Readme

Day03

Date:2021-07-26
Made By: 纸 石头 紫阳花

多分类问题

Softmax分类器

由于希望输出层满足离散概率分布的需求（概率和为0且概率都大于0），故希望输出之间带有竞争性。

故处理多分类问题时，在处理前几层时还是用Sigmoid函数进行激活，但在最后一个隐藏层，用Softmax函数来代替。

Softmax函数的输出满足每一个输出都大于零且输出之和等于1的条件。

Softmax函数：

$$P(y=i)=\frac{e^{z_i}}{\sum_{j=0}^{K-1}e^{z_j}},(i<k)$$

Z^l层表示最后一个隐藏层的第i个线性结点的输出，Z的维度为K（即共有K个特征）。

利用指数进行运算，可以保证输出结果大于0。分母部分的求和相当于将每一个分类的输出进行一次求和。

换言之，相当于是做了一次 归一化 操作。

损失函数

$$NLLLoss(\hat{Y}, Y) = -Y \log_2 \hat{Y}$$

NLLLoss（Negative Log Likelihood Loss）

numpy 实现损失函数

```
import numpy as np


y = np.array([1, 0, 0])
z = np.array([0.2, 0.1, -0.1])
y_predict = np.exp(z) / np.exp(z).sum()
loss = (-y*np.log(y_predict)).sum()
print(loss)
```

Pytorch 实现损失函数

```
import torch


# y是输出层，在需要做交叉熵损失时，输出层必须是LongTensor类型的张量
y = torch.LongTensor([0])
# z是最后一层隐藏层的输出结果(不要做激活，直接交给交叉熵损失)
z = torch.Tensor([[0.2, 0.1, -0.1]])
criterion = torch.nn.CrossEntropyLoss()
loss = criterion(z, y)
print(loss)
```

NLLLoss 和 CrossEntropyLoss之间的区别

MNIST数据集多分类问题的代码实例

构建数据集

```
import torch
from torchvision import transforms      # 针对图像进行处理的工具
from trochvision import datasets
from torch.utils.data import DataLoader
import torch.nn.functional as F         # 使用更流行的relu取代Sigmoid作为激活函数
import torch.optim as optim             # 构建优化器

batch_size = 64
# 构建图像处理工具
transform = transforms.Compose([
    transforms.ToTensor(),                      # Compose可调用对象将Pillow图像转换为Tensor
    transforms.Normalize((0.1307, ), (0.3081,))    # 为图像数据根据均值（0.1307）和标准差(0.3081)进行归一化
])

# 对数据集进行加载
train_dataset = datasets.MNIST(root="../dataset/mnist/",
                               train=True,
```

```
                                download=True,
                                transform=transform)
test_dataset = datasets.MNIST(root="../dataset/mnist/",
                                train=False,
                                download=True,
                                transform=transform)
# 构建数据加载器
train_loader = DataLoader(train_dataset,
                          shuffle=True,
                          batch_size=batch_size)
test_loader = DataLoader(test_dataset,
                          shuffle=False,
                          batch_size=batch_size)
```

设计模型

最后一个隐藏层不做激活

```
class Net(torch.nn.Moduel):
    def __init__(self):
        super(Net, self).__init__()
        self.l1 = torch.nn.linear(784, 512)
        self.l2 = torch.nn.linear(512, 256)
        self.l3 = torch.nn.linear(256, 128)
        self.l4 = torch.nn.linear(128, 64)
        self.l5 = torch.nn.linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        return self.l5(x)

model = Net()
```

损失函数与优化器

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = optimizer.SGD(model.parameters(),
                          lr=0.01,
                          momentum=0.5)     # momentum 代表冲量
```

训练函数

```
def train(epoch):
    running_loss = 0.0
    for batch_idx, data in enumerate(train_loader, 0)
    input, target = data
    optimizer.zero_grad()

    # Forward
    outputs = model(inputs)
    loss = criterion(outputs, target)
    # Backward
    loss.backward()
    # Update
```

```
    optimizer.step()

    running_loss += loss.item()
    if batch_idx % 300 = 299
        print("[%d, %5d] loss: %.3f" % (epoch + 1, batch_idx + 1, running_loss / 300))
        running_loss = 0.0
```

测试泛化能力

```
def test():
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = module(images)
            _, predicted = torch.max(outputs.data, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print("Accuracy on test set: %d %%" % (100 * correct / total))
```

训练周期

```
if __name__ = "__main__":
    for epoch in range(100):
        train(epoch)
        if epoch % 10 == 0
            test()
```

卷积神经网络基础

卷积神经网络工作概述

1. 图像输入、
2. 卷积层保留图像空间特征
   卷积神经网络直接按照图像原本的空间结构对图像信息进行保存
3. 下采样
   通道数不变，对图像的宽度和高度进行降维，降低运算需求
4. 重复卷积和下采样操作，将三阶向量展开为一阶向量
5. 利用全连接层映射到所需维度（10维）
6. 最终输出一个一阶向量
7. 利用Softmax分类器对分布进行计算，解决分类问题

故构建神经网络时首先要明确输入和输出的张量的维度。

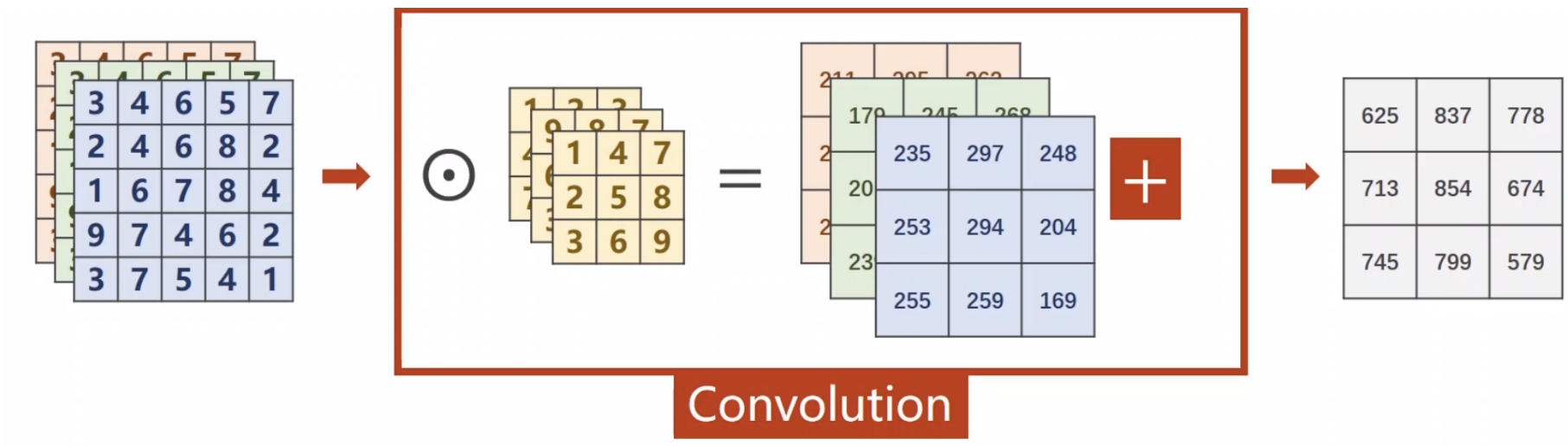
卷积和下采样的过程被封装为 特征提取器 (Feature Extraction) 通过卷积运算来找到某种特征。

经特征提取后的全连接层称为分类器。

Convolution

默认将图像的RGBChannel看作神经网络的Input Channel，故每个图像块都可以看作一个3xHxW的张量，对图像中每一个块进行遍历与卷积核进行卷积运算得到输出结果，然后拼接输出结果。卷积之后，图像的通道数、宽度、高度都可能改变。如果由多个通道则每个通道都要搭配一个卷积层，卷积层的数量和通道的数量保持一致，这些卷积层合称为一个卷积核。

多通道卷积运算后得到的n个矩阵应当做加法得到新矩阵，利用新矩阵进行下采样。



若要输出多通道的结果，可以使用多个卷积核，卷积核的数量与最终输出的通道数保持一致。

卷积层计算过程

```
import torch

in_channels, out_channels = 5, 10
width, height = 100, 100
kernel_size = 3         # 卷积核维度（一般用奇数）
batch_size = 1

input = torch.randn(batch_size,
                    in_channels,
                    width,
                    height)
# Conv2d模型 创建卷积层
conv_layer = torch.nn.Conv2d(in_channels,
                             out_channels,
                             kernel_size=kernel_size)
output = conv_layer(input)

print(input.shape)
print(output.shape)
print(conv_layer.weight.shape)
```

卷积层初始化常见权重：

padding：为了控制图像矩阵经卷积计算后的矩阵大小对Input矩阵外围用0进行填充，从而维持原输入图像的维度保持不变
示例：Input:5x5　Kernel:3x3　要维持Output维度也为5x5 则需要对Input进行padding操作，填充为7x7的矩阵。
padding的默认操作是填充零。

代码示例：

```
import torch

input = [3, 4, 6, 5, 7,
         2, 4, 6, 8, 2,
         1, 6, 7, 8, 4,
         9, 7, 4, 6, 2,
         3, 7, 5, 4, 1]
```

```
# 四个参数分别是
# batch的维度
# 通道的维度
# 宽度
# 高度
input = torch.Tensor(input).view(1, 1, 5, 5)
# 构建卷积层 输入输出都为一个通道 卷积核的维度设置为3 填充层设置位1层 不设置偏置量
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=3, padding=1, bias=False)
# 构造卷积核 输出通道数 输入通道数 宽度 高度
kernel = torch.Tensor([1, 2, 3, 4, 5, 6, 7, 8, 9]).view(1, 1, 3, 3)
# 利用卷积核数据对卷积层权重进行初始化
conv_layer.weight.data = kernel.data

output = conv_layer(input)
print(output)
```

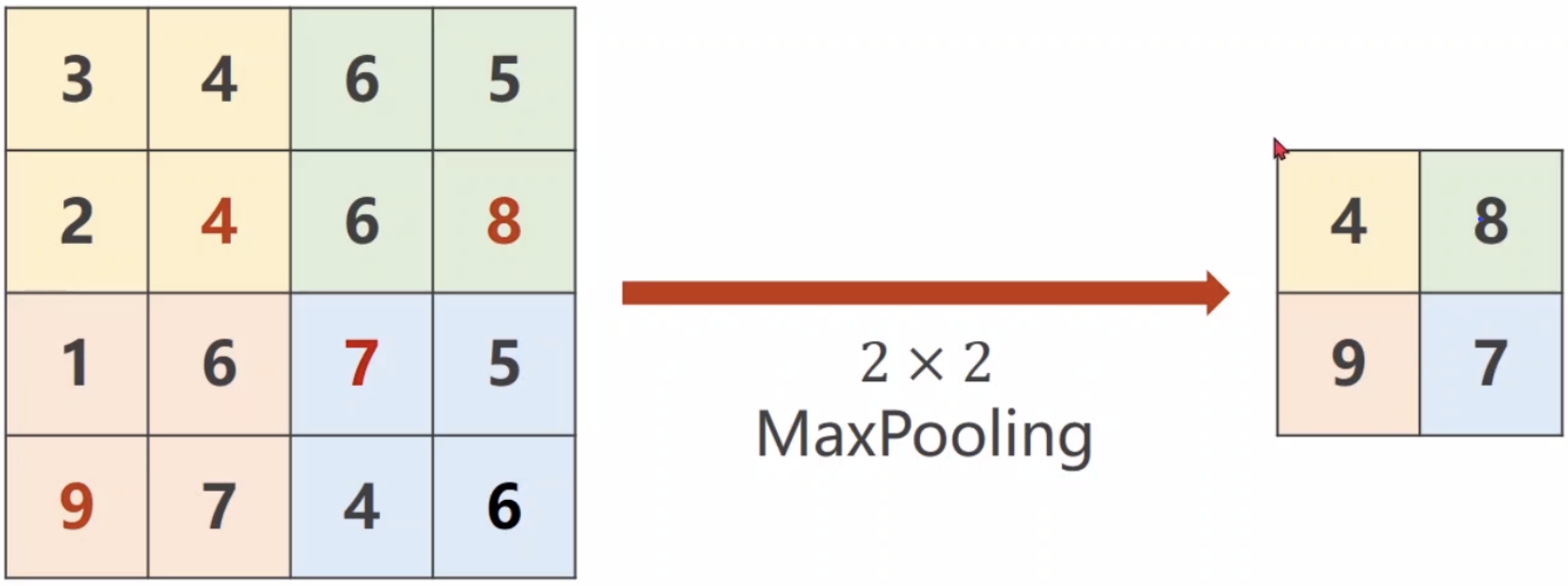
步长（Stride） 默认步长是1，决定卷积核进行卷积运算对Input矩阵每次处理的移位长度从而有效降低经Kernel处理后的数据维度。

```
import torch

input = [3, 4, 6, 5, 7,
         2, 4, 6, 8, 2,
         1, 6, 7, 8, 4,
         9, 7, 4, 6, 2,
         3, 7, 5, 4, 1]
# 四个参数分别是
# batch的维度
# 通道的维度
# 宽度
# 高度
input = torch.Tensor(input).view(1, 1, 5, 5)
# 构建卷积层 输入输出都为一个通道 卷积核的维度设置为3 填充层设置位1层 不设置偏置量
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=3, stride=2, bias=False)
# 构造卷积核 输出通道数 输入通道数 宽度 高度
kernel = torch.Tensor([1, 2, 3, 4, 5, 6, 7, 8, 9]).view(1, 1, 3, 3)
# 利用卷积核数据对卷积层权重进行初始化
conv_layer.weight.data = kernel.data

output = conv_layer(input)
print(output)
```

下采样

常用的一种下采样方式称为**MaxPooling** 即最大池化层。

最大池化层是没有权重的，nxnMaxPooling默认Stride为n（即池化层在Input矩阵中的作用范围是不重叠的）。MaxPooling将Input矩阵按照池化层宽高进行分割，取每个区域中的最大值拼接形成Output。



MaxPooling只能作用于同一个通道中，故MaxPooling不会改变通道数只会对图像矩阵的宽高造成影响。

MaxPooling代码实例

```
import torch

input = [3, 4, 6, 5, 7,
         2, 4, 6, 8, 2,
         1, 6, 7, 8, 4,
         9, 7, 4, 6, 2,
         3, 7, 5, 4, 1]
input = torch.Tensor(input).view(1, 1, 4, 4)
# 构建最大池化层
maxpooling_layer = torch.nn.MaxPool2d(kernel_size=2)

output = maxpooling_layer(input)
print(output)
```

卷积核维度如何进行确定

用于处理MNIST的简单神经网络实例

```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 卷积核维度对输出层数据的影响 输出层宽高 = 输入层宽高 - （kernel_size - 1）
        # 参数为输入通道数和输出通道数
        self.conv1 = torch.nn.Conv2d(1, 10, kernel_size=5)  # 从(batch, 1, 28, 28) 到 (batch, 10, 24, 24)
        self.conv2 = torch.nn.Conv2d(10, 20, kerner_size=5) # 从(batch, 10, 12, 12) 到 (batch, 20, 8, 8)
        self.pooling = torch.nn.MaxPool2d(2)    # 最大池化层
        self.fc = torch.nn.Linear(320, 10)      # 全连接层

    def forward(self, x):               # 前馈过程:  卷积-》池化-》激活-》降维 -》全连接 -》输出分类结果
        batch_size = x.size(0)
        x = F.relu(self.pooling(self.conv1(x)))
        x = F.relu(self.pooling(self.conv2(x)))
        # (batch, 20, 4, 4) -> (batch, 320)
        x = x.view(batch_size - 1)  # flatten
```

```
        # 由全连接层将维度降到10维
        # 需要用交叉熵损失时最后一层不进行激活
        x = self.fc(x)
        return x

model = Net()

criterion = torch.nn.CrossEntropyLoss()
optimizer = optimizer.SGD(model.parameters(),
                          lr=0.01,
                          momentum=0.5)
```

如何使用显卡进行计算

```
# 选择显卡cuda:0作为运行硬件
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# 将所有模型参数与权重转移到 CUDA 张量
model.to(device)
```

```
def train(epoch):
    running_loss = 0.0
    for batch_idx, data in enumerate(train_loader, 0)
    input, target = data
    # 将数据集也转移到GPU中，测试中亦然
    input, target = input.to(device), target.to(device)
    optimizer.zero_grad()

    # Forward
    outputs = model(inputs)
    loss = criterion(outputs, target)
    # Backward
    loss.backward()
    # Update
    optimizer.step()

    running_loss += loss.item()
    if batch_idx % 300 = 299
        print("[%d, %5d] loss: %.3f" % (epoch + 1, batch_idx + 1, running_loss / 300))
        running_loss = 0.0
```