基础知识

数据操作

- torch.cat((x,y),dim):对x,y张量进行合成，dim为0表示行叠加
- torch.randn((3,4)):生成3*4的正态分布的矩阵；torch.normal()可以设置均值和标准差
- 可使用指定索引写入数据:x[1,2]
- x[0:2,:]:表示选中0-1行的所有列

常见函数和操作

- .weight(bias).data:可以查看模型参数，支持修改，但不会被梯度追踪系统记录
- with torch.no_grad():可以临时禁用梯度操作，节省内存，也可以用来修改模型参数

线性代数实现

- a.T:转置
- a.mean():平均值
- 用axis来表示维度，在维度上做计算，axis为多少最后的shape相对应的维度就会丢失
- .dot:点积
- torch.norm():L2范数:向量每个元素平方和开根号
- torch.abs(u).sum:L1范数：绝对值求和
- .sum()可以求和，并使用dim参数在特定方向上求和

求导数

- $\frac{\partial \vec{y}}{\partial x}$是列向量
- $\frac{\partial y}{\partial \vec{x}}$是行向量
- 因此向量对向量求导最后是矩阵
- 自变量和因变量可以扩展到矩阵

正向传播与反向传播

- 正向传播空间复杂度为O(1)
- 反向传播空间复杂度为O(N)，因为要存储正向传播的计算结果
- 反向传播实际上是从尾到头使用链式求导，并根据正向传播计算出来的值求得梯度

自动求导

- 计算梯度前要设置自变量为.requires_grad_(True),保证计算图不被释放
- 调用一个张量的.backward()方法:执行反向传播算法
- 计算自变量梯度:x.grad
- 多次使用自动求导时要记得对梯度清零：x.grad.zero_()，否则梯度会累加
- 使用.sum()方法梯度变为1
- 向量对向量求导使用.sum().backward()可以避免传入'gradient'参数
- u = y.detach()可以让计算机将u视为常数，而不是函数

优化器(optim)

- SGD:随机梯度下降
- momentum梯度下降：根据梯度进行指数加权平均，这个方法可以让之前的梯度占取一部分的权重，并且离该点越近权重越大，使得当前的下降方向不完全由当前梯度决定，使得震荡不那么剧烈。并且使下降具有惯性。
  - $$v_t = \beta v_{t-1} + \quad (1-\beta)\frac{\partial J}{\partial w}$$
    $$w = w - \alpha v_t$$
- RMSprop梯度下降：用梯度平方进行指数加权平均，这个方法可以自适应调整学习率，还能防止震荡。$\epsilon \approx 10^{-8}$ ，防止分母为0。
  - $$v_t = \beta v_{t-1} + \quad (1-\beta)(\frac{\partial J}{\partial w})^2$$
    $$w = w - \alpha \frac{\frac{\partial J}{\partial w}}{\sqrt{v_t + \epsilon}}$$
- Adam算法；momentum和rmsprop的结合，核心是一阶矩和二阶矩的估计
  - $\beta 1$通常为0.9，$\beta 2$通常为0.999,g_t为梯度，α一般为0.001
    - $$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot g_t$$
      $$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot g_t^2$$
- 偏差修正：早期时提升收敛效率。
  - $$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
    $$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
- 更新:
  - $$\theta_{t+1} = \theta_t - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

感知机

- 用来进行二元分类预测，输出可以是-1(0)或1，不能拟合xor函数，其只能产生线性分割面

- 起始权重和偏置为0
- 当labels和预测值异号时，说明分类错误，进行梯度下降

XOR

用两条线性分割线来进行分类

线性回归

线性层中权重的行数(out_features)是输出神经元的数量，列数(in_features)是输入特征的数量

样本的行数是样本数量，列数是输入特征数量，偏置也和输出神经元数量有关

```
import torch
import random
# from d2l import torch as d2l
def synthetic_data(w, b, num_examples):
    X = torch.normal(0,1,(num_examples,len(w)))
    y = X @ w + b
    return X, y.reshape((-1, 1))
# 设置真实w和b生成训练样本
true_w = torch.tensor([[2], [-3.4]])
true_b = 4.2
features, labels = synthetic_data(true_w,true_b,1000)
# 抽取随机批量
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(
            indices[i:min(i + batch_size,num_examples)]
        )
        # 使用yield，执行一次返回一次；并且使用[]索引会让抽取到的每一行组合成一个新的张量
        yield features[batch_indices], labels[batch_indices]

batch_size = 10

w = torch.normal(0, 1,size = (2, 1), requires_grad=True)
b = torch.zeros(1,requires_grad=True)
# 定义模型
def linreg(X, w, b):
```

```
    return X @ w + b

# 定义损失函数
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape))**2 / 2

# 梯度下降
def sgd(params, lr, batch_size):
    with torch.no_grad():
        for param in params:
            # 在这里除以batch_size是因为前面的损失函数没有除，在这里除也是一样的
            param -= lr * param.grad / batch_size
            param.grad.zero_()

lr = 0.004
num_epochs = 3
net = linreg
loss = squared_loss

for epoch in range(num_epochs):
    for X,y in data_iter(batch_size, features, labels):
        l = loss(net(X,w,b),y)
        l.sum().backward()
        sgd([w, b], lr, batch_size)
    with torch.no_grad():
        train_l = loss(net(features, w, b),labels)
        print(f'epoch{epoch + 1}, loss {float(train_l.mean()):f}')
```

使用深度学习框架来简洁实现

- nn.MSEloss()表示均方差损失函数
- TensorDataset(*data_arrays)：将特征和标签打包为一个数据集，方便批量操作。
- next(iter(data_iter))将data_iter转变为一个迭代器
- torch里面optim模块存储了优化算法，如SGD
- net.parameters()表示把模型的所有参数传进优化算法中

```
import torch
from torch.utils import data
from torch import nn

# 生成数据的函数
def generate_data(w, b, num_examples):
    X = torch.normal(0, 1, (num_examples, len(w)))  # 使用相同的正态分布生成数据
    y = X @ w + b
    return X, y.reshape((-1, 1))
```

```python
# 设置真实参数
true_w = torch.tensor([[2], [-3.4]])
true_b = 4.2

# 生成较大的数据集，然后分割为训练集和测试集
num_total = 10  # 总样本数
train_ratio = 0.8  # 80% 作为训练集，20% 作为测试集
train_size = int(num_total * train_ratio)
test_size = num_total - train_size

# 生成数据
X, y = generate_data(true_w, true_b, num_total)

# 分割训练集和测试集
train_features = X[:train_size]
train_labels = y[:train_size]
test_features = X[train_size:]
test_labels = y[train_size:]

# 加载数据集
def load_array(data_arrays, batch_size, is_train=True):
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 32
train_iter = load_array((train_features, train_labels), batch_size)
test_iter = load_array((test_features, test_labels), batch_size, is_train=False)

# 定义模型
net = nn.Sequential(nn.Linear(2, 1))

# 初始化模型参数
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)

# 损失函数
loss = nn.MSELoss()

# 优化器
trainer = torch.optim.SGD(net.parameters(), lr=0.01,weight_decay=0.03)

# 训练循环
num_epochs = 20
for epoch in range(num_epochs):
    for X, y in train_iter:
        net.train()
        l = loss(net(X), y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    train_l = loss(net(train_features), train_labels)
    print(f'epoch {epoch + 1}, train loss {float(train_l):f}')
```

```
# 测试集上验证
with torch.no_grad():
    test_l = loss(net(test_features), test_labels)
    print(f'Final test loss: {float(test_l):f}')
```

softmax分类预测

加入了tensorboard进行数据可视化

```
import torch
import torchvision
from torch.utils import data
from torchvision import transforms
from torch import nn
from torch.utils.tensorboard import SummaryWriter
trainset = torchvision.datasets.FashionMNIST(
    "./data",train = True,transform=transforms.ToTensor(),download=False
)
teset = torchvision.datasets.FashionMNIST(
"./data",train = False,transform=transforms.ToTensor(),download=False
)
trainloader = data.DataLoader(trainset,batch_size=64,shuffle=True)
# 测试集不需要打乱
teloader = data.DataLoader(teset,batch_size=64,shuffle=False)
# dataset[0] 访问的是数据集中的第一个样本，这个样本是一个元组 (image, label)。

# 像素为28*28，把他们展开成784的向量
num_inputs = 784
num_outputs = 10
# 十个类别所以w有十列，一个图片784个像素所以有784行
# 使用线性层是为了让输入特征映射到类别概率分布，使用softmax进行概率提取
# 通过nn.flatten()展平输入值
net = nn.Sequential(nn.Flatten(),nn.Linear(784,10))


loss = nn.CrossEntropyLoss()

trainer = torch.optim.Adam(net.parameters(),lr=0.001,eps=1e-8,betas=(0.9,0.999))

num_train_epochs = 10

writer = SummaryWriter('softmax')
# num_test_epochs = 3
def train_model(net,dataloader,loss,trainer,num_train_epochs):
    net.train()
    print("Train:")
```

```
    for epoch in range(num_train_epochs):
        total_loss , total_correct = 0,0
        for X,y in trainloader:
            y_hat = net(X)
            l = loss(y_hat ,y)
            trainer.zero_grad()
            l.backward()
            trainer.step()
            total_loss += l.item()
            total_correct += (y_hat.argmax(dim=1) == y).sum().item()
        accuracy = total_correct / len(trainset)
        writer.add_scalar('Loss', total_loss,epoch)
        writer.add_scalar('Accuracy', accuracy,epoch)


def te_model(net,testloader,loss,trainer):
    # 评估模式
    net.eval()
    with torch.no_grad():
        print("Test:")
        total_loss, total_correct = 0, 0
        for X, y in teloader:
            y_hat = net(X)
            l = loss(y_hat, y)
            total_loss += l.item()
            total_correct += (y_hat.argmax(dim=1) == y).sum().item()
        print(f"loss:{total_loss:.4f},correct:{total_correct / len(teset):.4f}")

train_model(net, trainloader, loss, trainer,num_train_epochs)
te_model(net,teloader,loss,trainer)
writer.close()
```

损失函数

- L2 Loss:$l(y,y^{'})=\frac{1}{2}(y-y^{'})^2$
- L1 Loss:$l(y,y^{'})=|y-y^{'}|$
- Huber's Robust Loss
- MSE:均方差损失

数值稳定性

- 面对多层神经网络时，在求解梯度时会用到链式法则，这时候可能会出现梯度爆炸和梯度消失

- 梯度爆炸：
  - 使用relu作为激活函数。W^t会导致梯度爆炸

- 梯度消失：
  - 使用sigmoid函数，==红框内是梯度消失的罪魁祸首==

使得训练更加稳定(看不懂)

- 使用ResNet和LSTM激活函数
- 梯度裁剪
- 合理权重初始和激活函数
- 让每层的方差是一个常数，让均值为0

自定义层

- 在**init**方法里面定义输入输出维度，并用parameter定义权重和偏置
- 在forward里面进行线性变化

读写文件

- torch.save()可以保存张量，torch.load()可以读取张量
- 在torch.save()里面的参数使用net.state_dict()方法，将模型参数保存

- 使用新的模型名.load.state_dict(torch.load)方法将模型参数放进模型中

卷积

- 平移不变性和局部性
- v不受i，j影响，当a和b超出范围，参数为0

卷积层

- kernel：卷积核，可学习参数
- padding：输入层周围添加额外的行和列，(行数，列数)

- stride：行和列的移动步长，可以减小输出形状
- 多输入和多输出通道：
  - 多输入通道：彩色图片有RGB三个通道，转换为灰度会丢失信息
    - 每个通道都有一个卷积核(滤波器)，结果是所有通道卷积结果的和
  - 多输出通道：卷积核是四维的，一个输出通道和一个卷积核相乘，得出来的结果是多输出
- 原理：多输出通道意味着有多种不同的特征，会生成更多的特征图，得到图像更多的低级特征；多输入通道可以允许网络处理不同的输入特征，如RGB，某些四维的信息

池化

池化层（如最大池化或平均池化）的作用是进行下采样，主要目的是减少特征图的尺寸，降低数据的维度，同时保留重要的特征，减少计算量并防止过拟合。

最大池化

- 返回滑动窗口的最大值
- 没有可学习的参数
- 不会融合多输入通道，输出通道数=输入通道数

平均池化

- 返回滑动窗口的平均值

- 步幅大小和池化核大小相同

LeNet

- 输入层为32*32
- 卷积层：卷积核为5*5，输出为6 * 28 *28
- 平均池化层：池化核为5 * 5，输出为6* 14 *14
- 卷积层：卷积核为5 * 5，输出为16* 10 * 10
- 池化层：池化核为5 * 5，输出为16 *5 * 5
- 展平并使用两个全连接层，输出层分别为120与84
- 输出层维度为10

卷积层和全连接层后面要加激活函数，因为他们只能表示线性关系，会限制模型的表达能力

```
import torch
from torch import nn
import torchvision
from torch.utils import data
from torchvision import transforms
from torch.utils.tensorboard import SummaryWriter



class my(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Conv2d(1,6,kernel_size=5,padding=2),
                                 nn.Sigmoid(),
                                 nn.AvgPool2d(kernel_size=2,stride=2),
                                 nn.Conv2d(6,16,kernel_size=5),
                                 nn.Sigmoid(),
                                 nn.AvgPool2d(kernel_size=2,stride=2),
                                 nn.Flatten(),
                                 nn.Linear((16 * 5 * 5),120),
                                 nn.Sigmoid(),
                                 nn.Linear(120,84),
                                 nn.Sigmoid(),
                                 nn.Linear(84,10))
    def forward(self,X):
        return self.net(X)



trainset = torchvision.datasets.FashionMNIST("./data",train =
True,transform=transforms.ToTensor(),download=False)
trainloader = data.DataLoader(trainset,batch_size=256,shuffle=True)
teset = torchvision.datasets.FashionMNIST("./data",train =
False,transform=transforms.ToTensor(),download=False)
```

```
teloader = data.DataLoader(teset,batch_size=128,shuffle=True)


net = my()

writer = SummaryWriter('lenet')



loss = nn.CrossEntropyLoss()
trainer = torch.optim.Adam(net.parameters(),lr=0.001,betas=(0.9,0.999))

num_epochs = 10

for epoch in range(num_epochs):
    net.train()
    Loss = 0
    correct = 0
    for X,y in trainloader:
        y_hat = net(X)
        l = loss(y_hat,y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
        Loss+=l.item()
        correct += ((y_hat.argmax(dim=1)) == y).sum().item()
    accuracy = correct / len(trainset)
    writer.add_scalar('训练损失',Loss,epoch)
    writer.add_scalar('训练准确率', accuracy, epoch)


with torch.no_grad():
    Loss = 0
    correct = 0
    for X,y in teloader:
        net.eval()
        y_hat = net(X)
        l = loss(y_hat,y)
        Loss+=l.item()
        correct+=((y_hat.argmax(dim=1)) == y).sum().item()
print(f'Loss:{Loss:.4f},Accuracy:{correct / len(teset):.4f}')
```

AlexNet

- 主要改进：丢弃法，ReLu，MaxPooling
- 初级特征提取：在卷积网络的早期层，更多的输出通道捕捉具体的低级特征。
- 高级特征抽象：在网络的后期层，输出通道逐渐减少，以便将这些具体特征组合成更高级的抽象特征。
- 最终决策：全连接层将高维特征映射到最终的决策空间，以完成分类或回归任务。

VGG

更大更深的AlexNet，用GPU才跑得动

- 3*3卷积(padding:1,n层，m通道)
- 2*2最大池化层(stride:2)

VGG架构

- 多个VGG块后接全连接层
- 不同次数的重复块得到不同的架构VGG-16，VGG-19

NiN

NiN块

- 一个卷积层后跟两个全连接层(实际上是卷积层，1*1的卷积核)
    - 步幅1，无填充，输出形状跟卷积层输出一样
    - 起到全连接层的作用

NiN架构

- 无全连接层
- 交替使用NiN块和步幅为2的最大池化层
- 使用全局平均池化层得到输出
    - 全局平均池化指的是一个特征层输出一个类别

正则化层

Batch Normalization层

一般放在卷积层，全连接层之后，激活函数之前

出现的问题

- 固定小批量的均值和方差，控制均值和方差大小，一般放在激活函数前，学习出适合的偏移和缩放
- 对全连接层，作用在特征维；对于卷积层，作用在通道维
- 使用的方法：nn.BatchNorm2d(维度)

Dropout层

- 类似于正则化，可以防止过拟合
- 在神经网络和全连接层中有使用，其是让中间结果随机置零，得到子全连接层或子神经网络，得到
- 较小规模的映射，取他们的平均值从而优化模型的训练
- 一般放在卷积层或全连接层之后，可以增强模型的泛化能力

Resnet

在传统的卷积网络中，输入x 会直接通过卷积和激活函数被处理为H(x)，然后传递到下一层。在 ResNet 中的残差块，我们有两条路径：主路径：输入 x 经过卷积层、激活函数（ReLU）和卷积层的处理，得到输出F(x)。跳跃连接：输入 x 没有经过卷积处理，直接跳跃传递到输出部分。最后的输出不是 H(x)，而是：H(x)=F(x)+x 也就是说，最终输出是经过卷积处理的特征F(x) 加上输入x本身。

某一层的输入x可能已经捕捉到了大部分的信息，只需要微调一下。通过残差学习，网络可以专注于学会那些小的调整F(x)，而不是重新构造整个映射

个人理解

x为跳过卷积层的输出，F(x)为经过卷积层的输出

$$H(x) = F(x) + x$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H(x)} * (\frac{\partial H(x)}{\partial F(x)} + \frac{\partial H(x)}{\partial x}) = \frac{\partial L}{\partial H(x)} * (\frac{\partial H(x)}{\partial F(x)} + 1)$$

而加法可以保留输出层的梯度，从而防止梯度消失

ResNet架构

数据增强

可以使用对一张图片进行不同变化得到多张图片，获得更多的训练样本，提高模型的泛化能力

Transoforms关于数据增强的用法

- `transforms.RandomVerticalFilp(p=0.5)`：以指定概率p随机垂直翻转图片
- `transforms.RandAffine(degrees,translate=None,scale=None.shear=None)`：对图像进行仿射变换(包括旋转、平移、缩放和横切)
- `transforms.Grayscale(num_output_channels=1)`：将图片转换为灰度图像，`num_output_channels` 可以指定输出通道数。
- `transforms.RandomPerspective(distortion_scale=0.5, p=0.5)`：随机应用透视变换，`distortion_scale` 控制变形的程度，`p` 是变换应用的概率。
- `transforms.RandomErasing(p=0.5, scale=(0.02, 0.33), ratio=(0.3, 3.3))`：随机抹除部分图像，用于增强模型的鲁棒性。
- `transforms.RandomCrop(size, padding=None, pad_if_needed=False)`：随机裁剪图片，裁剪后的尺寸为 `size`，可以选择在裁剪前进行填充。
- `transforms.FiveCrop(size)`：对图片进行上下左右和中心裁剪，返回 5 张裁剪后的图片。
- `transforms.TenCrop(size, vertical_flip=False)`：返回上下左右和中心裁剪的图片，且每个裁剪再水平翻转一次，得到 10 张图片。
- `transforms.Lambda(lambda_func)`：自定义的变换操作，可以使用 lambda 函数定义自己的变换逻辑。
- `transforms.RandomInvert(p=0.5)`：以指定概率 `p` 随机反转图像的颜色。
- `transforms.RandomPosterize(bits)`：以随机方式降低图片的色深，将每个颜色通道的位数减少到 `bits`。
- `transforms.RandomSolarize(threshold)`：将像素值高于 `threshold` 的部分进行反转。
- `transforms.GaussianBlur(kernel_size, sigma=(0.1, 2.0))`：对图像应用高斯模糊，`kernel_size` 是模糊核的大小，`sigma` 是模糊程度。
- `transforms.AutoAugment(policy)`：自动数据增强，支持多种增强策略，如 `AutoAugmentPolicy.IMAGENET`，`AutoAugmentPolicy.CIFAR10` 等。
- `transforms.InterpolationMode`：用于图像缩放时的插值模式，可以选择不同的插值方式，如 `BILINEAR`、`NEAREST` 等。

微调

网络架构

- 一个神经网络一般可以分成两块
  - 特征抽取将原始像素变成容易线性分割的特征
  - 线性分类器来做分类(一般为softmax回归来做分类)

微调原理

- 使用预训练模型在源数据集上进行训练
- 源数据集通常远远大于目标数据集，以保证预训练模型在目标数据集上的表现较好
- 将预训练模型的权重赋复制到自己的模型进行权重初始化

- 输出层需要随机初始化，因为最后的输出类别不一样

训练

- 是一个目标数据集上的正常训练任务，但使用更强的正则化，因为模型已经学习的差不多了，要防止过拟合
- 使用更小的学习率
- 使用更少的数据迭代
- 源数据集远复杂于目标数据，通常微调效果更好

冻结

- 神经网络通常学习有层次的特征表示
  - 低层次的特征更加通用，因为学习的是底层的特征
  - 高层次的特征更跟数据集相关
- 可以固定底部一些层的参数，不参与更新，从而缩小模型

代码演示

列表推导式

```
# <expression>：这是每次循环中需要被计算和添加到最终列表的值。
# for <item> in <iterable>：遍历一个可迭代对象 <iterable>，每次迭代中 <item> 代表当前项。
# if <condition>：条件判断，只有满足条件的项才会被添加到最终列表。
[<expression> for <item> in <iterable> if <condition>]

# 根据name来分组
# net.named_parameters()返回一个(name,parameters)元组，name是参数名称，parameters是参数的值
arams_lx =[param for name,param in net.named_parameters()if name not
in['fc.weight','fc.bias']]
```

分组

```
trainer = torch.optim.SGD(
    [
        {'params': params_lx},                          # 第一组参数：模型中除全连接层外的参数
        {'params': net.fc.parameters(), 'lr': learning_rate * 10}  # 第二组参数：全连接层的参
数，学习率加大10倍
    ],
    lr=learning_rate,    # 默认学习率，用于第一组参数
    weight_decay=0.001  # 权重衰减，用于防止过拟合
)
```

微调代码

```
import torch
import torchvision
from torch import nn

# 使用预训练好的resnet18
net = torchvision.models.resnet18(weights = torchvision.models.ResNet18_Weights.DEFAULT)
# .fc是全连接层，重新定义输出维度
net.fc = nn.Linear(net.fc.in_features,2)
# 对最后一层随机初始化
nn.init.xavier_uniform_(net.fc.weight)

def train(net,learning_rate,batch_size=128,epochs=5,param_group=True):
    loss = nn.CrossEntropyLoss(reduction='none')
    if param_group:
        params_1x =[param for name,param in net.named_parameters()if name not
in['fc.weight','fc.bias']]
        trainer = torch.optim.SGD([{'params':params_1x},
{'params':net.fc.parameters(),'lr':learning_rate*10}],lr = learning_rate,weight_decay=0.001)
    else:
        trainer = torch.optim.SGD(net.parameters(),lr = learning_rate,weight_decay=0.01)
```

目标检测

边缘框

- 一个边缘框可以通过四个数字定义
  - (左上x，左上y，右下x，右下y)
  - (左上x，左上y，宽高)

目标检测数据集

- 每行表示一个物体
  - 图片文件名，物体类别，边缘框
- 上述会和图片放在同一文件夹里作为数据集

锚框

- 一类目标检测算法
  - 提出多个被称为锚框的区域(边缘框)
  - 预测每个锚框里是否含有关注的物体
  - 如果是，预测从这个锚框到真实边缘框的偏移

IoU-交互比

- IoU用来计算两个框之间的相似度
  - 0表示无重叠，1表示重合
- 这是Jcaquard指数的一个特殊情况
  - 给定两个集合A和B

    $J(A,B)=\frac{|A\cap B|}{|A\cup B|}$

赋予锚框标号

- 每个锚框是一个训练样本
- 将每个锚框，要么标注成背景，要么关联上一个真实边缘框
- 我们可能会产生大量的锚框
  - 这个导致大量的负类样本
  - 同时在训练的过程中我们会一张一张读入图片，因为锚框数量太多

- 横轴是边缘框即物体类别的数量
- 纵轴是锚框数量
- 计算每个锚框和边缘框的IoU值
- 选出最大值，并去除掉这一行一列，将对应锚框和边缘框关联起来，赋予锚框标号，**如图将锚框2和边缘框3关联起来**

使用非极大值抑制(NMS)输出

- 每个锚框预测一个边缘框
- NMS可以合并相似的预测
  - 选中是非背景类的最大预测值
  - 去掉所有其它的锚框和它的IoU值大于θ的预测
  - 重复上述过程直到所有预测要么被选中，要么被去掉

锚框的代码实现

理论推导

假设输入图像的高度为h,宽度为w。我们以图像的每个像素为中心生成不同形状的锚框，比例(长宽缩放比)为$s\in(0,1]$（即锚框占整张图片的大小）,宽高比为$r>0$（这里的r是归一化后的宽高比）。

推导过程

> 易知图片面积大小为wh，锚框面积大小即为whs^2。
>
> $$r=\frac{w_a^{\backprime}}{h_a^{\backprime}}=\frac{\frac{w_a}{w}}{\frac{h_a}{h}},$$
>
> 可以推出$\frac{w_a}{h_a}=\frac{r*w}{h}$。
>
> 可以得到锚框的实际宽高比为$\frac{w_a}{h_a}=\frac{r*w}{h}=\frac{w}{h}\frac{w_a^{\backprime}}{h_a^{\backprime}}$。
>
> 联立方程组得到$w_a=s*w*\sqrt{r},h_a=\frac{s*h}{\sqrt{r}}$
>
> 归一化得：$w_a^{\backprime}=s*\sqrt{r},h_a^{\backprime}=\frac{s}{\sqrt{r}}$

实现锚框原理

- **当给定一系列比例$s_1,\cdots,s_n$和一系列宽高比$r_1,\cdots,r_m$。当使用这些比例和长宽比的所有组合以每个像素为中心时，输入图像将总共有whnm个锚框（==wh为像素的总个数，nm为一个像素下的锚框个数==）。**
- 但这样计算的复杂度会升高，因此我们只考虑**以下组合**
- $(s_1,r_1),(s_1,r_2),\ldots,(s_1,r_m),(s_2,r_1),(s_3,r_1),\ldots,(s_n,r_1)$
- 基于上述组合，一个像素生成的锚框数量为==$n+m-1$==，对于整个输入图像，将生成==$wh(n+m-1)$==个锚框

物体检测算法

R-CNN

- 使用启发式搜索算法来选择锚框
- 使用预训练模型来对每个锚框抽取特征

自然语言处理

序列模型

==序列数据==:**实际中很多数据是有时序结构的，即数据会随着时间变化而变化**

统计工具

- 在时间t观察到,那么得到T个不独立的随机变量$(x_1,\cdots,x_T)\sim p(x)$
- 使用条件概率展开
  - $p(a,b)=p(a)p(b|a)=p(b)p(a|b)$
- 对于随机变量$(x_1,\cdots,x_T)\sim p(x)$
  - $p(x)=p(x_1)p(x_2|x_1)\cdot p(x_3|x_1,x_2)\cdot\ldots p(x_T|x_1,\cdots x_{T-1})$
- 对条件概率建模
  - 自回归模型:
    - $p(x_t|x_1,\cdots x_{t-1})=p(x_t|f(x_1,\cdots x_{t-1}))$

- 马尔科夫假设：当前数据只跟τ个过去数据点相关
  - $p(x_t|x_1,\cdots x_{t-1}) = p(x_t|x_{t-\tau},\cdots x_{t-1}) = p(x_t|f(x_{t-\tau},\cdots x_{t-1}))$
- 潜变量模型：引入潜变量h_t来表示过去信息$h_t = f(x_1,\cdots,x_{t-1})$,这样$x_t = p(x_t|h_t)$，要注意的是$h_t$和前一个h以及前一个x和当前的x相关
  -

使用一个正弦函数加上高斯噪音来作预测，随着预测的步长增加，效果会变差

```
import torch
from torch import nn
from torch.utils import data
import matplotlib.pyplot as plt

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)


# 数据生成
T = 1000
time = torch.arange(1,T+1,dtype = torch.float32)
x = torch.sin(0.01*time) + torch.normal(0,0.2,(T,))


import matplotlib.pyplot as plt

# 绘制原始数据
plt.figure(figsize=(12, 6))
plt.plot(time.numpy(), x.numpy(), label='Noisy Sine Wave', color='b')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Original Data: Noisy Sine Wave')
plt.legend()
plt.grid(True)



tau = 4
features = torch.zeros((T-tau,tau))
for i in range(tau):
    features[:,i] = x[i:T-tau+i]
labels = x[tau:].reshape((-1,1))

batch_size,n_train = 16,600
dataset = data.TensorDataset(features[:n_train],labels[:n_train])
train_iter = data.DataLoader(dataset,batch_size,shuffle=True)


net = nn.Sequential(nn.Linear(4,10),nn.ReLU(),nn.Linear(10,1))
net.apply(init_weights)
```

```
loss = nn.MSELoss()


trainer = torch.optim.Adam(net.parameters(),lr=0.01)

epochs = 5
for epoch in range(epochs):
    Loss = 0
    sample = 0
    for X,y in train_iter:
        trainer.zero_grad()
        l = loss(net(X),y)
        l.backward()
        trainer.step()
        Loss+= l.item() * X.size(0)
        sample += X.size(0)
    print(f'epoch:{epoch+1}, loss:{Loss/sample:.4f}')
# 单步预测
onestep_preds = net(features).detach().numpy()

plt.plot(time[tau:].numpy(), onestep_preds, label='One-step Predictions', color='r',
linestyle='--')
plt.xlabel('time')
plt.ylabel('x')
plt.legend()
plt.grid(True)
plt.show()
```

文本预处理

##