人工智能中的编程 Final Task3

苏王捷 *

December 24, 2024

1 Introduction

Task3 综合了前几次作业的成果，实现从零开始的神经网络：使用自己的 minitorch 库及自动微分，搭建了简单的 MLP 分类器和 CNN 网络，进行 MNIST 数据集的分类任务。

具体实现见 TinyTensor 文件夹。

2 Implementation

2.1 前五次作业的整合改进

使用 HW1 和 HW2 中实现的 cuda 代码，完成 Tensor 类和 7 个神经网络的层 (见 src 文件夹)；

使用 HW3 中的代码，通过 setup.py 和 pybind 构建 python 动态链接库以调用 cpp 和 cu 代码 (见 src 中的 bind_Layer.cpp、bind_Tensor.cpp 和主目录下的 setup.py)；

仿照 HW4 中可进行自动微分的 Tensor 类，在 python 端再搭建了一个 Tensor 类用于实现自动微分。将 HW4 中的 Tensor 类完全复制，加上了以自己的 Tensor 类作为底层储存的构建方法，实现自己的 Tensor 在 python 端的调用和自动微分功能。

同样仿照 HW4 中的算子，将 7 个神经网络层设置为 5 个基本算子 (Relu, FC, Conv, MaxPool 和 Softmaxcrossentropyloss)，继承自 TensorOp 类，拥有前向计算和反向传播功能，并在其功能中调用绑定的动态链接库接口，使用 cuda 代码实现 gpu 并行计算 (见 optimizer 文件夹中的 operators.py 文件), 在神经网络中调用对应函数即可实现神经网络运算过程中自动微分图的搭建及之后的反向传播；其中 Relu，FC，Conv 和 MaxPool 用于神经网络中间过程，Softmaxcrossentropyloss 用于计算 loss 和 error，并作为反向传播的起始层返回第一个梯度；

使用 HW4 中实现的自动微分代码，用于构建自动微分图 (见 optimizer 文件夹中的 autodiff.py 文件)；

其余 basic_operator.py 和 device.py 直接使用 HW4 中的代码；

使用 HW5 中实现的 SGD 和 Adam 优化器进行优化，对其结构稍加改变即可 (见 optimizer 文件夹中的 mnist_cnn.py 或 mnist_mlp.py 中的 SGD 和 Adam 方法)。

*College of Engineering, Peking University, wsu0605@stu.pku.edu.cn

2.2 分类任务

2.2.1 MLP

具体代码见 mnist_mlp.py 文件。

使用 torchvision 的 dataloader 提取数据并进行预处理展平，获得可进行前向计算的 Tensor；

搭建了一个两层线性层的 MLP 分类器，将各层使用的矩阵设置为神经网络的权重，并使用优化器进行训练和权重更新，每个 Epoch 结束后，在整个训练集上计算 loss 和 error，在整个测试集上也计算 loss 和 error，输出结果。

使用 SGD 优化器和默认参数，训练 10 个 Epoch，可以达到约 95% 的测试集准确率，结果见下图1, 整个过程约需要 10 分钟。

```
(code_in_ai) E:\PKU\code_in_ai\homework\Final\Task3\Tinytensor\optimizer>python mnist_mlp.py
| Epoch | Train Loss | Train Err | Test Loss | Test Err |
|     1 |    0.22337 |   0.08235 |   0.22461 |  0.08240 |
|     2 |    0.20536 |   0.07562 |   0.20523 |  0.07575 |
|     3 |    0.19474 |   0.06950 |   0.19398 |  0.06960 |
|     4 |    0.18662 |   0.06504 |   0.18539 |  0.06475 |
|     5 |    0.18018 |   0.06175 |   0.17860 |  0.06118 |
|     6 |    0.17494 |   0.05933 |   0.17309 |  0.05838 |
|     7 |    0.17060 |   0.05748 |   0.16852 |  0.05623 |
|     8 |    0.16691 |   0.05599 |   0.16465 |  0.05474 |
|     9 |    0.16375 |   0.05482 |   0.16133 |  0.05350 |
|    10 |    0.16098 |   0.05374 |   0.15843 |  0.05253 |
Accuracy after train overall: 94.75%
```

Figure 1: Results MLP

2.2.2 CNN

具体代码见 mnist_cnn.py 文件。

使用 torchvision 的 dataloader 提取数据并获得可进行前向计算的 Tensor；

搭建了一个含有两个卷积层和池化层及两个线性层的 CNN 网络，将各层使用的矩阵设置为神经网络的权重，并使用优化器进行训练和权重更新，每个 Epoch 结束后，在整个训练集上计算 loss 和 error，在整个测试集上也计算 loss 和 error，输出结果。

使用 SGD 优化器和默认参数，训练 10 个 Epoch，可以达到约 96% 的测试集准确率，结果见下图2, 整个过程约需要 25 分钟。

```
(code_in_ai) E:\PKU\code_in_ai\homework\Final\Task3\Tinytensor\optimizer>python mnist_cnn.py
| Epoch | Train Loss | Train Err | Test Loss | Test Err |
|     1 |    0.20955 |   0.06598 |   0.20893 |  0.06660 |
|     2 |    0.19470 |   0.05736 |   0.19390 |  0.05811 |
|     3 |    0.18454 |   0.05792 |   0.18365 |  0.05790 |
|     4 |    0.18001 |   0.05576 |   0.17906 |  0.05562 |
|     5 |    0.16662 |   0.05297 |   0.16544 |  0.05256 |
|     6 |    0.15198 |   0.05188 |   0.15047 |  0.05132 |
|     7 |    0.13550 |   0.04353 |   0.13344 |  0.04322 |
|     8 |    0.12713 |   0.04179 |   0.12502 |  0.04148 |
|     9 |    0.11819 |   0.04071 |   0.11611 |  0.04026 |
|    10 |    0.11335 |   0.03957 |   0.11142 |  0.03920 |
Accuracy after train overall: 96.08%
```

Figure 2: Results CNN

3 代码运行及结果复现

在提交的压缩包中，我包括了已经构建好的 python 动态链接库；如果不能成功运行，请尝试如下构建过程：

1. 删除整个 build, myLayer.egg-info, myTensor.egg-info 文件夹及 myLayer.cp310-win_amd64.pyd 和 myTensor.cp310-win_amd64.pyd 包；

2. 在 setup.py 中名称为 myLayer 的部分中，找到 CUDAExtension 项，修改其中的 library_dirs 和 include_dirs，将其指定为 cublas 库的路径和 cublas 头文件的路径；

3. 在 TinyTensor 文件夹目录下，运行 *python setup.py develop* 命令，构建动态链接库。

代码文件在 optimizer 文件夹中。在 optimizer 文件夹目录下，运行 *python mnist_mlp.py* 命令，即可使用 MLP 进行分类；运行 *python mnist_cnn.py* 命令，即可使用 CNN 进行分类。结果均会显示在命令行中。

我在本地跑得的结果保存在了 Task3 的 results 文件夹中。