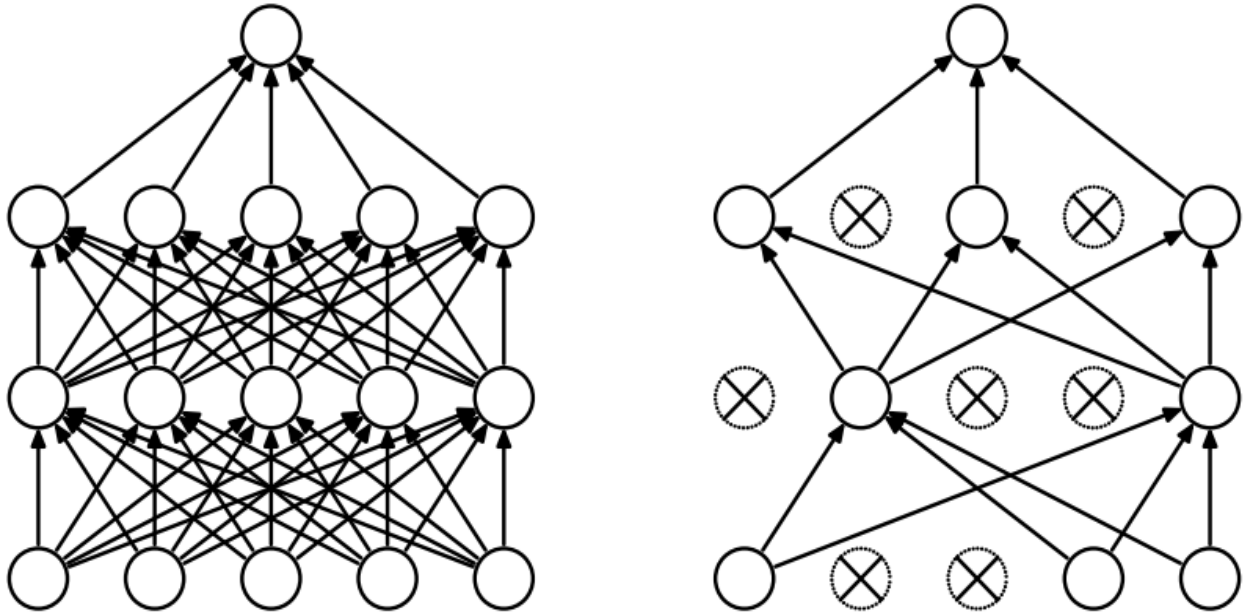


正则化-----Dropout正则化

- 随机删掉网络中**一定比例（可设置）**的隐藏神经元，输入输出神经元保持不变
- 把输入x通过修改后的网络前向传播，然后对修改后的神经元进行参数更新

不断重复这一过程：

- 恢复被删掉的神经元（**此时被删除的神经元保持原样，而没有被删除的神经元已经有所更新**）
- 从隐藏层神经元中随机选择一定比例大小的子集临时删除掉（**备份被删除神经元的参数**）。
- 先前向传播然后通过梯度下降法更新参数（**没有被删除的那一部分参数得到更新，删除的神经元参数保持被删除前的结果**）



。

正则化-----Dropout正则化

Dropout解决拟合的原理：

- **取平均**。每次进行Dropout就会随机舍弃掉一些神经元，剩下的神经元相当于组成一个新的网络，训练过程会产生很多这样的网络。假设相同数据训练5个不同神经网络分别输出5个不同结果，可以采取”5个结果取均值”或者”多数取胜的投票策略”。有利于让一些拟合的网络**相互抵消**
- **减少神经元之间复杂的共适应关系**。Dropout导致两个神经元不一定都在一个dropout出现。权值更新不再依赖有固定关系的隐含结点的共同作用，增加鲁棒性

正则化-----批量正则化Batch Normalization

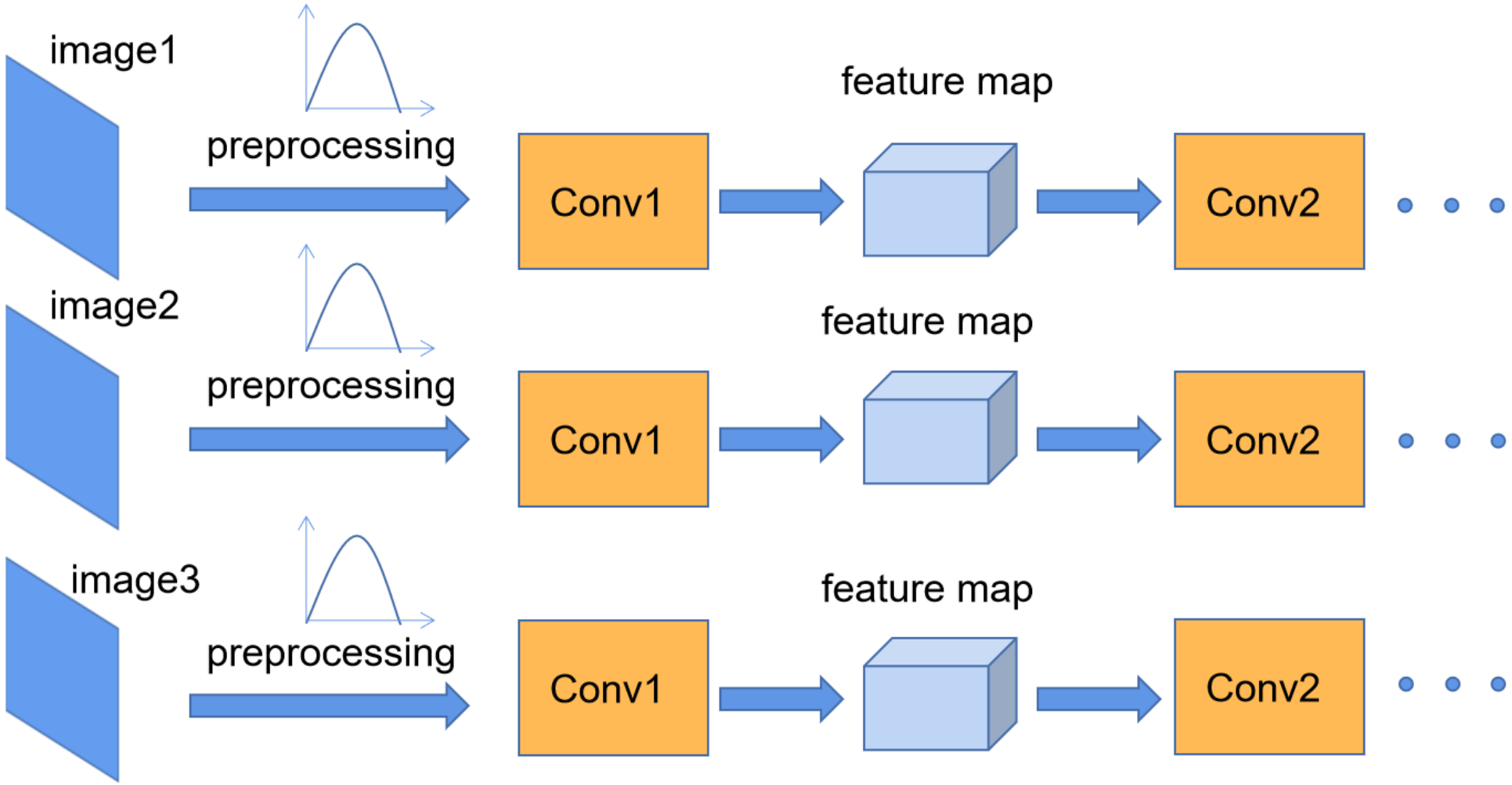
Batch Normalization是google团队在2015年论文《Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift》提出的

提出原因：如果将每一层的输入作为一个分布看待，随着训练更新参数，会导致相同的输入分布得到的输出分布改变了。

神经网络的每一层间，每轮训练时分布都是不一致，那么相对的训练效果就得不到保障，这种问题被称作**Internal Covariate Shift**，简称 ICS.

总而言之，这会导致每个神经元的输入数据不再是”独立同分布”,会有以下问题：
1、上层网络需要不断适应新的输入数据分布，**降低学习速度**
2、下层输入的变化可能趋向于变大或者变小，导致上层落入饱和区，使得**学习过早停止**
3、每层的更新都会影响到其它层，因此每层的参数更新策略需要尽可能的谨慎

正则化-----批量正则化Batch Normalization



正则化-----批量正则化Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

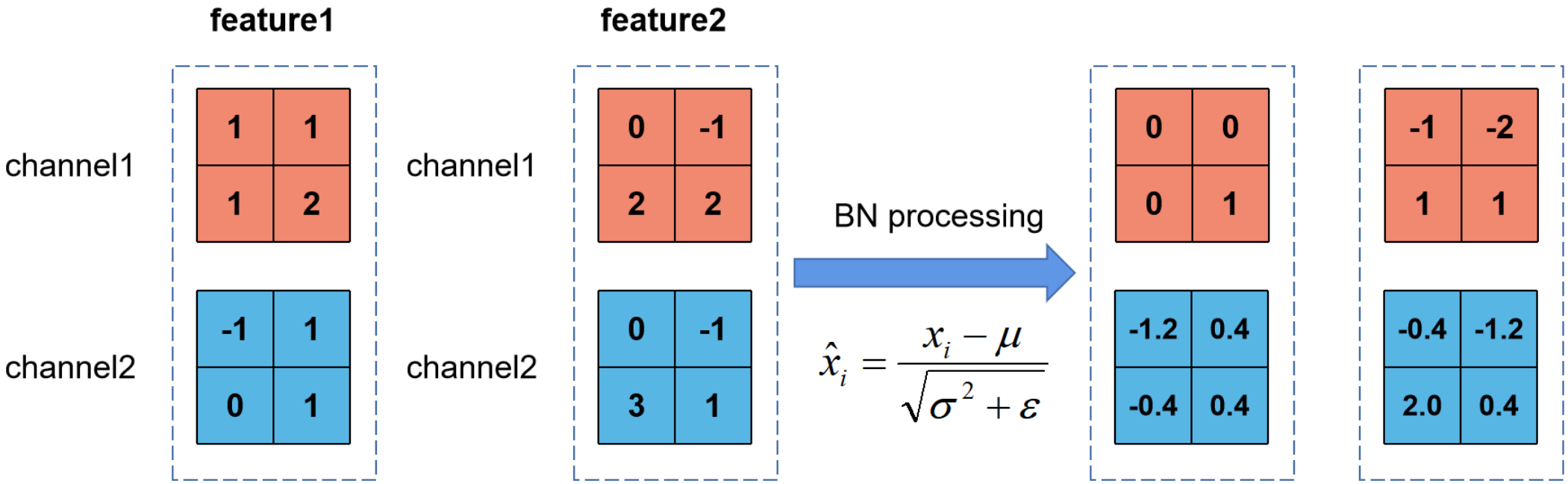
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

本质上就是在每一层输入的时候，又插入了一个**归一化处理**(归一化至:均值0,方差为1)再进入网络的下一层

不过他有γ，β两个参数，分别来**调整数值分布的方差大小**和**调节数值均值的位置**

正则化-----批量正则化Batch Normalization



$$x^{(1)} = \{1,1,1,2,0,-1,2,2\}$$

$$x^{(2)} = \{-1,1,0,1,0,-1,3,1\}$$

$$\mu_1 = \frac{1}{m}\sum_{i=1}^{m} x_i^{(1)} = 1 \qquad \sigma_1^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i^{(1)} - \mu_1)^2 = 1$$

$$\mu_2 = \frac{1}{m}\sum_{i=1}^{m} x_i^{(2)} = 0.5 \qquad \sigma_2^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i^{(2)} - \mu_2)^2 = 1.5$$

$$\mu = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} \qquad \sigma^2 = \begin{bmatrix} 1 \\ 1.5 \end{bmatrix}$$

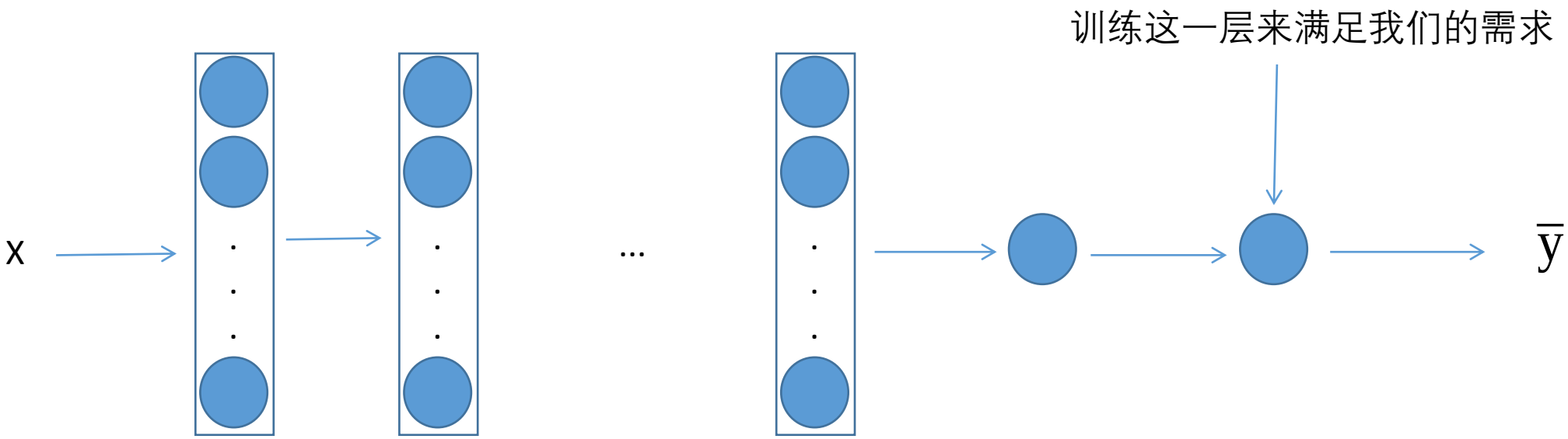
神经网络的迁移学习

在训练神经网络的时候，往往需要大量的数据来对网络的权重进行训练，这意味着我们不仅需要收集大量的数据，还需要对这些数据进行标注，而且还将耗费大量的时间去从头开始训练。

但是如果我们直接使用别人已经训练好的网络结构权重，我们就能很快地看到神经网络的效果。例如有的神经网络权重是在大量的图片上训练好的，这些图片里面可能有各种物品：人、猫、狗、车、花等等。如果我们现在只需要识别花，或者对花进行分类，那么预训练的权重就可能已经能够很好地提取出花的特征了。我们所需要做的就只是修改下网络最后用于分类几个网络层，将其修改为适合我们所需要的分类数即可。

例如，我们需要训练一个花的分类器，花的种类有5种：雏菊、蒲公英、玫瑰、向日葵、郁金香。那么我们只需要下载别人训练好的网络权重，然后将网络的最后几层修改下使其分类的数量为5。然后再冻结前面的网络权重，这样就可以在训练中只训练我们所修改的那几层网络权重，而不会改变前面别人训练好的网络权重。

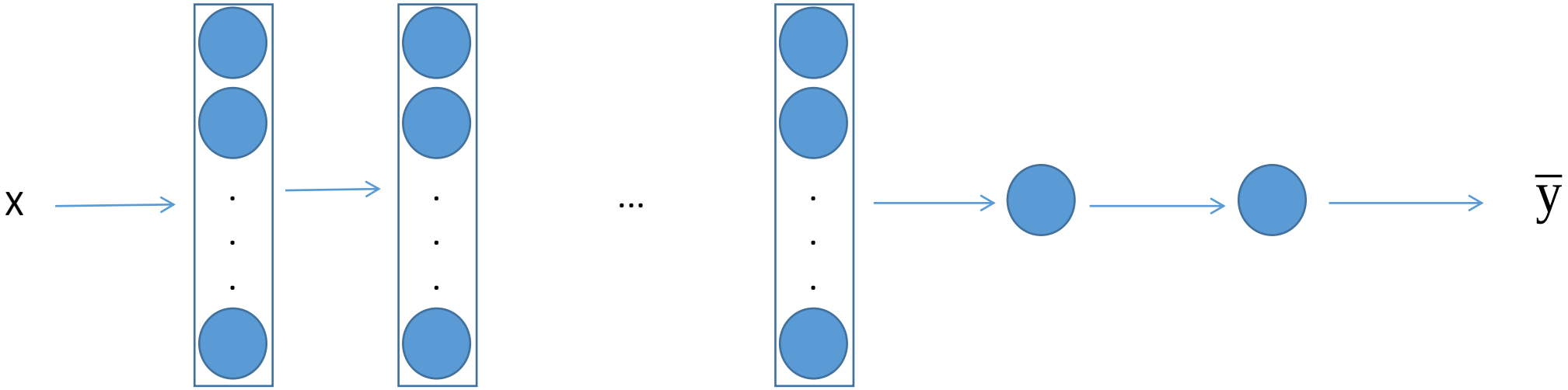
神经网络的迁移学习



冻结这些层的权重，使其在训练中不会被改变

神经网络的迁移学习

训练这些层来满足我们的需求



冻结这些层的权重，使其在训练中不会被改变

当我们已经有了较多的数据之后，可以考虑再多训练几层网络，以提高网络在自己数据集上的分类能力。

神经网络的迁移学习

使用已经在ImageNet上训练好的MobileNetV2模型来迁移到我们的花分类的任务上，花的种类一共有5类：daisy、dandelion、roses、sunflowers、tulips


daisy


dandelion


roses


sunflowers


tulips

神经网络的迁移学习


```
# 定义训练网络时使用哪张显卡或者是使用CPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print('usring {} device.'.format(device))

# 定义图片的预处理工作
dataTransform = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
}
```

首先要定义使用什么硬件资源来训练网络，然后再定义对输入图片的预处理操作。

神经网络的迁移学习


```
# 定义所使用的训练数据集
trainDataset = datasets.ImageFolder(root=os.path.join(dataRoot, 'train'),
                                    transform=dataTransform['train'])
trainNum = len(trainDataset)
flowerList = trainDataset.class_to_idx
classDict = dict((val, key) for key, val in flowerList.items())

jsonStr = json.dumps(classDict, indent=4)
with open('classIndices.json', 'w') as jsonFile:
    jsonFile.write(jsonStr)

batchSize = 8
numWorker = min([os.cpu_count(), batchSize if batchSize > 1 else 0, 8])

print('Using {} dataloader workers every process.'.format(numWorker))
trainLoader = torch.utils.data.DataLoader(trainDataset, batch_size=batchSize,
                                          shuffle=True, num_workers=numWorker,
                                          drop_last=True)

# 定义所使用的测试数据集
valDataset = datasets.ImageFolder(root=os.path.join(dataRoot, 'val'), transform=dataTransform['val'])
valNum = len(valDataset)
valLoader = torch.utils.data.DataLoader(valDataset, batch_size=batchSize, shuffle=False,
                                        num_workers=numWorker)

print('Using {} images for train, {} images for val.'.format(trainNum, valNum))
```

这里是定义了在训练过程中所使用的训练集和测试集，训练集用来训练网络，测试集用来计算网络的预测精度

神经网络的迁移学习


```
# 创建MobileNetV2网络，并加载预训练的权重
net = MobileNetV2(numClass=5)
modelWeightPath = './mobilenet_v2.pth'
assert os.path.exists(modelWeightPath), "file {} dose not exist.".format(modelWeightPath)
preWeights = torch.load(modelWeightPath)

# 去除分类器的权重
preDict = {k: v for k, v in preWeights.items() if 'classifier' not in k}
missingKeys, unexpectedKeys = net.load_state_dict(preDict, strict=False)

# 冻结特征提取的权重
for param in net.features.parameters():
    param.requires_grad = False
```

这里就是创建了一个MobileNetV2的网络，numClass=5是因为我们最终的分类类别是5类花。在加载完MobileNetV2在ImageNet上预训练的权重之后，我们将MobileNet的最后几个用于分类的层的权重去除，然后冻结网络前面用于特征提取的的权重，这样就能在训练中只更新网络最后的几个用于分类的层的参数，而不会改动前面用于特征提取的层的参数。

神经网络的迁移学习


```
# 开始在花分类数据集上训练模型
for epoch in range(epochs):
    net.train()
    runLoss = 0.0
    trainBar = tqdm(trainLoader)
    for step, data in enumerate(trainBar):
        images, labels = data
        optimizer.zero_grad()
        logits = net(images.to(device))
        loss = lossFunction(logits, labels.to(device))
        loss.backward()
        optimizer.step()

        runLoss += loss.item()
        trainBar.desc = 'train epoch[{}/{}] loss:{:.3f}'.format(epoch + 1, epochs, loss)

    # 每训练一轮就将模型进行评估，如果当前的模型精度要大于历史最大值，就将其保存在指定位置
    net.eval()
    acc = 0.0
    with torch.no_grad():
        valBar = tqdm(valLoader, colour='green')
        for valData in valBar:
            valImage, valLabel = valData
            outputs = net(valImage.to(device))
            pred = torch.max(outputs, dim=1)[1]
            acc += torch.eq(pred, valLabel.to(device)).sum().item()
            valBar.desc = 'valid epoch[{}/{}]'.format(epoch + 1, epochs)
    valACC = acc / valNum
    print('[epoch %d] train loss:%.3f test accuracy:%.3f' % (epoch + 1, runLoss / trainSteps, valACC))

    if valACC > bestAcc:
        bestAcc = valACC
        torch.save(net.state_dict(), './backup/mobileNetV2_{}.pth'.format(epoch))

print('Finished Training!')
```

这里是在训练集上进行训练，然后每一轮训练完之后会将模型在测试集上计算它的准确率，如果准确率大于历史最大准确率就会保存这个权重到指定的路径下

神经网络的迁移学习

在训练完模型之后，我们可以看看模型的预测效果：

```
Accuracy:0.9065934065934066
```

上面左图是模型预测tulips图片的结果，右图是模型在测试集上的准确率。可以看到，模型在花分类的数据集上已经有比较好的检测性能了，这表示我们成功地将模型从ImageNet的1000分类任务上迁移学习到花分类的任务上了。

神经网络的迁移学习

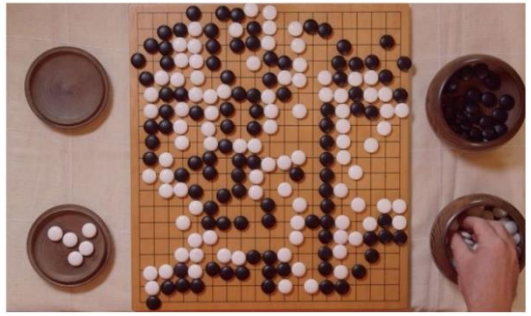
关于迁移学习，大致上有这样的一个规律：可供于训练的数据越多，那么我们需要冻结的网络层数越少，能够训练的层数越多。

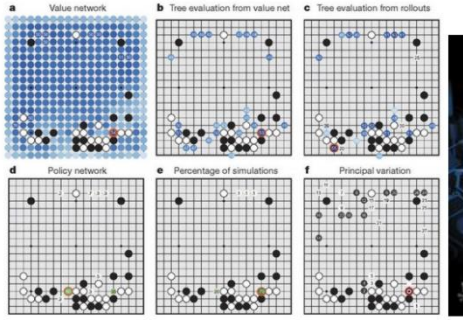
如果我们有大量的数据来训练，那么我们就只需将他人训练好的权重当做网络的初始权重，然后训练整个网络。

事实中，网上的公开数据集非常庞大，我们所下载的预训练权重已经从数据中学到很好的特征提取效果了，所以在我们进行自己的应用时，可以优先考虑使用那些预训练的权重作为网络的初始权重，这样就能站在巨人的肩膀上应用于自己的问题中。

卷积的应用

AlphaGo



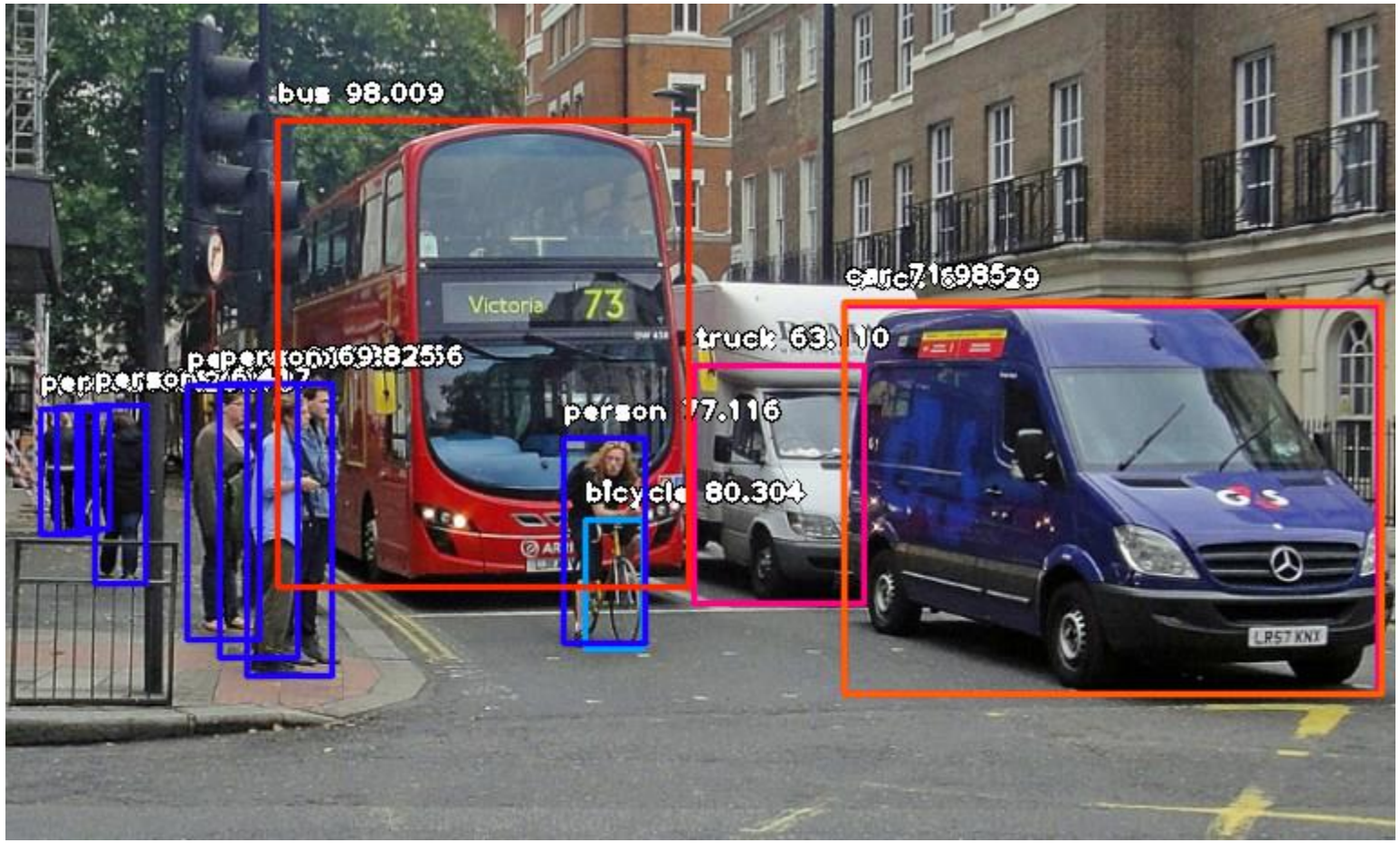


The input to the policy network is a 19 × 19 × 48 image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23 × 23 image, then convolves *k* filters of kernel size 5 × 5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21 × 21 image, then convolves *k* filters of kernel size 3 × 3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1 × 1 with stride 1, with a different bias for each position, and applies a softmax function. The match version of AlphaGo used *k* = 192 filters; Fig. 2b and Extended Data Table 3 additionally show the results of training with *k* = 128, 256 and 384 filters.

policy network:
[19x19x48] Input
CONV1: 192 5x5 filters , stride 1, pad 2 => [19x19x192]
CONV2..12: 192 3x3 filters, stride 1, pad 1 => [19x19x192]
CONV: 1 1x1 filter, stride 1, pad 0 => [19x19] *(probability map of promising moves)*

分布式系统：1202 个CPU 和176 块GPU
单机版：48 个CPU 和8 块GPU
走子速度：3 毫秒-2 微秒

目标检测（Object Detection）



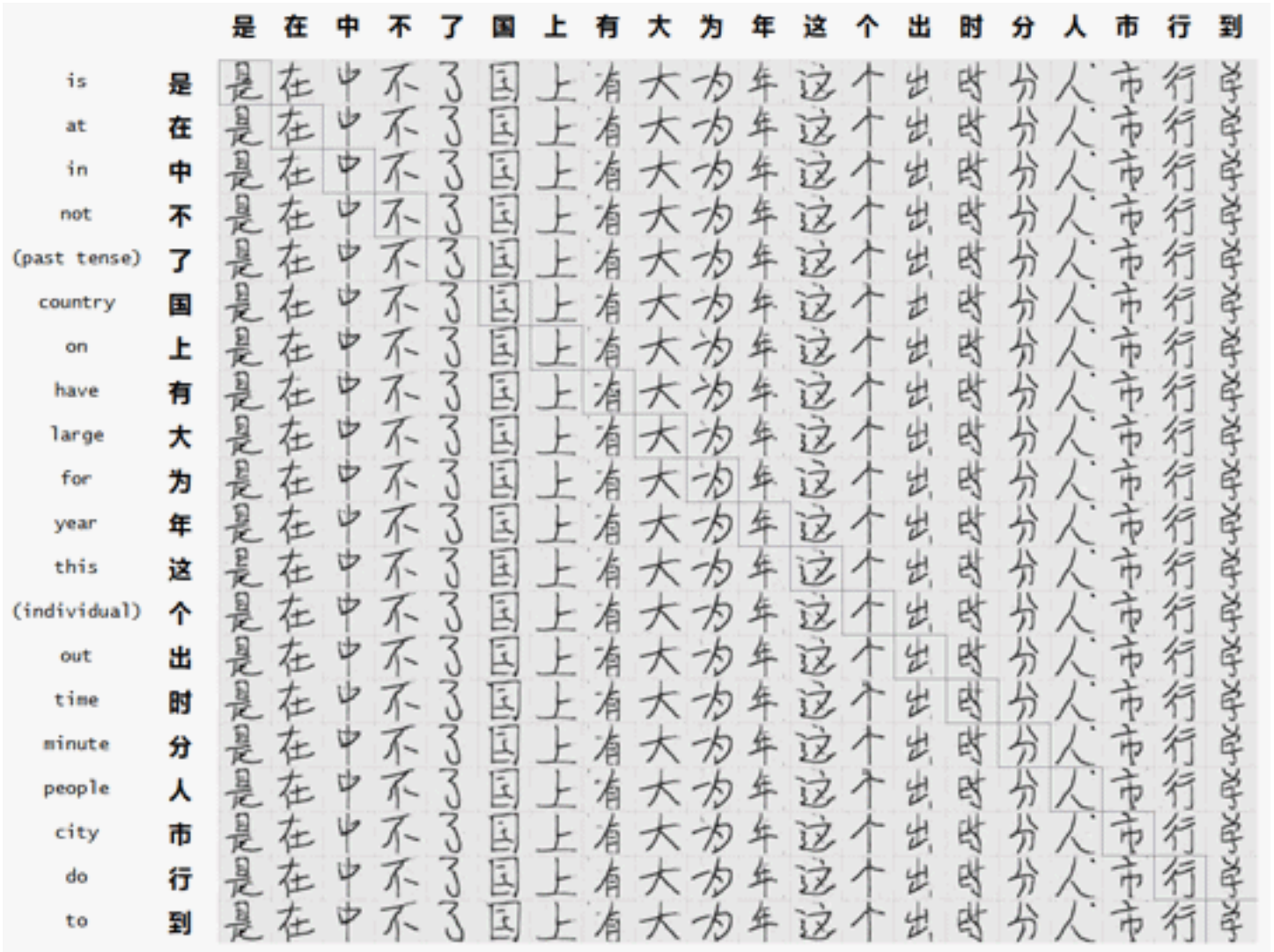
Mask RCNN

Figure 4. More results of **Mask R-CNN** on COCO test images, using ResNet-101-FPN and running at 5 fps, with 35.7 mask AP (Table 1).

OCR


THE WORLD FAMOUS
BRIXHAM
FISH MARKET
BARBER
THE
ROYAL
SCOTSMAN
SWINGBRIDGE
WESTON
BREWING
COMPANY
THIS WAY
HONOR • SERVICE • INTEGRITY
AMERICAN
BUILT
ARMS COMPANY
GARDA
BMW
CLUB
MALIN HEAD
MAY 12TH 200
HOTEL
GREEN

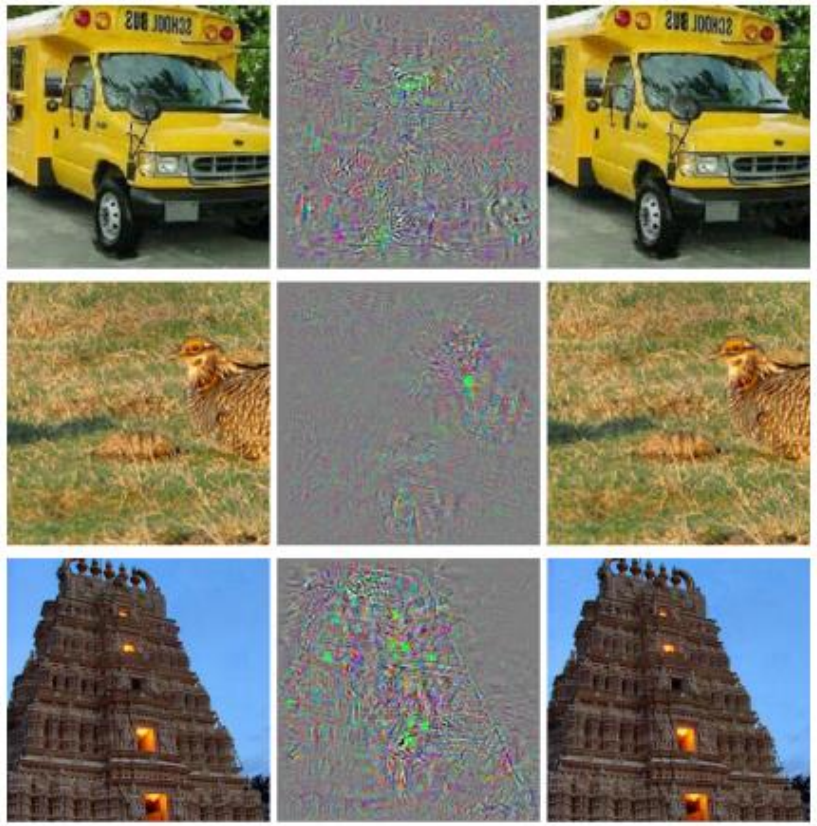
图像生成


是 在 中 不 了 国 上 有 大 为 年 这 个 出 时 分 人 市 行 到
is 是
at 在
in 中
not 不
(past tense) 了
country 国
on 上
have 有
large 大
for 为
year 年
this 这
(individual) 个
out 出
time 时
minute 分
people 人
city 市
do 行
to 到

Deep Dream

画风迁移

对抗样本



(a)

(b)

课后作业

- CNN的局部性假设合理吗?
  - 如何改进?

- 编程练习
  - https://github.com/nndl/exercise/
  - chap5_CNN

    https://github.com/nndl/exercise/tree/master/chap5_CNN
  - 图像分类

谢 谢！