# 训练深度神经网络的一系列提示和技巧

出处：https://towardsdatascience.com/a-bunch-of-tips-and-tricks-for-training-deep-neural-networks-3ca24c31ddc8

翻译以及示例代码添加：Bilibili UP-AiHimeragi(https://space.bilibili.com/21102826)

训练深度神经网络很困难。 它需要知识和经验才能正确训练并获得最佳模型。 在这篇文章中，我想分享我在训练深度神经网络时学到的东西。 以下提示和技巧可能对您的研究有益，并可以帮助您加快网络架构或参数搜索的速度。

现在，让我们开始吧.....

## 1.始终在网络中使用标准化层

如果您使用大批量（例如 10 或更多）训练网络，请使用 BatchNormalization 层。 否则，如果您使用小批量大小（例如 1）进行训练，请改用 InstanceNormalization 层。 请注意，如果增加批大小，BatchNormalization 会提高性能，而当批大小较小时，则会降低性能。 但是，如果使用较小的批量大小，InstanceNormalization 会略微提高性能。 或者您也可以尝试 GroupNormalization。

当在 PyTorch 中使用不同的归一化层时，可以通过简单的代码片段来实现。以下是对每种归一化层的示例代码：

1. **批标准化（Batch Normalization）**：

```
import torch
import torch.nn as nn

# 定义一个带批标准化的神经网络模型
class ModelWithBatchNorm(nn.Module):
    def __init__(self):
        super(ModelWithBatchNorm, self).__init__()
        self.fc1 = nn.Linear(10, 20)
        self.bn1 = nn.BatchNorm1d(20)  # 在全连接层后添加批标准化层
        self.fc2 = nn.Linear(20, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        return x

# 使用带批标准化的模型
model_with_batchnorm = ModelWithBatchNorm()
```

2. **InstanceNormalization**：

```
import torch
import torch.nn as nn

# 定义一个带实例归一化的神经网络模型
```

```
class ModelWithInstanceNorm(nn.Module):
    def __init__(self):
        super(ModelWithInstanceNorm, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.in1 = nn.InstanceNorm2d(16)  # 在卷积层后添加实例归一化层
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.in1(x)
        x = torch.relu(x)
        x = self.conv2(x)
        return x

# 使用带实例归一化的模型
model_with_instancenorm = ModelWithInstanceNorm()
```

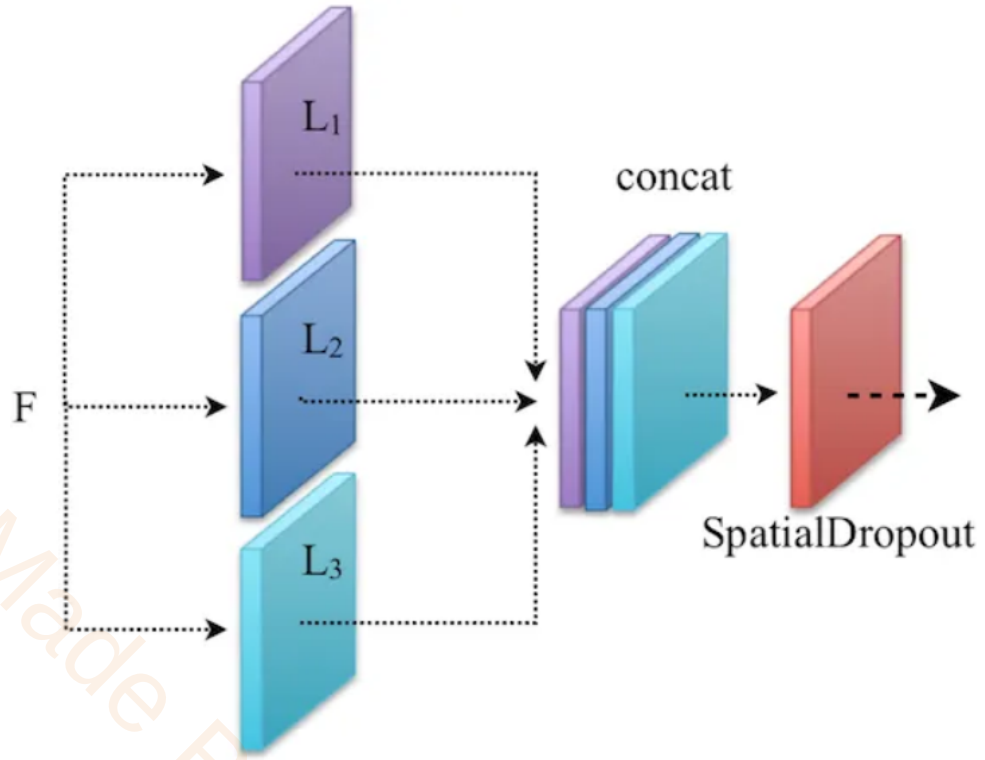
3. **组规范化（Group Normalization）**：

```
import torch
import torch.nn as nn

# 定义一个带组规范化的神经网络模型
class ModelWithGroupNorm(nn.Module):
    def __init__(self):
        super(ModelWithGroupNorm, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.gn1 = nn.GroupNorm(num_groups=4, num_channels=16)  # 在卷积层后添加组规
范化层
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.gn1(x)
        x = torch.relu(x)
        x = self.conv2(x)
        return x

# 使用带组规范化的模型
model_with_groupnorm = ModelWithGroupNorm()
```

这些示例展示了如何在 PyTorch 中使用不同的归一化层。你可以根据自己的需求和实际情况选择合适的归一化策略，并将其集成到你的模型中。

# 2.特征串联后使用 SpatialDropout

如果有两个或多个卷积层（例如 Li）对同一输入（例如 F）进行操作，请在特征串联后使用 SpatialDropout(https://arxiv.org/abs/1411.4280)。 由于这些卷积层对相同的输入进行操作，因此输出特征可能是相关的。 因此 SpatialDropout 会删除这些相关特征并防止网络中的过度拟合。 注意：它主要用于较低层而不是较高层。




SpatialDropout use-case

SpatialDropout代码：https://github.com/yukitsuji/ENet_chainer/blob/fbbb68c77d073f2cd00cfb7bf000fdfec54e18c9/enet/models/spatial_dropout.py

为了方便大家使用，up将提供pytorch框架下的代码：

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class SpatialDropout(nn.Module):
    """SpatialDropout regularization."""

    def __init__(self, dropout_ratio):
        super(SpatialDropout, self).__init__()
        if not 0.0 <= dropout_ratio < 1.0:
            raise ValueError('dropout_ratio must be in the range [0, 1)')
        self.dropout_ratio = dropout_ratio

    def forward(self, x):
        if self.training:
            batch_size, channels, height, width = x.size()
            # Generate a binary mask
            mask = torch.ones(batch_size, channels, height, width,
device=x.device)
            mask = F.dropout2d(mask, p=self.dropout_ratio, training=True)
            # Apply the mask to the input tensor
            x = x * mask
        return x


def spatial_dropout(x, ratio=0.1):
    """spatial_dropout(x, ratio=.1)"""
```

```
    if ratio < 0 or ratio >= 1:
        raise ValueError('dropout_ratio must be in the range [0, 1)')

    return SpatialDropout(ratio)(x)


if __name__ == '__main__':

    # 定义一个简单的CNN模型
    class CNN(nn.Module):
        def __init__(self):
            super(CNN, self).__init__()
            self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
            self.dropout = SpatialDropout(dropout_ratio=0.5)  # SpatialDropout层

        def forward(self, x):
            x = self.conv1(x)
            x = F.relu(x)
            x = self.dropout(x)  # 应用SpatialDropout
            return x

    # 创建CNN模型的实例
    model = CNN()

    # 将模型设置为训练模式
    model.train()

    # 生成一些虚拟输入数据
    input_data = torch.randn(1, 1, 28, 28)  # (batch_size, channels, height,
width)

    # 使用spatial_dropout函数对输入数据应用SpatialDropout
    output = model(input_data)

    # 将模型设置为评估模式
    model.eval()

    # 在评估期间，spatial_dropout函数将返回未经修改的输入
    output_eval = spatial_dropout(input_data)  # 没有应用dropout

    print("训练时输出的形状:", output.shape)
    print("评估时输出的形状:", output_eval.shape)
```

这段代码演示了如何实现空间dropout（Spatial Dropout）以及如何在训练和评估阶段使用它。空间dropout与传统的dropout类似，但是在特征图的维度上进行操作，而不是在向量上操作。

在这段代码中：

- `SpatialDropout` 类继承自 `nn.Module`，它接受一个参数 `dropout_ratio`，用于指定 dropout 的比例。
- `forward` 方法定义了如何应用空间dropout。在训练阶段，它会生成一个与输入张量相同大小的二值掩码，然后使用 `F.dropout2d` 函数将该掩码应用到输入张量上，实现空间dropout。
- `spatial_dropout` 函数是一个简单的包装器，用于方便地应用 `SpatialDropout` 类。
- 在 `CNN` 类中，`SpatialDropout` 层被应用在卷积层后面。
- 主程序部分展示了如何在训练和评估阶段使用模型以及如何应用空间dropout。

这个例子中的输出将显示训练时和评估时输出的形状，以展示在不同阶段的行为差异。

# 3.多尺度特征池化模块

要捕获对象周围的上下文信息，请使用多尺度特征池化模块。这可以进一步帮助提高准确性，并且该思想已成功应用于语义分割或前景分割。

这里up推荐空间金字塔池化，非常经典，空间金字塔池化（SPP）是一个消除网络固定大小约束的池化层，即 CNN 不需要固定大小的输入图像。 具体来说，我们在最后一个卷积层之上添加一个 SPP 层。 SPP 层汇集特征并生成固定长度的输出，然后将其输入到全连接层（或其他分类器）。 换句话说，我们在网络层次结构的更深层（卷积层和全连接层之间）执行一些信息聚合，以避免在开始时进行裁剪或扭曲。

代码如下(https://github.com/yifanjiang19/sppnet-pytorch):

```
import math
import torch
from torch import nn


def spatial_pyramid_pool(self, previous_conv, num_sample, previous_conv_size,
out_pool_size):
    '''
    previous_conv: a tensor vector of previous convolution layer
    num_sample: an int number of image in the batch
    previous_conv_size: an int vector [height, width] of the matrix features size
of previous convolution layer
    out_pool_size: a int vector of expected output size of max pooling layer

    returns: a tensor vector with shape [1 x n] is the concentration of multi-
level pooling
    '''
    # print(previous_conv.size())
    for i in range(len(out_pool_size)):
        # print(previous_conv_size)
        h_wid = int(math.ceil(previous_conv_size[0] / out_pool_size[i]))
        w_wid = int(math.ceil(previous_conv_size[1] / out_pool_size[i]))
        h_pad = (h_wid * out_pool_size[i] - previous_conv_size[0] + 1) / 2
        w_pad = (w_wid * out_pool_size[i] - previous_conv_size[1] + 1) / 2
        maxpool = nn.MaxPool2d((h_wid, w_wid), stride=(h_wid, w_wid), padding=
(h_pad, w_pad))
        x = maxpool(previous_conv)
        if (i == 0):
            spp = x.view(num_sample, -1)
            # print("spp size:",spp.size())
        else:
            # print("size:",spp.size())
            spp = torch.cat((spp, x.view(num_sample, -1)), 1)
    return spp
```

这个函数实现了在 PyTorch 中的空间金字塔池化（Spatial Pyramid Pooling）。它接受一个先前的卷积层、样本数量、先前卷积层特征图的大小以及期望的输出池化大小作为输入。然后，它使用不同的核大小应用最大池化，并将结果拼接起来。

这个函数通常用于目标检测或图像分类等任务的架构中，以有效地处理各种大小的输入。

当您想要在您的 PyTorch 模型中使用这个空间金字塔池化函数时，您可以像下面这样调用它：

```
import torch
from torch import nn

# 定义模型
class YourModel(nn.Module):
    def __init__(self):
        super(YourModel, self).__init__()
        # 定义卷积层等其他层

    def forward(self, x):
        # 假设 x 是您的输入图像或特征图
        # 假设您的模型已经定义了一个名为 spatial_pyramid_pool 的方法
        # 我们将在这里调用 spatial_pyramid_pool 方法
        # 假设 num_sample, previous_conv_size, out_pool_size 这些参数已经定义好了
        spp_output = self.spatial_pyramid_pool(x, num_sample,
previous_conv_size, out_pool_size)
        # 其他网络层的处理
        return spp_output

    def spatial_pyramid_pool(self, previous_conv, num_sample,
previous_conv_size, out_pool_size):
        # 这里是您的 spatial_pyramid_pool 函数的实现
        # 您可以将之前提供的函数实现放在这里

# 创建您的模型实例
model = YourModel()

# 假设您有一批输入图像 x，其形状为 [batch_size, channels, height, width]
batch_size = 32
channels = 3
height = 224
width = 224
x = torch.randn(batch_size, channels, height, width)

# 将输入传递给模型
output = model(x)

# output 将是您的模型的输出，即 spatial pyramid pooling 的结果
```

在这个示例中，我们创建了一个模型 `YourModel`，其中包含您定义的空间金字塔池化方法。然后，我们将一批输入数据传递给模型，模型将会调用空间金字塔池化方法并返回输出结果。

# 4.选择一个正确的优化器

有许多流行的自适应优化器，例如 Adam、Adagrad、Adadelta 或 RMSprop 等。SGD+momentum(https://cs231n.github.io/neural-networks-3/#sgd) 广泛应用于各个问题领域。 有两件事需要考虑：首先，如果您关心快速收敛，请使用 Adam 等自适应优化器，但它可能会以某种方式陷入局部最小值并提供较差的泛化能力。 其次，SGD+momentum可以实现找到全局最小值，但它依赖于稳健的初始化，并且可能比其他自适应优化器需要更长的时间才能收敛。 我建议您使用 SGD+momentum，因为它往往会达到更好的最佳状态。

# 5.关于学习率

有三个学习率起点可供选择（即 1e-1、1e-3 和 1e-6）。 如果您对预训练模型进行微调，请考虑低于 1e-3（例如 1e-4）的低学习率。 如果您从头开始训练网络，请考虑学习率大于或等于 1e-3。 您可以尝试这些起点并进行调整，看看哪一个最有效，然后选择那个。 另一件事是，您可以考虑使用学习率调度程序随着训练的进行而减慢学习率。 这也有助于提高网络性能。

# 6.关于数据

更多的数据胜过聪明的算法！ 始终使用数据增强，例如水平翻转、旋转、缩放裁剪等。这有助于大幅提高准确性。

# 7.关于GPU

你必须有高速 GPU 才能进行训练，但这有点昂贵。 如果你想使用免费的云GPU，我建议使用Google Colab。

# 8.在 ReLU 之前使用 Max-pooling 可以节省一些计算量。

由于 ReLU 会将值阈值化为零：f(x)=max(0,x)，而 Max 池化仅对最大激活值进行池化：f(x)=max(x1,x2,...,xi)，因此应使用 Conv→MaxPool→ ReLU 而不是 Conv →ReLU→ MaxPool。

例如，假设我们从 Conv 中得到了两个激活值（即 0.5 和 -0.5）：

因此 MaxPool > ReLU = max(0, max(0.5,-0.5)) = 0.5 而 ReLU > MaxPool = max(max(0,0.5), max(0,-0.5)) = 0.5 明白了吗？这两个操作的输出仍然是 0.5。在这种情况下，使用 MaxPool > ReLU 可以节省一次max操作。

# 9.考虑使用深度可分离卷积

考虑使用深度可分离卷积(https://arxiv.org/abs/1610.02357)运算，与普通卷积运算相比，速度快并且大大减少了参数数量。

代码如下(https://github.com/tstandley/Xception-PyTorch):

```
import torch.nn as nn
import torch

class SeparableConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=1, stride=1,
padding=0, dilation=1, bias=False):
        super(SeparableConv2d, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, in_channels, kernel_size, stride,
padding, dilation, groups=in_channels,
                               bias=bias)
        self.pointwise = nn.Conv2d(in_channels, out_channels, 1, 1, 0, 1, 1,
bias=bias)

    def forward(self, x):
        x = self.conv1(x)
        x = self.pointwise(x)
        return x


if __name__ == '__main__':
    # 假设输入是 3 个通道，高度为 32，宽度为 32 的图像
```

```
    input_tensor = torch.randn(1, 3, 32, 32)  # (batch_size, in_channels,
height, width)

    # 创建 SeparableConv2d 实例
    separable_conv = SeparableConv2d(in_channels=3, out_channels=64,
kernel_size=3, padding=1)

    # 执行前向传播
    output_tensor = separable_conv(input_tensor)

    # 打印输入和输出的形状
    print("Input shape:", input_tensor.shape)
    print("Output shape:", output_tensor.shape)
```

上面的代码实现了一个使用PyTorch的可分离卷积层。这种类型的层包含两个阶段：深度卷积和逐点卷积。

以下是代码中正在发生的简要说明：

1. 在`__init__`方法中：
    - `self.conv1`：这是深度卷积。它独立地在每个输入通道上操作，创建一组特征图。
    - `self.pointwise`：这是逐点卷积。它对深度卷积的输出应用1x1卷积，将其合并为最终输出。
2. 在`forward`方法中：
    - `x = self.conv1(x)`：这在输入`x`上执行深度卷积。
    - `x = self.pointwise(x)`：这对深度卷积的输出执行逐点卷积，生成最终输出。

与传统卷积层相比，这种架构在计算效率上更高，特别是当输入通道较多时。它经常用于深度神经网络中，以减少参数和计算量，同时保持表示能力。

# 10.不要放弃

不要放弃。 相信自己，你能做到！ 如果您仍然没有获得您想要的高精度，请调整您的超参数、网络架构或训练数据，直到获得您想要的精度👌。在这里，up也祝您科研顺利，毕业顺利，生活顺利，今日走过了所有弯路，从此人生尽是坦途，祝您成功！谢谢您的观看！