同济大学控制科学与工程系

机器学习课程

实验报告

实验名称　神经网络的简单实现

开课学期　2021–2022 学年第 2 学期

学生姓名　刘一凡　学　号　1952732

神经网络的简单实现

简介

本实验目标完成一个基础的神经网络“框架”，支持基本神经网络需要使用到的算法，包括线性、卷积层，激活函数，损失函数等。实验代码参考了当前市面流行的框架，如：Pytorch，Tensorflow，PaddlePaddle 等。

一、 算法实现

从训练流程讨论，主要包括前向传播，损失函数计算，反向传播，优化器更新参数。

1. 网络设置组件介绍

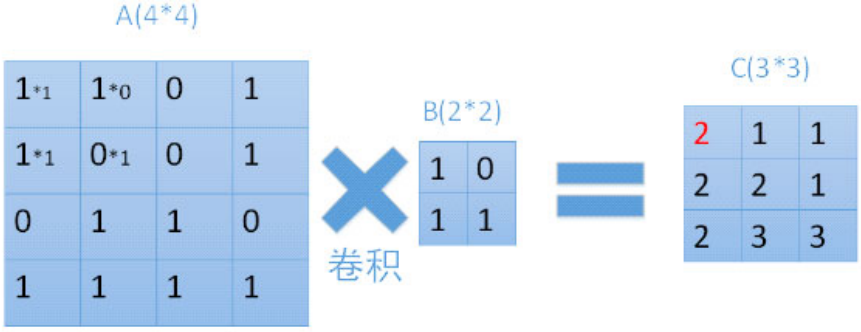
网络设置包括卷积层、线性层、激活函数等，在本实验中大致体现为图一。

```
self.ConvUnit = Jh.Unit(
    Conv2D(1, 4, kernel_size=(3, 3), padding=1),
    MaxPooling((2, 2)),
    BatchNormalization((4, 14, 14), 4),
    Relu(),
    Conv2D(4, 4, kernel_size=(3, 3), padding=1, stride=(1, 1)),
    MaxPooling((2, 2)),
    Flatten(),
)
```

图一 前向传播组成示例

① 卷积层

卷积层的前向传播如图二所示，在本实验中只实现了 Conv2D。在初始化时可以输入进、出通道数，卷积核大小，以及填充的参数设置。



图二 Conv2D 前向传播示意图

由于卷积层计算量占据了总资源消耗的一大半，在此针对卷积运算进行了一定的优化。输入数据形状为(N, C, H, W)，卷积核形状为$(C_, C, KH, KW)$，为满足上述运算，并尽量使用点乘计算可以将输入数据重组为(N, C, OH, OW, KH, KW)。例如，当卷积核为2×2，$stride=1$时：

$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}\rightarrow\begin{bmatrix}\begin{bmatrix}1&2\\4&5\end{bmatrix}&\begin{bmatrix}2&3\\5&6\end{bmatrix}\\\begin{bmatrix}4&5\\7&8\end{bmatrix}&\begin{bmatrix}5&6\\8&9\end{bmatrix}\end{bmatrix}$$

卷积层的反向传播需要计算ω, b, X的梯度。计算偏置b的梯度直接在传入梯度的最后两个维度求和即可(再除以$batch_size$)。计算X的梯度可以转变为传入梯度与ω旋转180^o后卷积。在进行卷积之前需要对传入梯度进行简单的填充。按照梯度求导法则可以得到在梯度矩阵各元素之间需要插入步长减一个 0，同时在外围补 0 保证输出维度满足传播要求。示例如下(步长 2)：

$$\begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&\delta_{1,1}&0&\delta_{1,2}&0&0\\0&0&0&0&0&0&0\\0&0&\delta_{2,1}&0&\delta_{2,2}&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix}\ conv\ \begin{bmatrix}k_{3,3}&k_{3,2}&k_{3,1}\\k_{2,3}&k_{2,2}&k_{2,1}\\k_{1,3}&k_{1,2}&k_{1,1}\end{bmatrix}$$

计算ω的梯度，同样也只需要在传入梯度之间补零即可，示意如下：

$$\begin{bmatrix}x_{1,1}&x_{1,2}&x_{1,3}&x_{1,4}&x_{1,5}\\x_{2,1}&x_{2,2}&x_{2,3}&x_{2,4}&x_{2,5}\\x_{3,1}&x_{3,2}&x_{3,3}&x_{3,4}&x_{3,5}\\x_{4,1}&x_{4,2}&x_{4,3}&x_{4,4}&x_{4,5}\\x_{5,1}&x_{5,2}&x_{5,3}&x_{5,4}&x_{5,5}\end{bmatrix}\ conv\ \begin{bmatrix}\delta_{1,1}&0&\delta_{1,2}\\0&0&0\\\delta_{2,1}&0&\delta_{2,2}\end{bmatrix}$$

如此，可以将卷积层的运算全部转变为卷积，而任何卷积的运算又可以使用之前提到的重构方法快速计算。

② 线性层

线性层的实现相较于卷积层更为简单，前向传播使用矩阵乘法即可实现。反向传播公式如图三所示：

```
# 参数梯度
# (inshape, outshape) = (inshape, m) @ (m, outshape)
self.W.gradient = self.mem["X"].T @ grad
self.b.gradient = grad.sum(axis=0)
# 数据梯度 (m,inshape) = (m,outshape) @ (outshape, inshape)
out_grad = grad @ W.T
```

图三 线性层反向传播

③ 池化（以最大池化为例）

最大池化的原理很容易理解，正向传播时取核范围内的最大值，反向传播时只有等于最大值的元素对应位置才有梯度继承，其他位置梯度为0.实现时若采用简单的循环将耗费大量资源，甚至远超卷积。在此我的解决办法是正向传播时生成对应最大值的零一矩阵，最大值位置取1，其他为0.反向传播时只要将传入梯度矩阵扩充再乘以该零一矩阵就可以得到梯度输出了。

④ 激活函数（以Relu为例）

类似与最大池化，正向传播支取大于零的部分，其他数据设为0.反向传播只有输入数据X大与0对应的梯度才得以保留。

⑤ 批量归一化（$BatchNormalization$）

对于前向传播到 BN 层的向量X，以此执行以下转化，其中γ、β为一个 BN 层神经元需要学习的两个参数。向量进行标准白化处理后分布更加稳定，随后进行的平移变换操作更是通过学习参数的形式加强了数据分布的稳定性。

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, y_i = \gamma \hat{x}_i + \beta$$

通常再卷积层后链接BN层再是激活函数。在此参数即为γ、β。BN层可以将数据控制在激活函数的非饱和区，可以有效地防止梯度消失与梯度爆炸。注意在进行预测是要使用训练时记录的参数值。

2. 损失函数与优化器

① 交叉熵损失函数

常见的损失函数有交叉熵损失函数和均方误差损失函数。在此以交叉熵损失函数为代表。

交叉熵损失函数把模型的输出值当成一个离散随机变量的分布列。设模型的输出为：$\hat{Y} = f(X)$，其中$f(X)$表示模型。$\hat{Y}$是一个 m X n 矩阵，如下所示：

$$\begin{bmatrix} \hat{y}_{11} & \hat{y}_{12} & \cdots & \hat{y}_{1n} \\ \hat{y}_{21} & \hat{y}_{22} & \cdots & \hat{y}_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ \hat{y}_{m1} & \hat{y}_{m2} & \cdots & \hat{y}_{mn} \end{bmatrix}$$

把这个矩阵的第i行记为$\hat{y}_i$,它是一个R^{1Xn}向量， 它的第 j 个元素记为$\hat{y}_{ij}$。

交叉熵损失函数要求$\hat{y}_i$具有如下性质：

$$0 <= \hat{y}_{ij} <= 1$$
$$\sum_{j=1}^{n} \hat{y}_{ij} = 1, \quad n = 2, 3, \ldots$$

模型有时候并不会保证输出值有这些性质， 这时损失函数要把$\hat{y}_i$转换成一个分布列:$\hat{p}_i$，转换函数的定义如下：

$$S_i = \sum_{j=1}^{n} e^{\hat{y}_{ij}}$$
$$\hat{p}_{ij} = \frac{e^{\hat{y}_{ij}}}{S_i}$$

设数据x_i的类别标签为$y_i \in R^{1Xn}$. 如果x_i的真实类别t，y_i满足：

$$若 j = t： \ y_{ij} \ = \ 1$$
$$若 j \neq t： \ y_{ij} = 0$$

使用的是 one-hot 编码。交叉熵损失函数的定义为：

$$J_i = \frac{1}{m} \sum_{j=1}^{n} -y_{ij} ln(\hat{p}_{ij})$$

② Adam 优化器

最简单的梯度下降法很容易下如局部最优，由此产生了优化器。常见的优化器有$Momentum$，$Adagrad$，以及常用的集大成者$Adam$。

若规定，动量：v_t, $v_0 = 0$；梯度：g_t；梯度累加变量n_t, $n_0 = 0$；超参数：β_1, β_2。有如下公式：

动量更新：

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

对g_t^2加权平均：

$$n_t = \beta_2 n_{t-1} + (1 - \beta_2) g_t^2$$

在t时刻：

$$v_t = (1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i = 1 - \beta_1^t$$

偏差修正：当t较小时，过去各时间步小批量随机梯度权值之和会较小。例如，当$\beta_1 = 0.9$时， $v_1 = 0.1 g_1$ 。为了消除这样的影响，对于任意时间步t，我们可以将 v_t再除以$1 - \beta_1^t$，从而使过去各时间步g_t权值和为 1.

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t}$$
$$\hat{n}_t = \frac{n_t}{1-\beta_2^t}$$

那么(η 是学习率，ϵ 防止分母为 0)：

$$g'_t = \frac{\eta \hat{v}_t}{\sqrt{\hat{n}_t + \epsilon}}$$

最后：

$$x_t = x_{t-1} - g'_t$$

二、 实验结果

实验采用上述网络框架实现经典的 LeNet5 网络，网络定义如图四所示。网络使用了 2D 卷积，包括卷积核大小设置，步长设置，以及 padding。还使用了 Relu 函数，平均池化，拉平层，线性层等。

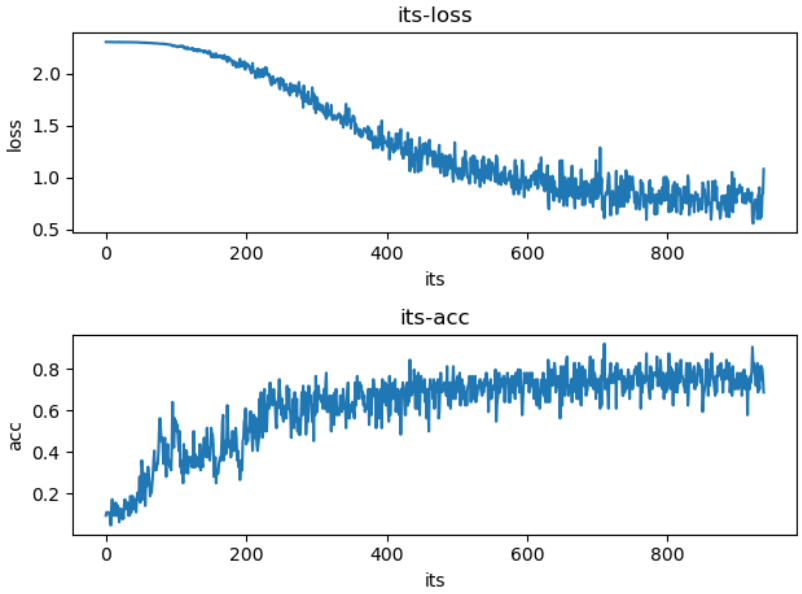
```
self.ConvUnit = Jh.Unit(
    Conv2D(1, 6, kernel_size=(5, 5), stride=1, padding=(2, 2)),
    Relu(),
    AvgPooling((2, 2)),
    Conv2D(6, 16, kernel_size=(5, 5), stride=1),
    Relu(),
    AvgPooling((2, 2)),
    Conv2D(16, 120, kernel_size=(5, 5), stride=1),
    Relu(),
    Flatten(),
)
self.LinearUint = Jh.Unit(
    Linear(120, 84),
    Relu(),
    Linear(84, class_dim),
)
```

图四 LeNet5 网络定义

使用上述 LeNet5 网络对 MINST 数据集进行训练和预测，故在工程中导入了 Tensorflow 加载该数据集，若无此需要可以不导入 tf。训练过程如图五，同时 Model 类有简单的绘图函数，可以展示 acc 和 loss 的变化情况，如图六。

```
epoch: 1/2:  26%|███       | 245/938 [01:50<04:39,  2.48it/s, loss=1.930951, acc=0.515625]
```

图五 训练过程截图



图六 训练过程中某一个 batch 的 acc 和 loss 曲线

在训练完成后调用 `model.save(path)`可以保存该模型至指定位置。训练完成后可以调用 `pre_y = model.predict(X_test)` 以及 `acc = computeAccuracy(pre_y, Y_test)`计算模型准确度。注意在预测阶段需要使用 `model.eval()`，以此将 BN 和 dropout 等环节设置为预测模式。

实验结果如图七，打印了一部分预测与真实结果，同时计算了模型的 ACC。

```
预测数字: [0, 6, 9, 0, 1, 5, 9, 7, 3, 4]
真实数字: [0 6 9 0 1 5 9 7 3 4]
acc= 0.8331
```

图七 测试结果

三、 实验总结

本次实验使我深刻理解了机器学习，特别是神经网络的算法原理以及相关的数学知识。让我逐渐摆脱了原来只会调包的学习习惯，也正是这次亲手从底层写算法代码给了我许多启发。我认识到之前自然而然的操作背后所隐藏的原因与意义，同时感叹当前市面上提供的网络框架包的高度包容性与严谨性。

实验期间我记录了每天的想法与实现事项，附于工程目录 docs/Log.md 文件中。