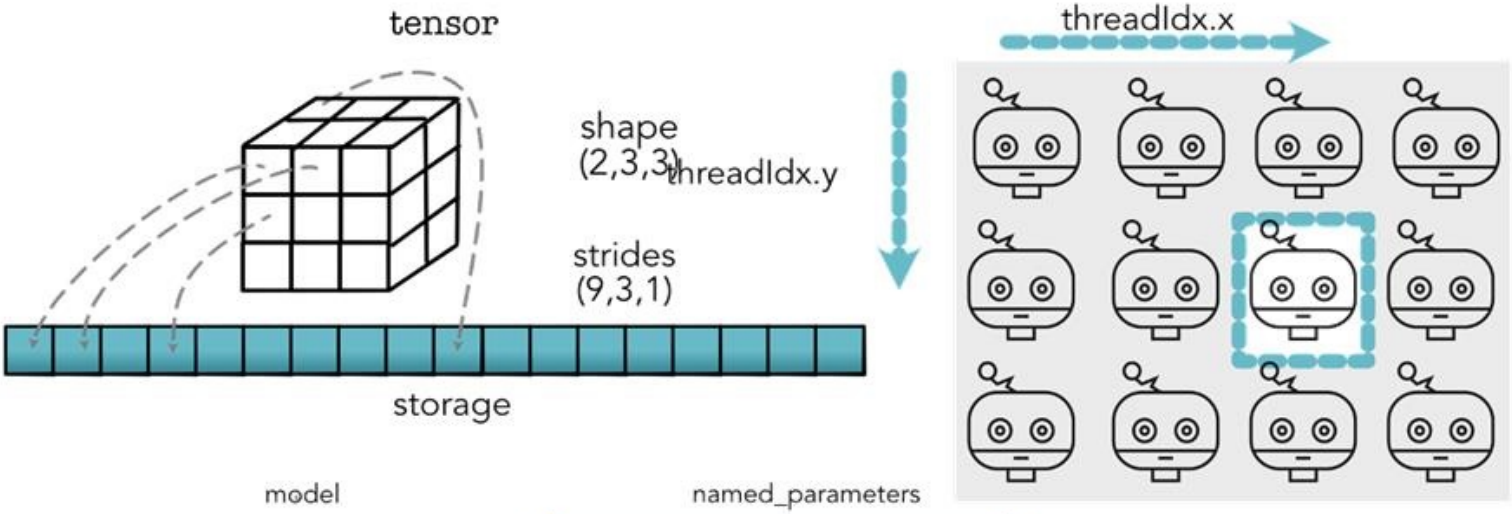
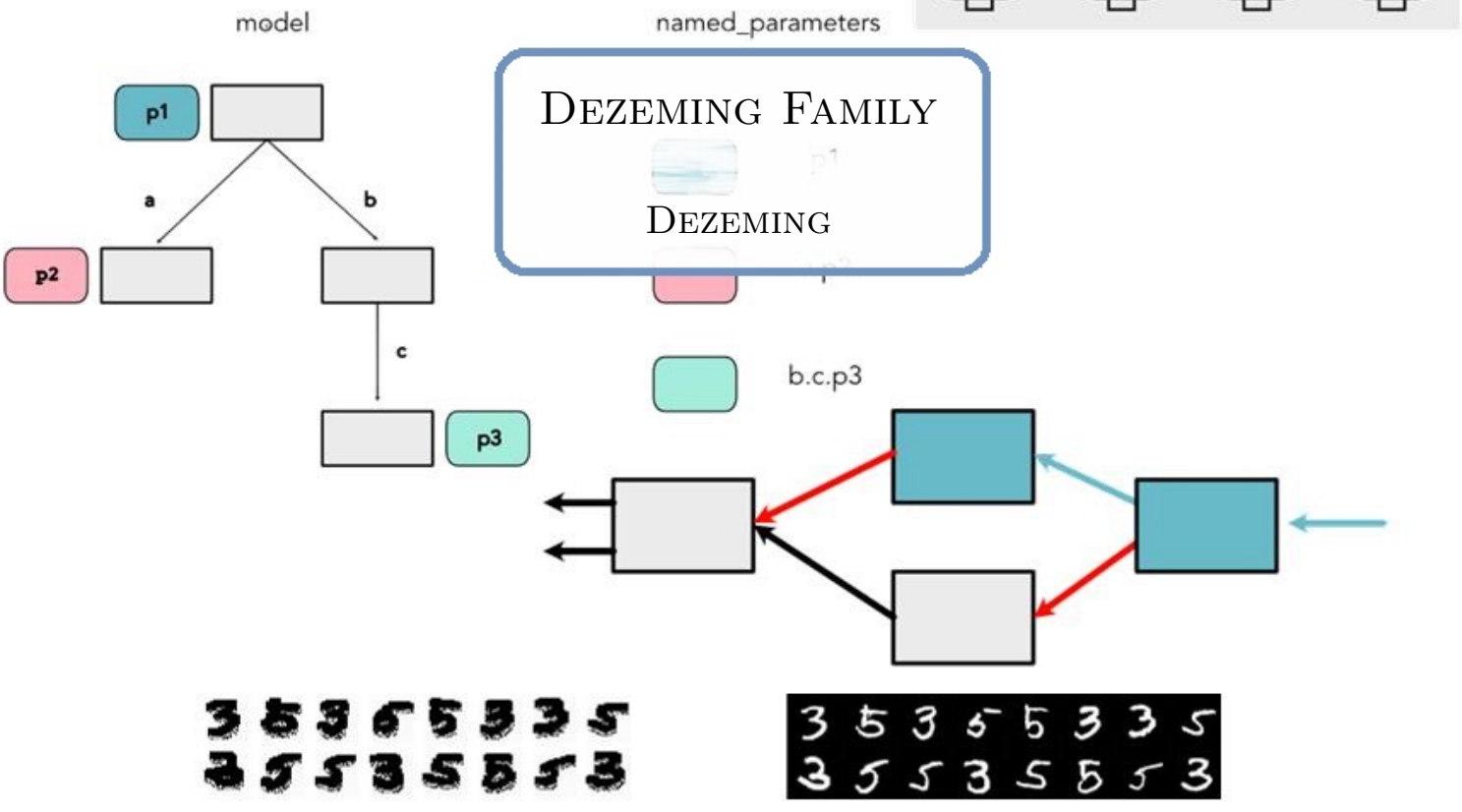
minitorch

MiniTorch 学习全攻略





Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列书，可以从我们的网站 *[https://dezeming.top/]* 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

0.1 本书前言

MiniTorch[1][2] 是康奈尔大学副教授（pre-tenure）Alexander Rush 构建的一个 DIY 教学库，面向希望了解深度学习系统内部概念的机器学习工程师。它是对 torch api 的纯 Python 重新实现，设计简单、易于阅读、容易进行测试和扩充。miniTorch 最终的库可以运行 Torch 代码。

该项目是为康奈尔理工学院的“机器学习工程”课程开发的，这是一门硕士课程，涵盖了训练、调整、调试、可视化和部署机器学习系统中的系统级问题。

涉及的知识：

- 基本神经网络和模块
- 标量的自动微分
- 张量、视图 (Views) 和步幅 (Strides)
- 并行张量运算
- NUMBA 中的 GPU/CUDA 编程
- 卷积和池化
- 高级神经网络函数

由于该课程目前没有线上公开课，所以我们国内玩家也不能非常细致地了解方方面面。本文的内容是我在学习过程中，把知识点进行整理和归纳总结，进行详细地记录，并对一些概念和方法提出自己的思考过程。

本文开始写于 2022 年 1 月 13 日，在我写下前言时，由于科研任务繁重，还有不少杂事，我也不知道自己要用多久才能完本，这是一次全新的挑战。

学习本课程需要的基本知识：

- 基本的神经网络原理知识
- python 的基本使用
- Git 的使用与 Github（见 DezemingFamily 的《Git 学习教程》）

我是在 2022 年 1 月 16 号开始 clone 这个项目的，而且我发现这个项目的函数和一些注释与 2021 年以及 2020 年都有些不同，但是基本上都是一致的。在叙述时，我采用中英文混杂的方式，例如“自动微分与 auto-difference”和“梯度与 gradient”，但我会在名词最开始出现的时候注明其中英文对照。在学习和实践过程中，本人参考了 Github 上很多其他人的代码（如 [18][19] 等），由于版本之间的不同，很多比较旧的实现方案需要进行一定的修改，但多亏别人的工作，使得我能即使发现自己实现的思路中的问题，并能够进行改正。

感谢 Alexander Rush 教授能够提供这么好的学习教程，虽然没有在线视频课公开课，但还是可以通过各种网络资源（比如 Cornell 大学学生的 Github 作业）来学习和理解该课程。这也是我第一次将深度学习和链式法则、雅克比矩阵结合起来。以及感谢 Python 视界公众号、机器之心公众号。我是在 Python 视界公众号 2021 年 12 月 15 日推送的文章中了解到的该项目，而在此之前，我就早早想自己实现一个神经网络框架，而该课程使我少走了很多弯路。

20220202：完成自动微分章节。20220217：完成张量章节。20220223：完成 fast 操作章节。20220227：全部完成。

1. 准备工作

本章讲述开始学习 *miniTorch* 之前的准备工作。

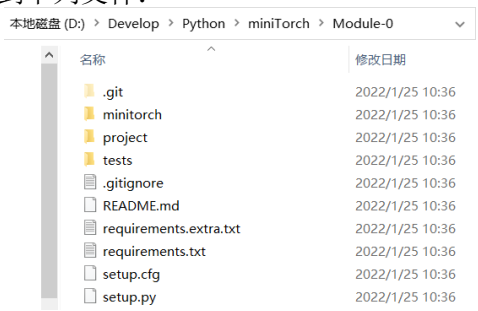
1.1 建立环境

首先需要安装 python3.7 以上的版本，这里建议使用 anaconda 来进行安装和管理，开发时可以使用 pycharm。

我们从 [1] 上把源码下载（使用 Git 进行 clone，先 fork 到自己的 github 里。Git 的使用参见《Git 学习教程》）：

```
git clone git@github.com:你的用户名/Module-0.git
```

为了更简洁（我们在第二章会把整个源码结构进行讲解），我们单独 clone 每个 module。进入 module0 文件夹，可以看到下列文件：



现在参考 [2] 的搭建环境方法。我们在该路径下安装所有需要的工具，在 cmd 命令行进入该目录，然后运行 pip 安装指令：

```
python -m pip install -r requirements.txt
python -m pip install -r requirements.extra.txt
conda install llvmlite
```

可能你的环境已经安装了一些功能，所以再次安装会先卸载然后再进行安装。使用 anaconda 安装时，会显示 llvmlite 卸载失败，所以需要单独安装（见上面最后一行）。执行上述第一行命令时，pytest-astropy 出现错误可以不用管。

现在执行：

```
python -m pip install -Ue .
```

. 表示当前目录的 python 工程。-U 表示如果已经安装了，就升级到最新版，-e 表示 editable，pip 会直接从版本控制工具中进行安装。这行命令会把当前的 miniTorch 给安装到 python 的 site-packages 里。

现在进入 python 环境，如果导入 minitorch 没有问题，就说明环境搭建正确：

```
>>> import minitorch
```

1.2 作业任务的完成方法

注意，此处只是根据 [2] 进行的描述，加了一些自己的描述方法，实际中我们会对每个作业进行更深入地分析和描述。

每个 MiniTorch 任务的结构都模仿了为真正的开源项目做出贡献的过程，对贡献者有特定的需求。本节列出了对参与者的代码的期望。

1.2.1 代码样式

需要保持代码的条理化和干净化，以便更容易调试、优化和记录。为了帮助完成这一过程，在所有作业上都要使用所需的格式。

修复样式错误可能是一个烦人的过程。但是，有一个很好的技巧可以自动修复大多数格式问题。我们使用 black [4] 自动重新格式化所有代码，以满足大多数需求（请参阅 [4] 如需了解更多详细信息）。black 是 psf（Python 软件基金会）下的一个工具，我们前面已经安装了这个工具，可以在 cmd 中运行（不是在 python 环境中）：

```
black .
```

作为项目持续集成的一部分，需要检查代码样式。可以通过在主目录中运行 flake8linter 来执行此操作：

```
flake8
```

由于这两个工具（Black 和 flake8linter）我以前都没有用过，所以我花了一些时间整理它们的资料并发布在了“程序设计与编程语言-python”栏目里。

1.2.2 作业测试

每个作业都有一系列测试，要求您的代码通过。

这些测试位于 tests/目录中，并采用 pytest 格式，见 [5]。该目录中名为 test 的任何函数都将作为测试组件的一部分运行。每个作业有 4 个任务组，都需要通过才算完成。

```
>>> pytest -m task0_0
```

除了运行所有测试的完整任务外，还可以在单个文件中运行测试，包括：

```
>>> pytest tests/test_operators.py
```

甚至是一个特定的函数测试：

```
>>> pytest tests/test_operators.py -k test_sum
```

Pytest 将隐藏测试中的所有 print 语句，除非测试没有通过。

1.2.3 文件

在整个代码库中，需要以标准化的样式记录所有函数。文档 (doc) 对于 Python 代码库是至关重要的，使用它来传达对许多函数的需求。函数应具有以下形式的 DocString（称为 Google docstyle）：

```
def index(ls, i):
    """
    List indexing.

    Args:
        ls (list): A list of any type.
        i (int): An index into the list

    Returns:
        Value at ls[i].
    """
    ...
```

此处 [6] 列出了此代码格式的完整描述。项目要求您始终保持文档符合标准。如果文档格式不正确，将抛出 Lint 错误（我也不知道会在哪里抛出该错误，我在实际学习中并没有在意这些）。

1.2.4 持续集成

持续集成（Continuous Integration (CI)），就是每天多次进行集成、编译测试，意图更好地把握进度和减少 BUG。

我一直没有弄明白这里说的是什么意思，但是这并不妨碍学习 miniTorch，为了完整性我还是把该过程写在下面。

除了本地测试之外，为了方便在每次代码推送时在服务器上自动运行和检查测试，还建立了相应的工程。通过提交代码，推送到服务器，然后登录到 GitHub，就可以看到自己在任务上的完成情况。这一过程需要几分钟，但这是一种简单的方法来跟踪作业的进度。

具体而言，可以运行：

```
$ git commit -am "First commit"
$ git push origin master
```

然后转到您的 GitHub 并单击“Pull requests”。单击请求本身会提供一个链接，显示当前工作进度。

Notice 1.1 (Module 的参考答案) 每一个 Module 的内容完成以后，我会把它们打包好，把代码放在网站的本书链接旁。我强烈建议读者先自己去摸索着实现，遇到实在难以解决的问题再来参考我的实现代码，这样才能更好地锻炼自己的能力。

2. 基础-Fundamentals

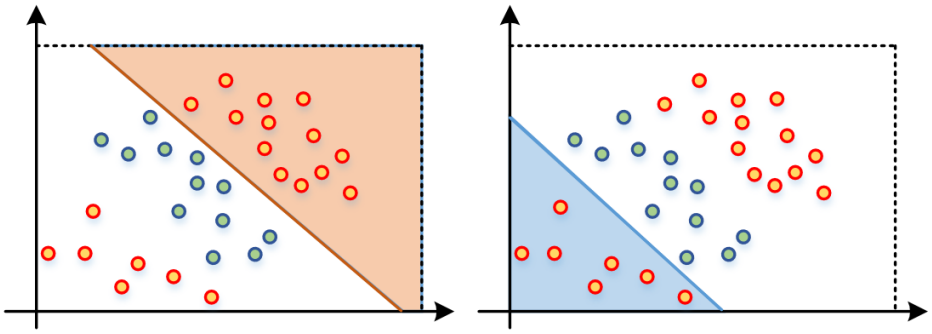
本章讲述开始学习 *miniTorch* 的基本模块，完成 *task0* 的全部内容，并理解整个工程。

2.1 机器学习基础摘要

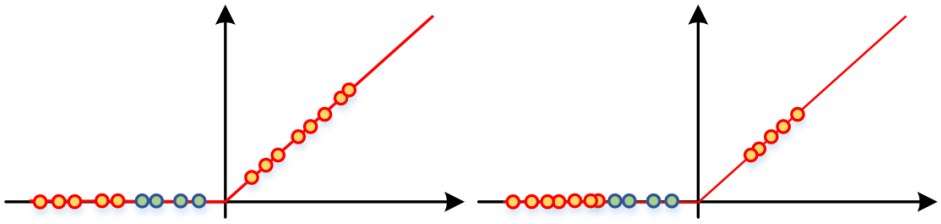
所谓梯度，即为所有参数取一组导数（称为梯度），然后降低损失值。

直观地说，神经网络将分类这个过程分为两个或多个阶段。每个阶段都使用一个线性模型将数据变换为新的点。最后一步是对变换后的点进行线性分类。

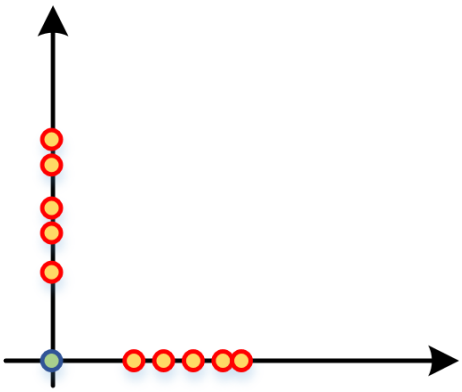
我们举个例子，见下图的分类，分别用两条分界线进行分类：



然后把 ReLU 函数作用于它们距离两个分界线的距离：



将作用后的两个值分别作为其变换后的横纵坐标，可以看到，现在很容易就能进行分类了：



从数学角度上来说，其实就是先使用 ReLU 做一次映射：

$$h_1 = ReLU(w_1^1 \cdot x_1 + w_2^1 \cdot x_2 + b_1)$$
$$h_2 = ReLU(w_1^2 \cdot x_1 + w_2^2 \cdot x_2 + b_2) \tag{2.1.1}$$

然后再做一次线性分类：

$$m(x_1, x_2) = w_1 \cdot h_1 + w_2 \cdot h_2 + b \tag{2.1.2}$$

当前的神经网络模型看起来很简单，但有效地建立它需要建立系统基础设施。然而，一旦我们有了这个基础设施，我们将能够轻松地支持大多数现代神经网络模型。

2.2 当前工程的简单分析

打开文件夹以后，就能看到三个主要的文件夹：minitorch、project 和 tests。

其中 minitorch 是我们主要的工程文件，也是 miniTorch 的主体；tests 文件夹下是用来测试 miniTorch 的功能的程序；project 主要是将 miniTorch 建立模型，然后做一些实际测试和可视化。

2.3 完成 task0.1 与 task0.2

每个 Module 都有一套帮助完成任务的指南 (Guides)。

开始代码就是 Module-0，本介绍性 Module 重点介绍未来的 Module 中用于测试和调试的几种核心技术，还包括一些基本的数学基础工具。在本 Module 中，您将开始为 MiniTorch 构建一些基础设施。

我们把 operator.py 里面的 task0.1 任务完成（都是很简单的函数），然后在 cmd 命令行进入 Module-0 文件夹，运行：

```
python -m pip install -Ue .
```

显示安装成功即可。然后 cmd 执行：

```
pytest tests/test_operators.py -k test_relu
```

如果提示缺少某些库或模块，就用 pip 来建立。成功通过测试的标志是：

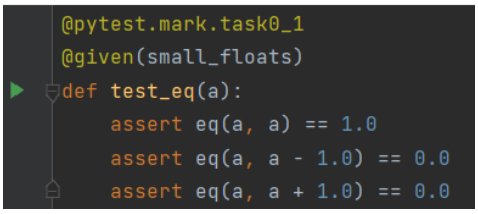
```
2 passed, 32 deselected
```

如果我们执行命令：

```
pytest tests/test_operators.py -k test_sigmoid
```

则会报错，这是因为我们并没有实现 test_sigmoid 函数里面的测试内容（完成该内容属于 task0.2）。

在 pycharm 中，可以通过直接点击 test_operators.py 文件左边的绿色按钮来执行测试：



task0.2 的内容也并不复杂，只是由于 test_symmetric 等几个函数并没有输入参数，你可以自己加上参数列表，我不清楚这样会不会破坏程序本身，但我觉得没什么问题：

```
@pytest.mark.task0_2
@given(small_floats, small_floats)
def test_symmetric(a, b):
    assert mul(a,b) == mul(b,a)
```

2.4 完成 task0.3

task0.3 就有一定的难度了。首先我们注意，map 的返回是一个函数，因此应该这样写：

```
def map(fn):
  def process(ls):
    arr = []
    for item in ls:
      arr.append(fn(item))
    return arr
  return process
```

reduce 函数就是将一个初始值与 list 里的元素作为 fn 函数的输入，得到的输出再与下一个元素作为 fn 的输入，直到把整个列表的元素都遍历完：

```
def reduce(fn, start):
  def process(ls):
    ans = start
    for item in ls:
        ans = fn(ans, item)
    return ans
  return process
```

task0.3 不止有编写的任务，还要写一些测试代码。测试代码这里只给出 test_sum_distribute 的程序：

```
def test_sum_distribute(ls1, ls2):
  assert_close(add(sum(ls1),sum(ls2)), sum(addLists(ls1,ls2)))
```

2.5 Modules 与 task0.4

2.5.1 Modules

深度学习又大又复杂，通用模型可以包括跨越数百个非正式模块组 (informal module groups) 的数亿个学习参数。为了处理如此复杂的系统，重要的是要有数据结构，将复杂性抽象出来，以便更容易访问和操作特定组件，并将共享区域组合在一起。

在编程方面，模块 (Modules) 已经成为一种流行的范式，可以将参数组合在一起，使它们易于管理、访问和寻址。关于这种设置，机器学习并没有什么特别之处（MiniTorch 中的一切都可以在没有模块的情况下完成），但它们让实现深度学习框架变得更轻松，代码也更有条理。

除了 Module，我们还定义一个参数 (Parameter) 类，现在我们把 Parameter 看作一个持有者——它只是一个存储一个值的特殊的对象。Parameter 是存储在模块中的特殊容器，它被设计用来保存一个变量，但我们允许它保存任何用于测试的值。

为了更好的管理，Parameter 与 Module 分组，模块提供了一种存储和查找这些参数的方法。打开 Module.py，就能看到 Module 类和 Parameter 类。

Module 是一种递归的树形数据结构：Module 形成一个存储参数 (parameters) 和其他子模块 (submodules) 的树，它们构成了神经网络堆栈的基础。

Module 内部三个成员：_modules 是用来存储子模块 (child modules) 的字典；_parameters 是用来存储参数 (Parameter) 的字典；training 是一个 bool 类型的变量，用来表示是训练模式 (training mode) 还是计算模式 (evaluation mode)。

这种基础设施的主要好处是，它允许我们使用 named_parameters() 将模块展平 (flatten)，以获取其所有参数——这将返回 module 和所有派生的 sub-modules 中所有参数的字典。这里的 names 指的是字典中的 keys，它给出了树中每个参数的路径（类似于 python 的 dot 符号）。重要的是这个函数不仅返回当前 module 的参数，还递归地收集下面所有 modules 的参数。在 [7] 中介绍 named_parameters 有简单的代码来解释这种字典的使用。

2.5.2 完成 task0.4

任务 0.4 是来实现一个树形的数据结构，在每个节点上存储 named Parameter。这样的数据结构让用户很容易创建树，可以通过遍历找到所有感兴趣的参数。

我们先看一下 Module.train 的实现：

```
def train(self):
  def update(cur):
      cur.training = True
      for child in cur.__dict__["_modules"].values():
```

```
            update(child)
    update(self)
```

其实就是用 ___dict___ 函数来索引到所有子模块的 training 设置为 True。(注意老版的 miniTorch 是用一个名为 mode 的字符串表示模式，可以赋值为"train" 或"eval")。

然后再看一下 named_parameters 函数的实现：

```
def named_parameters(self):
    res = {}
    def helper(name, node):
        prefix = name + "." if name else ""
        for k, v in node._parameters.items():
            res[prefix + k] = v
        for k, v in node._modules.items():
            helper(prefix + k, v)
    helper("", self)
    return res
```

我们完成这几个成员函数，通过 test_module.py 的相关测试，就说明成功了。

2.6 Visualization 与 task0.5

虽然测试有助于保持正确性，但探索性分析对于获得直觉也是至关重要的。当你陷入困境时，通常最好的办法就是查看你的数据和输出。可视化我们的系统并不能证明它是正确的，但它通常可以直接帮助我们找出哪里出了问题。在整个开发过程中，我们将使用可视化来观察中间状态、训练进度、输出，甚至最终模型。

本节并不是必须的，但可视化通常是一个很重要的手段，所以也建议学习。

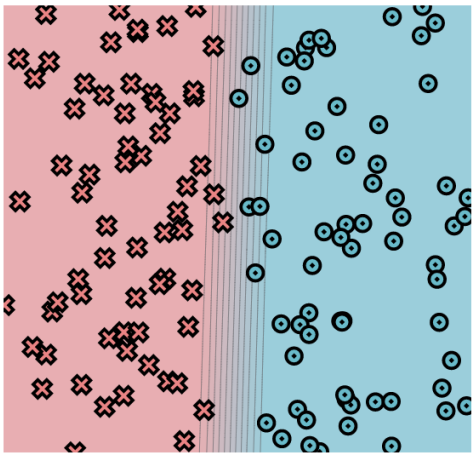
我们将使用的主要库是 streamlit [8]，只需要浏览一下 readme.txt，就可以看到，它相当于一个简单的 GUI 工具，可以将本地的一些数据发送到服务器上进行共享和可视化测试。

2.6.1 启动 streamlit

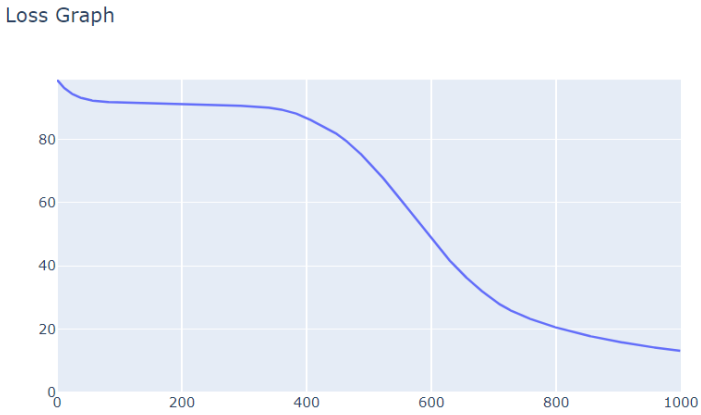
启动 streamlit，需要在 cmd 运行以下命令：

```
streamlit run app.py -- 0
```

然后胡乱输入一个邮箱，就可以自动打开一个网页。我们手动操作一下网页即可。我设置了训练 1000 轮：



训练过程的损失值下降过程：



可能大家会好奇，我们明明只定义了几个函数，但是却能进行网络的训练，这是为什么？这当然不是来训练我们的 Module，而是在 project 文件夹下的已经定义好的内容（run_torch 中的 pytorch 工具）。

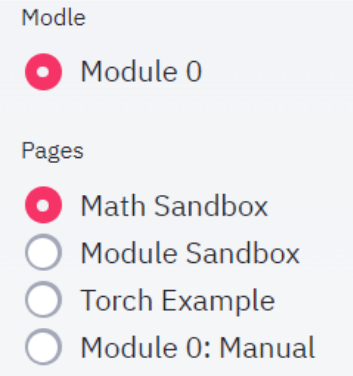
2.6.2 完成 task0.5

本节的意义只在于学习和使用 streamlit 工具。本节的任务是启动 streamlit 服务器并打印数据集的图像。手动创建分类器，将线性数据集分割为正确的颜色。

我们查看 app.py 文件的下面这个代码段：

```
if module_selection == "Module 0":
    ......
```

表示我们当前要可视化的模块 0。对比打开的网页：



其中，Math Sandbox 可以可视化一些数学函数；Module Sandbox 用来可视化定义的 MyModule；Torch Example 用来可视化 Torch 实例；Module 0: Manual 是我们自己实现的一些功能，由于这里相比于以前有很多改动（看 Git 提交历史就能看到），我在这里也不介绍详细操作了，不同的版本可能你需要修改的内容很不一样。

另外就是 named_parameters 和 pytorch 的 named_parameters 返回类型不一样，在 pytorch 中返回两个值，分别是模型的参数名 list 和实际参数 list；而 miniTorch 的返回是一个字典（键值对 list）。因此我们定义的 named_parameters 并不能很好地兼容当前搭建的可视化程序，会报错。

综上所述，由于 miniTorch 教学项目的 BUG 太多，task0.5 的内容不再详细叙述。

3. 自动微分-Autodiff

本章讲述开始学习自动微分的实现。对于“微分”和“差分”只是对应于连续或者离散的系统的描述，这里并不会明确区分。本章只是介绍 *miniTorch* 的 *Guide* 的流程和实现方案，在我学习时，我对源码中的内容花了很长的时间进行分析，但是本章并没有把分析过程写在这上面。为了本教程更完整地记录，在下一章我会把 *Fundamentals* 和 *Autodiff* 两个 *Module* 的程序代码原理进行细讲。

3.1 中心差分与 task1.1

本模块介绍如何只使用标量值来构建 miniTorch 的第一个版本。这涵盖了自动微分这个关键技术。然后，您将能够使用代码来训练初步的模型。

我们需要熟悉微分法则以及中心差分（有限差分）方法，可以自行百度或者参考 [9]。符号微积分可以明确计算出微积分式，但是对于非显示函数，这种方式无法计算出导数，因此黑箱函数会使用有限差分方法。

我们调用 git clone 命令把 Moudle-1 给 clone 到本地，然后将 Moudle-0 中的 operators.py 和 module.py 复制到 Module-1 的相应位置。

task1.1 的 central_difference 是求多元函数的第 i 个变量的偏导：

```
def central_difference(f, *vals, arg=0, epsilon=1e-6):
  arg_1 = [i for i in vals]
  arg_1[arg] += epsilon
  m = f(*arg_1)
  arg_1[arg] -= 2 * epsilon
  n = f(*arg_1)
  return (m - n) / (2 * epsilon)
```

cmd 命令行进入 Module-1 的主目录，执行以下命令来更新 miniTorch：

```
python -m pip install -Ue .
```

然后从 test_scalar 程序中测试。

> **Notice 3.1 (生成工程)** 注意如果使用 IDE 生成工程时（例如 pycharm），当我们从 Module-0 切换到 Module-1 时 IDE 可能跟踪记录的文件是以前 Module-0 的，因此仍然会生成 Module-0 的工程。此时测试就会出现“未实现错误”。建议把与当前工作不相关的工程放到其他目录（IDE 不会直接检索到的目录）下。

3.2 标量类的功能

这项任务需要熟悉 minitorch.Scalar。请务必首先仔细阅读关于跟踪变量 [10] 的指南，并熟悉 Python 的数值覆盖 [11]。

我们上一节学习了两种微分方法（符号求解和中心差分），符号求解是精确的，但不够灵活；中心差分很灵活，但是不够精确。本节我们介绍一个折中方案——自动微分。

我们将看到，自动微分的工作原理是收集函数中使用的计算路径的信息，然后将这些信息转化为计算导数的过程。与黑盒方法不同，自动微分将允许我们使用这些信息来更精确地计算每一步。然而，为了收集有关计算路径的信息，我们需要跟踪函数的内部计算——这可能很难做到，因为 Python 不会直接公开其输入在函数中的使用方式，即给定一个函数的输入，我们得到的只是输出，而不会得到输入在函数内部的计算过程。

3.2.1 跟踪计算

跟踪计算的主要技巧出人意料地简单明了：

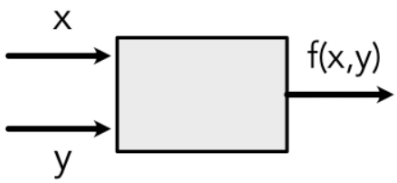
- 用代理类（称为 Variables）替换所有 Python 值。
- 将所有数学运算符替换为代理运算符，称为 Functions。
- 把 Variables 功能增强以记住过去应用于它们的 Functions。

对于表示方法，我们将新引入的 Functions 和 Variables 大写，以区别于 Python 的 functions 和 variables。

Variables 的行为应该与 Python 的数字完全相同，这样用户就无法分辨它们之间的区别。它看起来很简陋但在当前系统下显得很整洁。我们实际上只是创建了一个通用 Variable 类，见 autodiff.py 的 minitorch.Variable 类。

作为用户，我们永远不能直接改变或操纵 Variable 的值。我们必须创建一个作用于变量的函数类：minitorch.FunctionBase。该函数可以对 Variable 的参数进行操作，以生成 Variable 类型的输出，同时跟踪内部历史记录，它通过 FunctionBase.apply() 来调用。

我们可以把一个函数想做是一个盒子，输入 Variable 类型的变量 x（或者存在多个输入变量），将 x 的内容拆出 (unwarp) 进行计算，并将结果包装 (warp) 成另一个 Variable 变量：



3.2.2 标量函数

为了让整个 Variable/Function 的概念更具体，让我们关注 Scalar 类，它是 Variable 的一个子类。它被用于包装单个标量浮点数（存储在数据属性 (data attribute) 中），名为 minitorch.Scalar，它是自动微分跟踪标量值的重新实现。Scalar Variable 的行为尽可能接近标准 Python 的数字，同时也跟踪导致数字被创建的操作，它们只能由 ScalarFunction 来操作。

ScalarFunction 是处理和产生标量变量的数学函数的包装器，这是一个静态类，从不会被实例化。我们在这里使用该类将 forward 和 backward 的代码组合在一起。

我们可以用一个静态方法将一个简单的函数实现为一个类：ScalarFunction.forward()，例如：

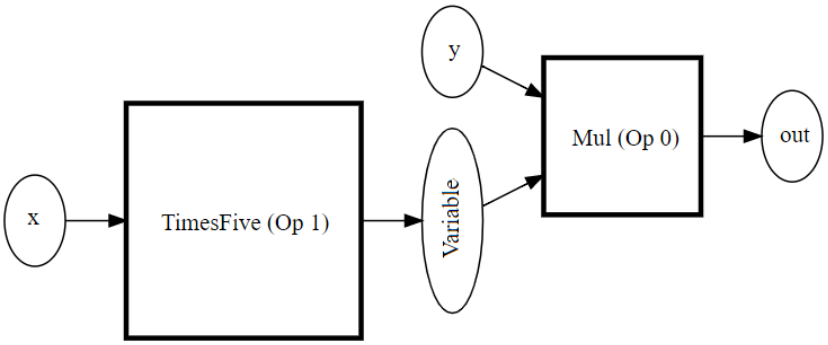
```
class Mul(ScalarFunction):
    @staticmethod
    def forward(ctx, x, y):
        return x * y
```

在 forward() 函数中，x 和 y 总是数字（不是变量）。forward() 函数处理并返回未包装的值。

关键的是，我们不能直接调用 forward() 函数，而是调用 apply()。这是因为“apply”在内部将 Variable 输入转换为浮点值以进行调用，创建历史记录的追踪，并将输出包装为 Variable。这里 z 和 out 都是 Variable（TimesFive 是一个类似于 Mul 的函数，表示把标量乘以 5）：

```
x = minitorch.Scalar(10., name="x")
y = minitorch.Scalar(5., name="y")
z = TimesFive.apply(x)
out = Mul.apply(z, y)
```

这段计算的历史由函数的链式来表示：



3.2.3 Syntactic Sugar(句法改进)

我们的函数还有一个小问题。这就是我们使用 Mul 的代码需要调用 apply，这样显得很丑。我们希望调用 * 就可以调用相乘，其实这很简单，可以参考 [11]。

3.3 完成 task1.2

task1.2 都是在 minitorch/scalar.py 文件里完成的。

用于操作标量的函数,例如 Mul,我们记作 ScalarFunction。我们首先需要实现 minitorch.scalar.ScalarFunction.fo 函数，它的参数是：

- ctx：一个容器对象，用于保存调用 backward 可能需要的任何信息。
- *inputs：float 类型的参数 list，n 个 float 类型的数。

根据样例可知，forward() 函数需要调用：

```
ctx.save_for_backward(...)
```

但是有的函数（例如比较大小）并不需要保存信息（例如 Add 函数），此外，不同的函数需要保存的信息也不一样（例如 Sigmoid 函数），我们暂时先忽略这些要保存的信息，等后面学习 backward 后再介绍。

3.4 自动微分与链式法则、完成 task1.3

3.4.1 自动微分

在 ScalarFunction 类中，调用 apply 就会调用内部的 forward() 函数。在表示时，我们用大写的首字母来区分我们定义的函数和变量与 Python 自带的函数和变量。

现在，我们加入了附加信息，这个附加信息属于给出单个函数的导数的类。自动微分背后的技巧是使用一系列函数调用来计算导数。就像正向计算函数 f(x) 一样，我们需要一种 backward 方法来提供这种局部导数信息。

对于每个函数，我们都需要提供一种 backward 方法来计算其导数信息。明确地说，对于给定参数 d_{out}，backward 需要计算 $f'(x) \times d_{out}$。

对于足够简单的函数，例如 $f(x) = 5x$，可以求得导数为 $f'(x) = 5$。但是对于复杂的函数，例如多元函数，需要求对于每个元的偏导，比如 $f(x,y) = x * y$，则这就是一个二元函数。

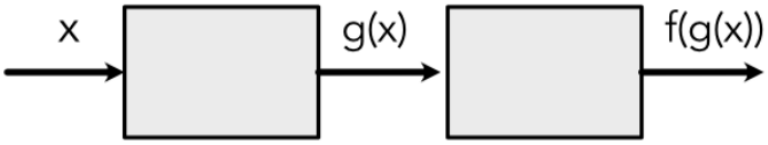
有些时候 backward 计算 $f'(x)$ 需要用到 x 的值，例如 $f(x) = x^2$，$f'(x) = 2x$。但是此时 backward() 函数并没有 x 值传入，但是没有关系，ctx 会记录这些信息，我们以 $f(x) = x^2$ 为例：

```
class Square(ScalarFunction):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x * x
```

```
@staticmethod
def backward(ctx, d_out):
    x = ctx.saved_values
    f_prime = 2 * x
    return f_prime * d_out
```

这种类型的函数需要在 forward() 时显式保存我们在 backward() 时可能需要的任何内容。这是一种代码优化，可以限制计算过程所需的存储量。

对于链式法则：



求导公式可以写为：

$$f'_x(g(x)) = \Big(f(g(x))\Big)' = g'(x) \times f'_{g(x)}(g(x)) \tag{3.4.1}$$

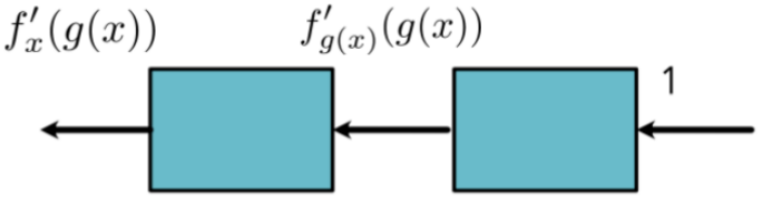
在上图中的计算过程如下：

$$z = g(x) \tag{3.4.2}$$

$$d_{out} = f'(z) \tag{3.4.3}$$

$$f'_x(g(x)) = g'(x) \times d_{out} \tag{3.4.4}$$

下图是将函数视为盒子的观点派上用场的地方：



这里的 1 表示最开始的 backward() 的输入 d_{out}，之后的 backward() 输出作为左边的 backward() 的输入参数的 d_{out}。

对于多变量函数而言，例如二元函数 $f(g(x,y))$（注意设此时 $z = g(x,y)$ 是一个标量），backward 表示如下：

$$f'_x(g(x,y)) = g'_x(x,y) \times f'_{g(x,y)}(g(x,y)) \tag{3.4.5}$$

$$f'_y(g(x,y)) = g'_y(x,y) \times f'_{g(x,y)}(g(x,y)) \tag{3.4.6}$$

或者可以表示为：

$$z = g(x,y) \tag{3.4.7}$$

$$d_{out} = f'(z) \tag{3.4.8}$$

$$\begin{cases} f'_x(g(x,y)) = g'_x(x,y) \times d_{out} \\ f'_y(g(x,y)) = g'_y(x,y) \times d_{out} \end{cases} \tag{3.4.9}$$

3.4.2 链式法则与完成 task1.3

task1.3 在 minitorch/autodiff.py 文件里，即 chain_rule 函数：

```
dericatives = cls.backward(ctx, d_output)
result = []
i = 0
if isinstance(dericatives, tuple):
    for val in inputs:
        if not is_constant(val):
            result.append((val, dericatives[i]))
        i = i + 1
else:
    for val in inputs:
        if not is_constant(val):
            result.append((val, dericatives))
        i = i + 1
return result
```

在上面的程序中，cls.backward 可能会返回一个 tuple（多个输入的函数，例如 Add）或者一个 value（单个输入的函数，例如 Exp）。

3.5 反向传播算法与 task1.4

backward() 函数告诉我们如何计算一个运算的导数。链式法则告诉我们如何计算两个连续运算的导数。在本节中，我们将展示如何使用这些函数来计算任意一系列运算的导数。

实际上，这看起来像是以从右到左的相反顺序重新运行我们的图表。然而，我们需要确保按照正确的顺序进行。反向传播的关键实现挑战是确保我们以正确的顺序处理每个节点，即我们首先处理每个使用变量的节点，然后再处理该变量本身 [12]。

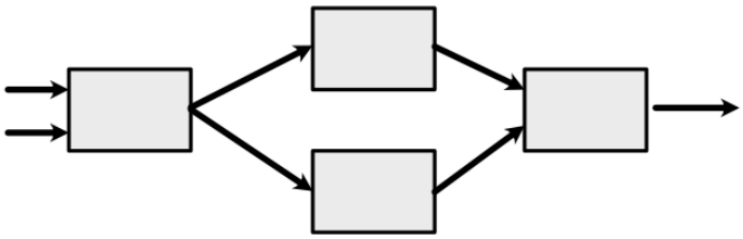
3.5.1 Running Example

我们有一个函数 $h(x,y)$ 和两个变量 x 和 y，我们想求 $h'_x(x,y)$ 和 $h'_y(x,y)$：

$$z = x \times y \tag{3.5.1}$$

$$h(x,y) = \log(z) + \exp(z) \tag{3.5.2}$$

我们设上面的计算（包括 +、×、log 和 exp）都是用 ScalarFunctions 实现的，即可以保存历史，这意味着最终输出变量构建了一个计算图，如下所示（注意盒子表示函数，也称为节点 (node)；箭头表示输入和输出的 Variable）：



最左边的两个箭头表示输入的 x 和 y；中间上方的盒子表示 log 函数，中间下方的盒子表示 exp 函数；最后一个节点是 + 函数，输出 $h(x,y)$。这里的 x 和 y 就是叶子变量 (leaf Variable)。

利用 backward() 方法就可以计算出所有节点的微分，我们可以随机应用这个规则，在每个节点聚集结果值时对其进行处理。然而，这可能是相当低效的。例如，考虑一个顺序：右、上、左、下。虽然这会处理每个节点，但有些导数不会到达其原始变量，所以顺序会变为：右、上、左、下、左（先计算完“下”的微分再计算出左的微分）。在实际过程中，我们会将整个过程排序为右、上、下、左（或者右、下、上、左）这个顺序，依次处理反向传播的导数。

3.5.2 拓扑排序

为了解决这个问题，我们将按拓扑顺序处理节点。我们首先注意到我们的图是有向的，并且是非循环的。方向性来自 backward() 函数，缺少循环是选择每个函数必须创建一个新变量的结果。

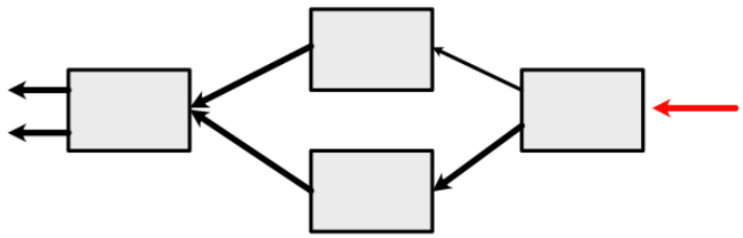
有向无环图的拓扑排序是一种确保在其 ancestor 之后不处理任何节点的排序，例如，在我们的示例中，左节点不能在顶部或底部节点之前处理。排序可能不是唯一的，它不会告诉我们是先处理顶部节点还是底部节点。

拓扑排序有几种易于实现的算法。由于图算法超出了本文的范围，我们建议使用拓扑排序的伪代码部分中描述的深度优先搜索算法 [13]。

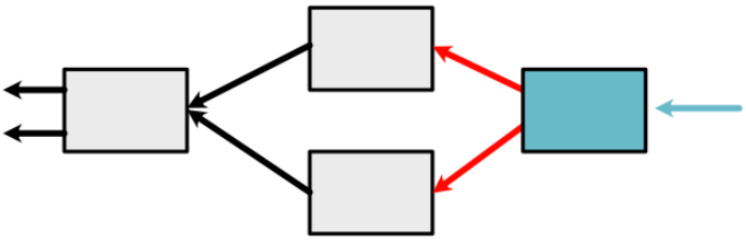
具体的代码较长，可见本书给出的代码文件。由于该部分代码和后面的 backpropagate 息息相关，现在就讲解并不能很容易讲得清楚明白，因此详细原理会放在下一章进行介绍。

3.5.3 Backprop

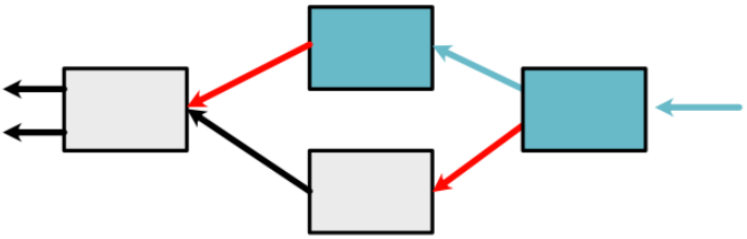
一旦定义了顺序，我们就一次处理一个节点。我们从最左边的（h 函数）节点开始处理：



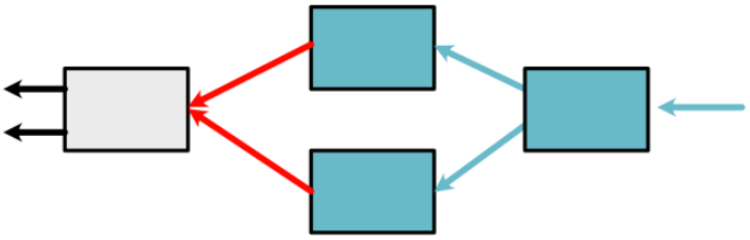
然后我们用链式法则处理函数。这将调用 + 的 backward()，并给出两个红色 Variable（对应于向前传递的 $\log(z)$、$exp(z)$）的导数。您需要在字典中跟踪这些中间的红色导数值：



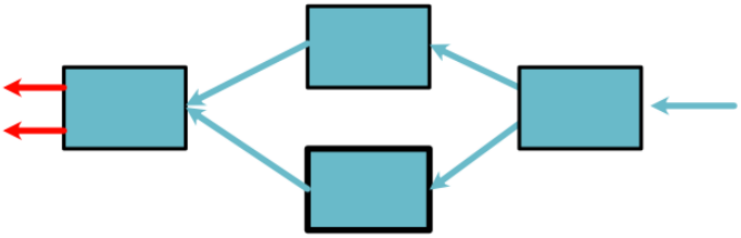
然后我们再处理上面那个节点（log 函数），此时左边节点的红色输出之一作为上面的节点的输入，计算以后得到一个新的红色输出（左边的红箭头）：



现在处理下面的那个节点。注意现在有一个有趣的结果：我们会产生一个新的红色箭头（见下图左边下方的红箭头），而这个红箭头也指向了刚才我们计算过的 z。这是一个有用的例子，是两个参数的链式法则的结果，这个变量的导数是每个导数的相加和。在实践中就是把它加到你的字典里。



最后再处理最左边的节点。在处理完这个变量之后，此时，剩下的就是我们的输入叶变量。



当我们按顺序到达叶变量时，例如 x，我们存储该变量的导数。由于这个过程的每一步都是链式规则的应用，我们可以证明，如果 x 和 y 是 minitorch.Variable 的实例，则 $h'_x(x,y)$ 和 $h'_y(x,y)$ 分别是：

```
x.derivative
y.derivative
```

3.5.4 Algorithm

如上图所示，每个红色箭头代表一个构造的导数，该导数最终在链式法则中传递给 d_{out}。从作为参数传入的最右边的箭头开始，backpropagate 应该运行以下算法：

第 0 步：调用 topological sort 来得到有序序列。

第 1 步：创建 Variable 和当前导数的字典。

第 2 步：对于每个反向顺序的节点，从队列中提取一个完整的 Variable 和导数。

如果 Variable 是一个叶子节点，加上它最后的导数（accumulate_derivative），并且循环到第 1 步。

如果 Variable 不是一个叶子节点，则执行下面三个步骤：(1) 调用 last_fn.backprop_step，并使用 d_{out} 作为导数来创建它（该函数会调用 chain_rule 来生成 (Variable, 导数) 对，对于多输入函数则是多个这样的对）。(2) 循环遍历链式法则产生的所有 (Variable, 导数) 对。(3) 在字典中累积 Variable 的导数 (检查.unique_id)。

> **Notice 3.2 (backward() 给微分赋值)** 注意在自动微分中，如果用过 pytorch 就会看到它的梯度需要在每一轮训练时清零，这是因为它的微分在计算后并不是被赋值到 _derivative 里，而是被累加到上面。

为了讲解的更细致和更有条例，详细的代码会放在下一章进行讲解。

3.5.5 task 中的 forward() 和 backward()

我们本节要完成的内容与 ctx 参数有关。之所以放在最后来介绍是因为本小节内容其实我感觉放在 task1.3 就很合适，放在 task1.4 就有点太迟了。不过我们还是按照作业顺序来进行讲解。

对于给定的例子，一个是 Add 函数，一个是 Log 函数。其中，Add 不需要使用 ctx 保存数据，因为 $f(a,b)=a+b$ 对 a 和对 b 求偏导得到的值都是 1；Log(a) 就是数学上的 $\ln a$，对 a 求导以后得到 $\frac{1}{a}$，因此需要保存当时的 a 值。

对于乘法 $f(a,b)=a\times b$，$f'_a(a,b)=b$，$f'_b(a,b)=a$。

对于 Inv 函数，可以用到前面实现过的 operators.inv_back 函数。

对于 Sigmoid 函数，求导得：

$$\left(\frac{1}{1+e^{-x}}\right)'=\frac{e^{-x}}{(1+e^{-x})^2}=\frac{1}{1+e^{-x}}(1-\frac{1}{1+e^{-x}}) \tag{3.5.3}$$

我们设 $b=\frac{1}{1+e^{-a}}$，得到上式为 $b(1-b)$，因此我们储存 b 即可。

对于 LT、EQ 这两个函数，backward() 返回 0.0, 0.0 即可。

测试在 test_autodiff.py 文件中，为 test_backprop 函数。注意由于 Variable.backward() 需要调用 backpropagate() 函数，所以需要实现 backpropagate 函数才能测试我们实现的 ScalarFunction 的 backward() 是否正确。

3.6 模型训练与 task1.5

本节完成后，你就能实现一个可以训练的神经网络。

3.6.1 Linear 类与 Network 类

我们打开 project/run_scalar.py 文件。看到 Network 类和 Linear 类。在 main 函数中定义了 HIDDEN=2。

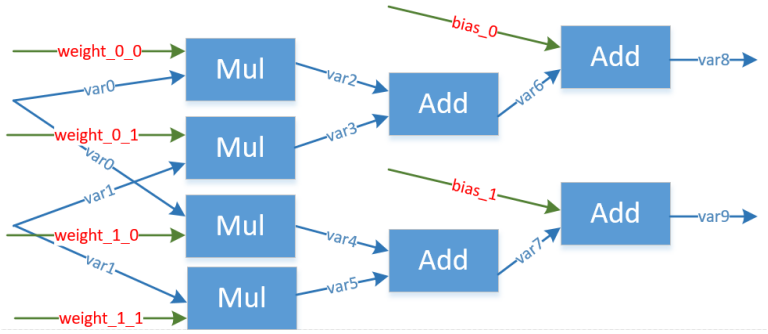
我们在 Network 类下实现（hidden_layers 就是传入的 HIDDEN，我们需要注意的是，）：

```
def __init__(self, hidden_layers):
  super().__init__()
  self.layer1 = Linear(2, hidden_layers)
  self.layer2 = Linear(hidden_layers, hidden_layers)
  self.layer3 = Linear(hidden_layers, 1)
```

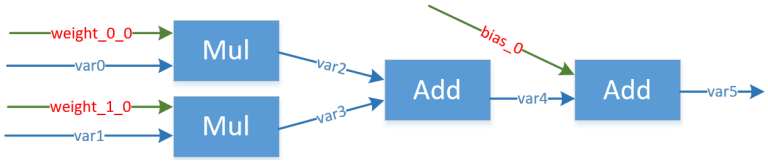
然后现在分析一下 Linear 的初始化函数。比如第一层网络的输入是 2,输出也是 2,那么 self.weights 根据输入创建一个二维数组（2X2）：[[weight_0_0,weight_0_1],[weight_1_0,weight_1_1]]。之后再根据输出增加 2 个偏置 [bias_0,bias_1]。因此，Linear(2,2) 会创建 6 个 Scalar 实例。

在 Network 的初始化函数中，总共会创建 6+6+3=15 个变量，作为权重 (weight) 或偏置 (bias)。

Linear 运行 forward 后，会生成一堆新的 Variable，但是这些 Variable 在每次运行完以后都会不再有用，因为我们要训练的内容是 weight 和 bias，而这些 weight 和 bias 一定都是叶子（我们只会保存叶子的导数）比如下图所示：



因此，2 个 Variable 输入到 Linear1 以后，会产生 8 个 Variable，最后面输出的两个 Variable 再经过 Relu 又生成 2 个 Variable——也即是说，总共会产生 10 个 Variable。对于最后一层 Linear，输入为 2 个 Variable，输出为 1 个 Variable：



所以中间会生成 4 个 Variable，加上最后 sigmoid 生成的第 5 个 Variable，总共会生成 5 个 Variable。

3.6.2 ScalarTrain 类与 SGD 类

该类的主要功能在 train 函数上。

在 train 函数中，看到 optim 优化器是 SGD，SGD 负责搜集所有的参数（weight 和 bias），

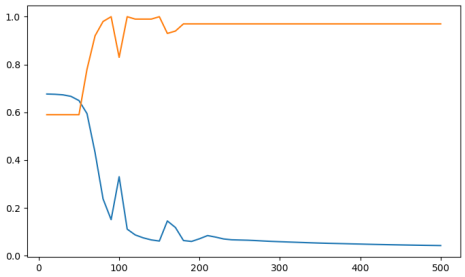
然后等反向传播算法完成以后，就将这些参数更新。SGD 是随机梯度下降，每次会随机使用一个样本来更新网络权重。

这里计算 loss 的方法就是 logistic 回归的交叉熵。

3.6.3 模型训练与小结

现在，执行 run_scalar.py 文件里的 main 函数，就可以开始训练了。如果你的程序训练中会报错，那就需要一定的调试功底，多设置一些打印输出来查看运行状态。我在完成本章内容时光调试查错的时间就超过了 20 多个小时，几乎在每步中都写了不少打印信息，即使在我通过了全部 test 以后，反向传播和链式法则的代码仍然有一些 bug。

我把训练过程的 loss 值和正确率保存在了数组里，并绘制出来：



至此，我们的自动微分章节就全部完成了。

本章仅仅是按照教程 [2] 的顺序讲解，并没有讲解一些源码上的内容，下一章我会将前面的基础和自动微分两个章节的内容综合起来进行讲解。

4. 前两个 Module 的详细分析

本章详细分析 *minitorch* 目录下的四个目录的源码。*project* 目录下虽然也有重要的代码需要掌握，但前面的描述已经很清楚了，这里就不再赘述了。

4.1 本章的基本介绍

在我完成了 Module-1 的内容以后，感到自己在源码阅读和思考上花费了不少的时间，而 github 上的教程并没有很详细的源码分析，这也是因为我并没有机会直接参与到这门课中，只能一点一点地琢磨。

我决定用单独的一章将源码的分析记录下来，一是为了给别人一个参考，二也是为了自己更好地理解和总结。

4.2 运算与 operators.py

operators.py 文件里面的函数分为两部分：一部分是一些加减乘除等的基本函数；另一部分是作为练习的高阶函数，例如 map、zipWith，这里的 map 和 zipWith 和 reduce 返回的都是一个函数。

4.3 模型与 module.py

module.py 只由两个类构成：Module 和 Parameter。尽管本文件定义的这两个类并没有在 autodiff.py 和 scalar.py 两个文件中用到（我们训练的模型也暂时不会使用这两个类），但它们是构建整个系统的基础。

4.3.1 Parameter 类

首先是 ___init___ 方法。该方法在赋值是和 update() 函数都会运行同样的一段代码：

```
self.value = x
if hasattr(x, "requires_grad_"):
    self.value.requires_grad_(True)
    if self.name:
        self.value.name = self.name
```

hasattr 就是 has attribute 的简写，即判断对象 x 有没有某个属性（该类有没有以该字符串为名的变量或函数）。在 test_module.py 文件中对该内容进行了测试：

```
class MockParam:
  def __init__(self):
      self.x = False

  def requires_grad_(self, x):
      self.x = x
# 测试程序
def test_parameter():
  t = MockParam()
  q = minitorch.Parameter(t)
  print(q)
  assert t.x
  t2 = MockParam()
  q.update(t2)
  assert t2.x
```

然后是 ___repr___ 方法，返回程序开发者看到的该类信息。开发者在命令行输入该类的某个实例化后的变量名其实就是打印出该类的 ___repr___ 方法。该方法调用了 repr 来将类的成员变量 value 值输出。

___str___ 这是当我们 print 该类时调用的函数，打印出该类的信息。这里还是打印出成员变量 value 的值。

4.3.2 Module 类

该类的函数较多，我们分为覆盖 python 默认函数的函数和其他操作函数。

___init___ 一共定义了该类的三个变量：_modules（在该类的实例中存储其他 Module）、_parameters（存储该类实例的 Parameters）和 training（用来区分是“训练”还是“计算”）。

对于已有 Module 实例，___setattr___ 函数负责把属性（其他的 Module 实例和 Parameter 实例）加到实例中。

对于已有 Module 实例（设 module1），___getattr___(self, key) 函数负责把已有的属性取出（简单介绍一下，注意如果该函数的输入参数 key 可以正常追溯到，例如初始化函数中定义了一个变量 a，则调用 module.a 时并不会触发 ___getattr___）。

___repr___ 函数里面的内容看起来比较长，这是因为需要把 Module 里的全部 sub Module 和 Parameters 都整理起来，所以用到了递归。

___call___ 函数会调用 forward() 函数，但是由于 Module 基类的 forward() 函数返回未定义错误，所以只有 Module 的派生类覆盖了父类的 forward() 之后才能调用。例如在测试文件中的：

```
class ModuleRun(minitorch.Module):
    def forward(self):
        return 10
```

关于参数的下面三个函数因为前面详细介绍过了，这里不再赘述：

```
named_parameters
parameters
add_parameter
```

train 函数和 eval 函数分别将本类的实例以及实例中的子 module 实例的 training 全都设置为 True 或 False。

4.4 自动微分与 autodiff.py

autodiff.py 文件分为四个板块：Variable 类、控制可选 tuples 的代码、Functions 有关的类、反向传播有关的函数。这里，凡是 FunctionBase 及其派生类都统称为 Function，用来区分其他的函数。

4.4.1 控制可选 tuples 的代码

该板块有两个函数：wrap_tuple 和 unwrap_tuple。tuple 的好处是元祖不会被修改，因此管理会更容易（比如每次计算时我们先从 tuple 中拆出里面的元素再计算，计算完以后在包装成 tuple，防止在其他地方被修改）。

对于一个变量 x，包装成 tuple 的方式很简单，加括号和逗号即可：$(x,)$。

4.4.2 Functions 有关的类

Function 与 Variable 是相辅相成的，在本节介绍中也会涉及不少 Variable 类中的内容，大家应该先熟悉一下这几个类的成员变量和函数有哪些。

Context 类是在 Functions 调用 forward() 时用来保存当前信息的类。该类有两个成员变量：_saved_values（存储保存的值，这个值会被封装为一个 tuple）和 no_grad（是否要保存梯度，False 表示要保存梯度，一般都是要设置为 False）。

> **Notice 4.1 (tuple 与返回值)** 当一个函数有多个返回值时（比如返回两个 float 类型的数），其实就是返回一个 tuple，如果只用一个变量来接收多个返回值，这个变量就是 tuple 类型的；如果用两个变量来接收这个函数的返回值，这两个变量就都是 float 类型的。

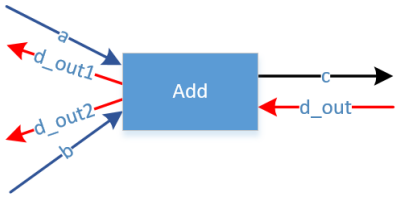
save_for_backward() 函数存储 Functions 做 backward() 时需要用到的信息。

saved_values() 函数返回存储的值（tuple 类型），如果判断 no_grad 为 True 时则会报错（assert 失败）。saved_tensors 是与张量有关的，我们暂时还没有涉及，所以暂时不需要管（当前这里也只是调用 saved_values()）。

[1] History 类

History 类用来存储操作某个变量的 Function 的历史，一共有三个成员变量：last_fn 表示上一次调用的函数；ctx 是 Context 实例，保存信息用于 backward()；inputs 是 last_fn 调用 forward() 函数时的输入参数（在当前工程里，它其实就是前面的变量，比如 c=Add(a,b)，a 和 b 就是 c 的 History 的 inputs）。

backprop_step() 函数用来返回与 inputs 相关的导数 (derivative)，也就是下图中的红色部分（调用 last_fn.chain_rule）：



[2] FunctionBase 类

FunctionBase 类里面只有 chain_rule 是我们实现的，其他函数都是工程里已经存在的。

variable() 函数没有实现，注释中说在派生类中实现，ScalarFunction 类中，variable = Scalar，也就是说该函数即 Scalar 的构造函数。

apply(cls, *vals) 函数是用户调用的，这是一个类方法 (classmethod)，它的注释很详细，该函数一共做了四件事：(1) 判断一下输入参数是否需要梯度；(2) 对 Function 调用创建一个 Context；(3) 为 Function 调用 forward() 函数；(4) 把 Context 附到新生成 Variable 的 History 里。

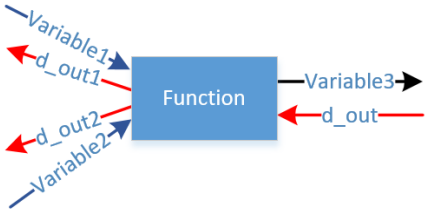
我们先说 (1)，其实在调用时，比如 Add，可以调用两个 Variable，比如 Add(a,b)；也可以调用一个变量和一个数字，比如 Add(a,2.0)；甚至调用两个数字，比如 Add(1.0,2.0)。此时，就需要判断输入的 vals 里面的参数是 Variable 还是数字。只要存在 history 不为的 None 的输入 Variable，need_grad 就设置为 True。如果输入全都是数字，那么我们也不需要存储当前 Variable 的 Context，因为当前 Variable 将是一个定值，不必训练。

在第 (4) 步时，调用 cls.variable() 函数，其实对于 ScalarFunction 来说就是调用 Scalar 类的构造函数。

> **Notice 4.2 (类方法与 cls)**] self 作用于实例本身，而 cls 则作用于类本身，使用 cls 可以直接调用类的静态方法。

chain_rule() 函数是我们自己实现的函数，就是调用 backward() 之后，将不是常量的 Variable 及其导数返回。

我们画一个示意图来描述上述的过程：



如上图，深蓝色箭头表示 forward() 过程，红色表示 backward() 过程。当调用 Var3=Add(Var1,Var2) 后，生成的 Var3 的 history.last_fn 就是这个 Add 函数。在该计算图中，由于 Var1 和 Var2 并不需要任何函数来生成，所以它们的 history.last_fn 都是 None，换言之，它们都是叶子。我们知道：

$$d\_out1 = d\_out \times \frac{dFunction(Var1, Var2)}{dVar1} \tag{4.4.1}$$

$$d\_out2 = d\_out \times \frac{dFunction(Var1, Var2)}{dVar2} \tag{4.4.2}$$

4.4.3 Variable 类

该类我们已经很熟悉了，这里简单介绍一些方面。

_derivative 表示计算得到的导数，该值会在 backpropagate() 函数中被更新。

unique_id 用于记录每一个 Variable 的索引，区分不同的 Variable。

注意，我们上一节刚刚说过，当 self.history.last_fn 为 None 时，表示该 Variable 是一个叶子。

4.4.4 反向传播有关的函数

is_constant() 函数判断某个 Variable 是不是常量，判断依据是有没有 history，如果没有 history 则说明该 Variable 当做一个常量处理。

本小节剩下的两个函数较为复杂，我们将主要精力放在这上面。可以说，这两个函数彻底理解了、能自己写出来，就说明自己对当前的项目非常了解了。

[1] 拓扑排序

关于拓扑排序可见 DezemingFamily 的《有向无环图的拓扑排序》，我们使用基于深度优先搜索的算法来排序：

```
def topological_sort(variable):
  PermanentMarked = []
  TemporaryMarked = []
  result = []

  def visit(n):
      # Don't do anything with constants
      if is_constant(n):
          return
```

```
        if n.unique_id in PermanentMarked:
            return
        elif n.unique_id in TemporaryMarked:
            raise(RuntimeError("Not a DAG"))

        TemporaryMarked.append(n.unique_id)

        if n.is_leaf():
            pass
        else:
            for input in n.history.inputs:
                visit(input)
        TemporaryMarked.remove(n.unique_id)
        PermanentMarked.append(n.unique_id)
        # 插入到开头
        result.insert(0,n)

    visit(variable)

    return result
```

[2] 反向传播

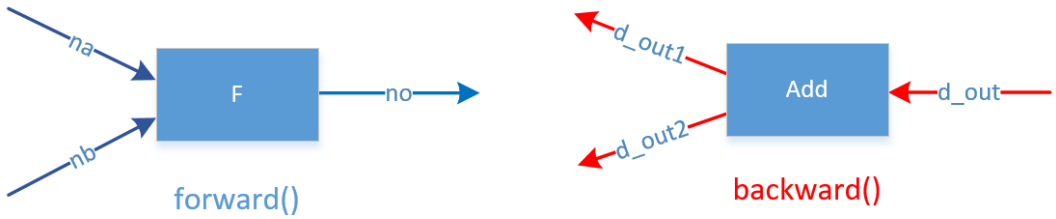
注意我们当前的函数都是标量函数，也就是函数的输出都是标量。

Variable 和 Function 都有 backward() 函数，其中 Function 的 backward() 函数调用的就是其派生类实现的 backward() 函数，而 Variable 的 backward() 函数其实调用的就是 backpropagate() 函数。这个关系一定要理清，否则很难写出正确的代码。

我们先来分析 History.backprop_step() 函数：

```
def backprop_step(self, d_output):
    return self.last_fn.chain_rule(self.ctx, self.inputs, d_output)
```

该函数由 Variable 调用，我们以下图为例（设函数为 Mul）：



调用 Mul 的形式大致如下：

```
no = Mul(na, nb)
```

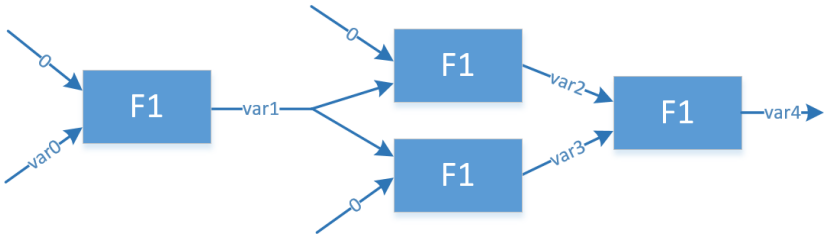
调用 Mul 函数后，no 的 history 里的 ctx 就存储了 na 和 nb 的当前值，inputs 就存储了 na 和 nb 两个 Variable 的引用。no 调用 backprop_step 函数的形式大致如下：

```
[(na, d_out1), (nb, d_out2)] = no.history.backprop_step(d_out)
```

我们以 test_autodiff.py 文件的第四个计算图为例，手动实现一下 backpropagate 的过程。

```
def test_backprop4():
  # Example 4: F1(F1(0, v1), F1(0, v1))
  var0 = minitorch.Scalar(0)
  var1 = Function1.apply(0, var0)
  var2 = Function1.apply(0, var1)
  var3 = Function1.apply(0, var1)
  var4 = Function1.apply(var2, var3)
  var4.backward(d_output=5)
  assert var0.derivative == 10
```

示意图如下：



记作：

$$\begin{aligned} F(var0) &= F1(F1(0,F1(0,var0)),F1(0,F1(0,var0))) \\ &= 2\times F1(0,F1(0,var0))+10 \\ &= 2\times\Big(F1(0,var0)+10\Big)+10 \\ &= 2\times var0+50 \end{aligned} \tag{4.4.3}$$

根据反向传播的 d_output=5，计算得到导数：

$$d_{out}\times\frac{dF(var0)}{dvar0}=5\times 2=10 \tag{4.4.4}$$

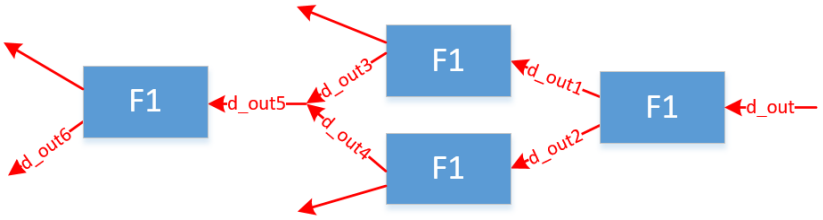
同理，我们可以求出（注意下面的 F_b 其实就是 F1）：

$$F_a(var1)=2\times v1+30 \tag{4.4.5}$$

$$F_b(var2,var3)=var2+var3+10 \tag{4.4.6}$$

$$\frac{dF_b(var2,var3)}{dvar2}=1 \tag{4.4.7}$$

所以，根据反向传播算法，我们先画出反向传播的过程：



上图的 $d\_out1 = 5, d\_out2 = 5; d\_out3 = 5, d\_out4 = 5, d\_out5 = d\_out3 + d\_out4 = 10;$ $d\_out6 = 10$。

函数实现并不难：

```
def backpropagate(variable, deriv):
  order = topological_sort(variable)

  derivs = {variable.unique_id: deriv}
  for node in order:
    d_output = derivs[node.unique_id]
    if node.is_leaf():
      node.accumulate_derivative(d_output)
    else:
      for input, d in node.history.backprop_step(d_output):
        if input.unique_id not in derivs:
          derivs[input.unique_id] = 0.0
        derivs[input.unique_id] += d
  return
```

根据要求，derivs 存储了所有中间变量的导数，而叶节点的导数被累加到了叶子本身的 _derivative 属性中。

4.5 标量与 scalar.py

scalar.py 比较简单，都是 autodiff.py 中的基类的派生类和具体实现，主要都是一些标量函数。

derivative_check(f, *scalars) 函数是用于测试的函数，该函数使用 central_difference() 来检测我们的 backward() 的正确性。根据该函数的前几行：

```
  for x in scalars:
    x.requires_grad_(True)
```

创建的输入都要调用 requires_grad_，该函数会为变量 x 创建 history（如果没有 history，则这个变量将是一个常量，输入都是常量则得到的 Variable 也将会被当做是一个常量）。

5. 张量-Tensors

本章介绍当前的自动微分机的重点内容——张量。张量是数据最根本的形式，而标量只是张量中最特殊的一类。前两章我们已经可以构建出一个神经网络框架，而本章我们则可以实现更高的效率。*Module-2* 的基础代码部分的讲解我会安排在下一章，所以在本章如果有涉及基础部分的内容，可以先参考下一章。

5.1 张量数据

我们当前的系统是完全正确且可以运行的，但它的效率并不高——每个变量都需要产生一个标量对象，而且还需要生成计算图来记录所有操作。在训练中还需要一个 for 循环来重复多次上述过程。

当我们引入张量时则会有效提升性能：张量将许多重复的操作组合在一起，以节省 Python 开销，并将分组操作传递给更快的实现。我们需要先熟悉张量索引 (indexing)，我们先介绍张量的指南 [14]。同时，阅读有关在 Torch 或 NumPy 中使用张量/数组的教程也会有所帮助。

5.1.1 什么是张量

张量是一个简单的概念，是任意维的多维数组。这是一种方便、高效的数据保存方式，与快速算子和自动微分配合使用时，数据保存功能会变得更加强大。标量就是一个最简单的张量，即 0 维张量；向量就是所谓的 1 维张量。2 维张量看起来像一个矩阵。

我们把有 5 个元素的向量形状 (shape) 表示为 (5,)，其大小 (size) 为 5。把 2X3 的矩阵形状 (shape) 表示为 (2,3)，其大小 (size) 为 6。

首先，张量可以很容易地改变维度的顺序。例如，我们可以变换矩阵的维数。对于一般张量，我们把这个操作称为置换。调用 permute 可以任意地重新排列输入张量的维数。例如，对形状 (2,5) 的矩阵调用 permute(1,0) 可以得到形状 (5,2) 的矩阵。为了索引到置换矩阵中，我们使用张量 [j,i] 而不是张量 [i,j] 来访问元素。

其次，张量使添加或删除额外维度变得非常容易。比如形状为 (1,5,2) 的张量和形状为 (5,2) 的张量存储的内容量一样大。我们希望在不改变数据的情况下增加或减少张量的维数，我们将通过一个 view 函数来实现这一点：对初始形状为 (5,2) 的张量使用 view(1,5,2)。(5,2) 矩阵中的元素 [i,j] 现在是三维张量中 [0,i,j]。

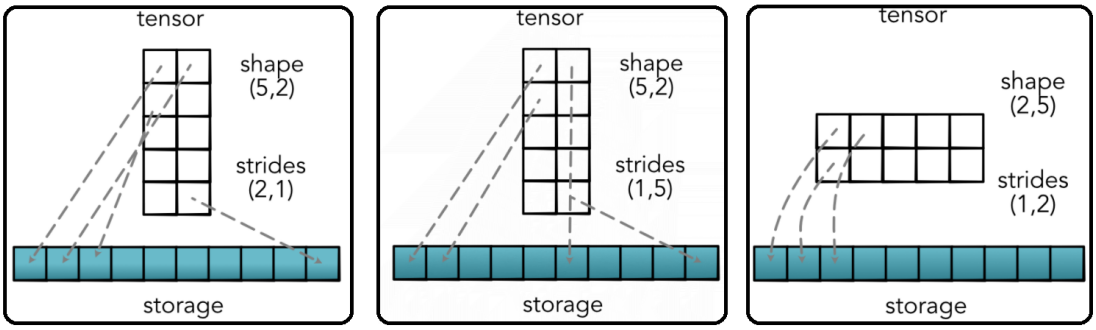
上述两种操作都不会改变输入张量本身。view 和 permute 都是张量 trick，即只修改我们如何看待张量的操作，而不修改其任何数据。另一种说法是，它们不以任何方式移动或复制数据，只移动或复制向量的外部包装（即访问方式会改变，但是数据本身并不改变）。

5.1.2 张量 strides

构建好张量后，用户只需要知道张量的 shape 和 size。然而，我们需要跟踪一些重要的实施细节，为了让代码更简洁，我们需要将张量的内部数据从面向用户的部分中分离出来。除了形状，Minitorch.TensorData 还管理张量存储 (storage) 和步幅 (strides)：

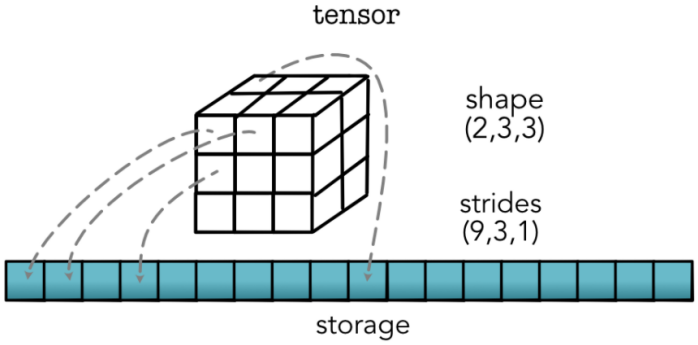
Storage 是数据保存的地方，它永远是一个 1-D 数组，不管张量的维度和 shape 是什么样子，它永远是 1 维的。也就是说，我们可以用不同的张量 shape 来指示同一个底层数据。

Strides 是一个 tuple，提供从用户的索引到一维存储位置的映射。比如下图：



注意 strides 和 storage 的排列方式是相关的。先看左图，每列与相邻列之间的存储位置是相邻的，而每行和下一行之间的位置相差 2。再看中图，它的 shape 也是 (5,2)，但是它的存储排列变了，相邻行之间紧挨着，而列之间相差 5。左图这种情况叫做连续映射 (contiguous mapping)，因为它满足自然增长顺序（即 bigger strides left，strides 中的左边元素值大于右边的元素值）；中图是非连续 (non-contiguous)strides，因为排列是先从上到下，再从左到右（可以参考代码中 is_contiguous 函数辅助理解）。右图也是非连续 strides，strides 为 (1,2)。

strides 很容易扩展到三维情况：



对于 strides 为 (s_1, s_2) 的张量，如果想要查找 tensor$[i,j]$，则为 storage$[s_1 \cdot i, s_2 \cdot j]$。

5.1.3 完成 task2.1

minitorch.TensorData 实现了 tensor，包括索引、存储和转换，以及 strides。在我们开始使用面向用户的 minitorch.Tensor 之前，需要先把三个函数实现。由于这些函数比较简单，所以直接给出答案：

```
def index_to_position(index, strides):
  pos = 0
  for i, v in enumerate(index):
      pos += v * strides[i]
  return pos
```

to_index 函数实现如下：

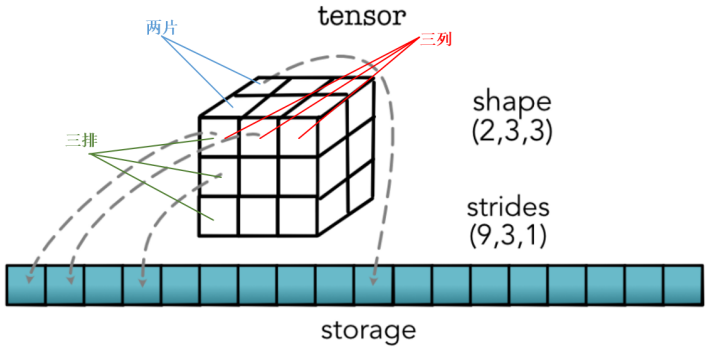
```
def to_index(ordinal, shape, out_index):
  for d in range(len(shape) - 1, -1, -1):
      out_index[d] = ordinal % shape[d]
      ordinal = ordinal // shape[d]
```

注意 to_index 并不一定是 index_to_position 的逆版本，如果是连续映射则互为逆版本。

TensorData.permute 实现如下，返回的是一个新的 TensorData 对象：

```
def permute(self, *order):
  return TensorData(self._storage, tuple([self.shape[i] for i in order
      ]), tuple([self.strides[i] for i in order]))
```

我们可以以一个三维张量为例，来解释 permute 函数的机理。我们以上面图示中的三维 tensor 为例，shape 为 (2,3,3),strides 为 (9,3,1)，我们记做“两片三排三列”。



如果我们把 shape 的维度变换一下，*order 为 (1,0,2)，则新的 shape 改为 (3,2,3)，它就不再是一个连续 strides 了。现在记做“三排两片三列”，当我们想访问第一排第二片第三列的数据，则计算式为：0x3+1x9+1=10。可以看到，strides 的元素顺序和 shape 都是进行同样的交换。

5.2 张量广播

5.2.1 什么是张量广播

广播 (Broadcasting) 使得张量很容易使用且有效，特别是在 zip 操作中（回忆前面我们实现过的 zipWith 函数）。到目前为止，我们所有的 zip 操作都假设两个大小和形状完全相同的输入张量。然而，有许多有趣的例子来 zip 两个不同大小的张量。

也许最简单的情况是，我们有一个大小为 3 的向量，并希望在向量的每个位置都加上一个标量常数：

```
vector1 + tensor([10])
```

但是这种操作将失败，因为我们正在向具有形状为 (3,) 的向量添加形状为 (1,) 的张量。我们可以写作如下方式，但是这样很低效：

```
vector1 + tensor([10, 10, 10])
```

广播其实是一种协议，它允许我们自动地将第一个表达式的解释来暗示第二个表达式。在 zip 函数内，当用形状 (3,) 的向量 zip 10 时，我们假设 10 是一个形状为 (3,) 的向量。

5.2.2 三条规则

规则一：通过假设维度被拷贝了 n 次，任何 size 为 1 的维度都可以与 size 为 n>1 的维度来 zip。

把规则一应用于 shape 为 (4,3) 的矩阵：

```
matrix1 + tensor([10])
```

在这个例子中，我们试图把 zip 作用于 shape 为 (4,3) 的矩阵与 shape 为 (1,) 的 1 维向量，这里不仅 shape 不同，维度也不同。然而，给数据增加一个额外的 shape 为 1 的维度并不改变张量的大小，因此我们允许我们的协议这么做。如果我们要加上一个空维度并且应用两次规则一，即从维度的 (1,1) 扩充为 (4,1)，然后再扩充为 (4,3)——则我们可以把上面的表达式理解为下面的简化版本：

```
matrix1 + tensor([10] * 12, shape=(4, 3))
```

规则二： shape 为 1 的额外维度可以添加到一个张量中，以确保与另一个张量的维度数量相同。

最后，还有一个问题是在哪里添加空维度。在上述示例中，并不会出现这种问题；但在更复杂的情况下可能会出现。因此，我们引入另一条规则：

规则三：任何 size 为 1 的额外维度只能隐式添加到 shape 的左侧。

这条规则的影响是使流程易于遵循和复制，我们总是知道最终输出结果的 shape。例如：

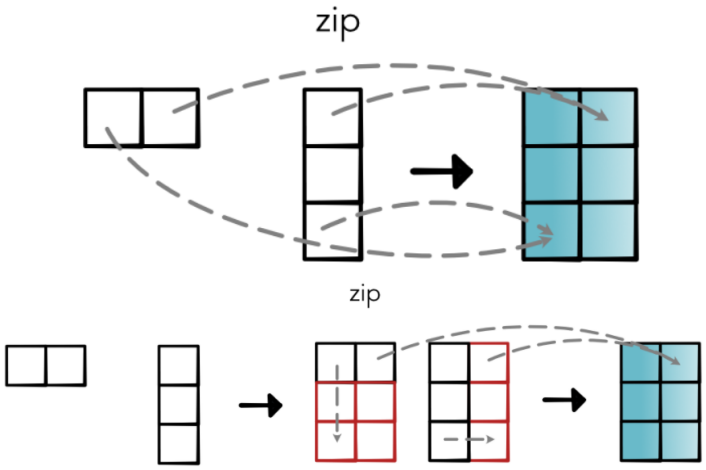
```
# 由于 (4,3) 和 (4,) 不匹配，所以这会失败
matrix1 + vector2.view(4)
# 下面两个表达式是相等的
matrix1 + vector1
```

```
matrix1 + vector1.view(1, 3)
```

我们可以根据需要多次应用广播：

```
# 输出结果的shape是(3, 2)
tensor1.view(1, 2) + tensor2.view(3, 1)
```

示意图如下：



一个更复杂的例子：

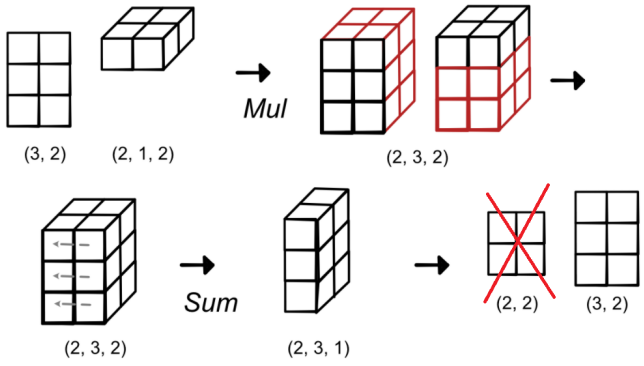
```
# 输出结果的shape为(7, 2, 3, 5)
tensor1.view(1, 2, 3, 1) + tensor2.view(7, 2, 1, 5)
```

最后我们提两点：

- 广播只是与 shape 有关。它没有以任何方式考虑 strides——这纯粹是一个 high-level 的协议。
- 广播对 backward 过程有一定影响，但任何 task 都不需要管它，因为基础代码库中会实现这些内容。

5.2.3 矩阵相乘的例子

考虑一个 3×2 的矩阵和一个 2×2 的矩阵，我们可以先把第二个矩阵变为 $2\times1\times2$ 的张量，然后元素相乘。之后再把对应的列相加，就能得到最终结果（我认为应该是生成 3×2 的矩阵，而非 2×2）：



不过这样的效率并不会很高，因为需要创建新的 tensor。

5.2.4 完成 task2.2

shape_broadcast 函数就是实现上面的三个 rules，首先需要比较 shape1 和 shape2 的 size，然后在 size 低的 shape 上扩充 1。之后，将两个 shape 的对应元素相互比较，取大的元素作为输出结果：

```
def shape_broadcast(shape1, shape2):
  l = max(len(shape1), len(shape2))
  if len(shape1) > len(shape2):
      shape2 = [1 for i in range(l - len(shape2))] + list(shape2)
  else:
      shape1 = [1 for i in range(l - len(shape1))] + list(shape1)
  ans = []
  for i in range(l):
      if shape1[i] != shape2[i] and shape1[i] != 1 and shape2[i] != 1:
          raise IndexingError('violation␣of␣broadcasting␣rules')
      ans.append(max(shape1[i], shape2[i]))
  return tuple(ans)
```

我们要明确的是，shape(2,5) 和 shape(3,3) 没法进行广播；shape(2,5) 和 shape(1,5) 是可以进行广播的。

broadcast_index 函数根据广播规则将 big_shape 的 big_index 转换为较小的 shape 的 out_index。它可能比给定的 shape 更大或有更多维度。其他的维度可能需要映射到 0 或删除。

这是什么意思呢，首先，shape1(1,5) 有可能被扩展为 shape2(2,5)，此时它的第 1 个元素比 shape1 的第一个元素更大；或者扩展为 (3,2,5)，此时它维度比 shape1 大。

扩展就相当于复制，我们设 tensor1 的 shape 为 (2,5)，扩展为 tensor2，shape 为 (3,2,5)。当我们想索引 shape 为 (3,2,5) 的 tensor2 中的某个坐标为 [2,1,4] 的元素时，我们完全不需要考虑第 1 个元素索引，而是直接索引 tensor1[1,4] 即可。

但是，如果 tensor1 的 shape 为 (1,5)，扩展为 tensor2，shape 为 (3,2,5)，则其实第二个维度也是复制过去的，所以 tensor2[2,1,4] 引用到 tensor1 时，第一个元素要设置为 0，即 tensor1[0,4]。该函数实现如下：

```
def broadcast_index(big_index, big_shape, shape, out_index):
  for i in range(len(shape)):
    offset = i + len(big_shape) - len(shape)
    out_index[i] = big_index[offset] if shape[i] != 1 else 0
```

5.3 张量操作

高阶张量的运算并不容易，如果对 python 中的高阶函数和语法比较熟悉，会对这比较有帮助。

5.3.1 基本操作

现在我们想重新实现所有关于张量的数学运算。我们的目标是让库用户感到简单和直观，我们可以将这些操作分解为一元变换：

```
# 返回与tensor_a相同shape的张量。
tensor_a.log()
tensor_a.exp()
-tensor_a
...
```

二元操作：

```
tensor_a + tensor_b
tensor_a * tensor_b
tensor_a - tensor_b
```

reduction 操作：

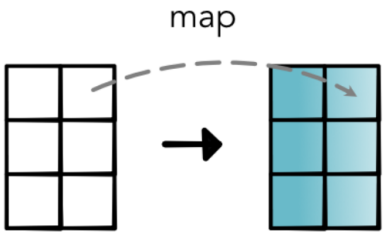
```
#返回一个新的张量，其中dim-1的大小为1，表示dim-1上的sum/mean
tensor_a.sum(1)
tensor_a.mean(1)
...
```

这些操作会借助 map 等函数来进行实现，我们只是先知道有这些函数，暂时先不考虑实现过程。

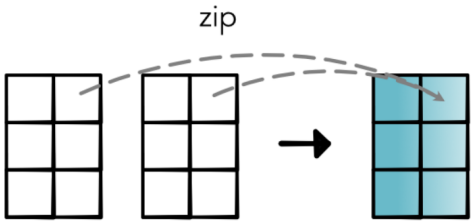
5.3.2 核心操作

我们可以单独实现这些操作中，但我们也可以懒一点，注意，这些操作具有结构上的相似性。这些操作看起来非常像我们在 Fundamentals 中实现的高阶函数。

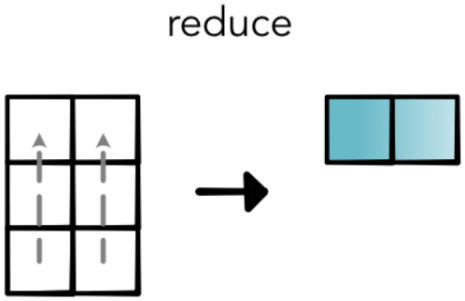
有些函数只需能够访问到张量中的每个位置，它不需要知道张量的形状或大小。我们可以将这些操作视为应用 map 函数：



有些函数需要在输入张量之间成对操作，这种操作称为 zip 操作。如果我们假设张量具有相同的大小和形状，这种类型的操作只是将这两个张量对齐，并对每对元素应用一个运算：



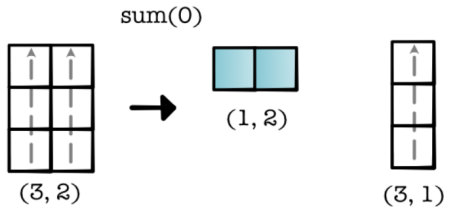
第三类函数需要去把 cells 给组合到一个单个 tensor 中。下图，我们将 shape 为 (3,2) 的张量生成 shape 为 (1,2) 的输出，这相当于输出中的每个元素都是 shape 为 (3,1) 的张量通过 reduce 来得到的。



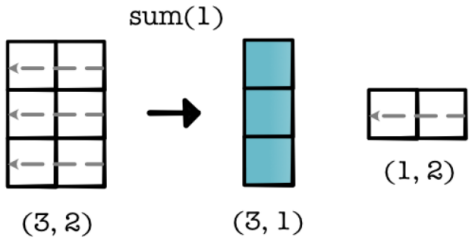
5.3.3 Reduction 操作

鉴于 Reduction 操作最难理解，我们在这个小节里着重分析。

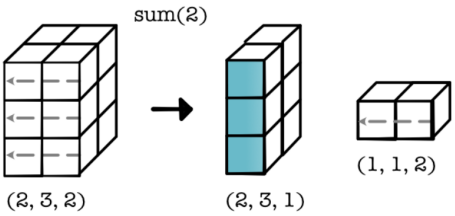
我们需要选择一个 reduction 的维度，例如去 reduce 维度 0：



以及去 reduce 维度 1：

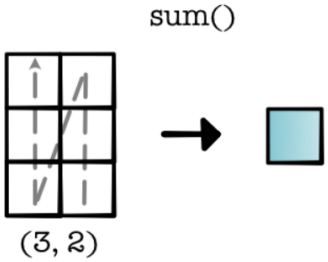


在更高维的情况，例如下面对维度 2 进行 reduce：



在实际操作中，我们可以把最终生成的输出中的每个元素进行访问，计算 reduce 得到的该元素的值。例如上图，最终生成的是 size 为 6 的张量，我们需要依次访问这 6 个位置。

最后，说一种特殊的 reduction 操作，我们会对整个张量进行 reduce，得到一个标量：



这些实现在当前都需要遍历操作，因此效率比较低。在下一个 Module 中我们会介绍更有效的实现方案。

5.3.4 完成 task2.3

task2.3 可以分为两部分，第一部分是三个主要函数：tensor_map、tensor_zip 以及 tensor_reduce 函数；第二部分是使用这三个主要函数来实现的其他具体函数，例如加减乘除（TensorFunction 的 forward 功能）。

第一部分

tensor_map 会返回一个函数，看到这个函数的参数列表，我们可以想到它需要用到我们之前实现的那几个函数。由于 out 参数是 array 类型的，所以我们只需要循环遍历 array，来计算生成 out 中的每一个元素即可。我们通过 to_index 函数把索引 i 映射到 out_index 坐标上，之后根据广播机制找到它对应在 in 的坐标，然后将 in 的坐标转为 in 存储上的位置，然后执行映射即可：

```
def tensor_map(fn):
  def _map(out, out_shape, out_strides, in_storage, in_shape,
      in_strides):
  out_index = np.array(out_shape)
  in_index = np.array(in_shape)
  for i in range(len(out)):
      to_index(i, out_shape, out_index)
      broadcast_index(out_index, out_shape, in_shape, in_index)
      data = in_storage[index_to_position(in_index, in_strides)]
      map_data = fn(data)
      out[index_to_position(out_index, out_strides)] = map_data
  return _map
```

实现了上面这个函数以后，tensor_zip 就很容易了，这里不再赘述。

reduce 中，我们依次处理 out 的每个元素。将序数 i 转换到 out 的 storage 上的存储位置。之后处理要被 reduce 的维度。

```
def tensor_reduce(fn):
  def _reduce(out, out_shape, out_strides, a_storage, a_shape,
      a_strides, reduce_dim):
```

```
    out_index = np.array(out_shape)
    for i in range(len(out)):
      to_index(i, out_shape, out_index)
      o_index = index_to_position(out_index, out_strides)
      for j in range(a_shape[reduce_dim]):
        a_index = out_index.copy()
        a_index[reduce_dim] = j
        pos_a = index_to_position(a_index, a_strides)
        out[o_index] = fn(a_storage[pos_a], out[o_index])
  return _reduce
```

第二部分

在当前新的版本中，所有的函数都会在 make_tensor_backend 里进行收录。

由于基本函数的实现都很简单，我们不讲解代码。只提一下，因为 Tensor 的构造中需要保证 backend 不为 None，所以 Permute.forward 函数要传入 backend：

```
class Permute(Function):
  def forward(ctx, a, order):
      return Tensor(a._tensor.permute(*order),backend=a.backend)
```

顺利通过 task2.3 的所有的 test，就说明我们的实现基本没有问题了。

5.4 张量自动微分

5.4.1 张量 Variables

接下来，我们考虑张量框架中的自动微分。我们现在已经从标量和导数转移到向量、矩阵和张量。这意味着会涉及可怕的多元微积分学。然而，我们实际上并不需要复杂的术语或大量的专业数学。事实上，除了一些名称的更改，我们已经在 Autodiff 中构建了几乎所有我们需要的东西。

关键的想法是，就像我们有标量和标量函数一样，我们需要构造张量和张量函数（我们称之为函数）。这些新对象的行为与对应标量对象非常相似：

- Tensors 不能被直接操作，而是需要通过一个 Function 来计算。
- Functions 必须实现 forward 和 backward 函数。
- 这些计算都要被追踪，并且允许通过链式法则进行反向传播。

需要了解的主要新术语是梯度 (gradient)。正如张量是标量的多维数组一样，梯度也是这些标量的导数的多维数组。考虑下面的代码，首先是以前的标量函数：

```
# Assignment 1 notation
out = f(a, b, c)
```

```
out.backward()
a.derivative, b.derivative, c.derivative
```

然后是张量函数：

```
# Assignment 2 notation
tensor1 = tensor(a, b, c)
out = g(tensor1)
out.backward()
# shape (3,)
tensor1.grad
```

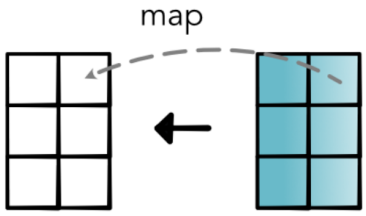
现在，tensor1 的梯度是一个张量，存储了它每个元素的微分。backward 也不再使用用 d_{out} 作为参数，而是使用 $grad_{out}$ 作为一个参数，这个参数由所有的 d_{out} 组成。

你会发现梯度和多变量 (multivariate) 有很多不同的表示法。在这个 Module 中你应该忽略这些并按照你所知道的关于导数的术语和方法来理解，事实证明你可以在不需要考虑更高的维度的前提下完成大多数机器学习。

只要我们把 gradient 当做 tensors 的导数，我们就可以明白通过单变量的规则来计算梯度多么容易了。

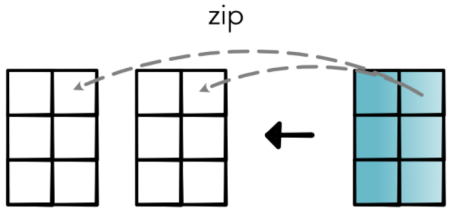
map

对于 tensor 中的每个元素 x，都有对应的 $g'(x)$ 和 d_{out}，我们需要把它们都计算出来，然后一一对应过去：



zip

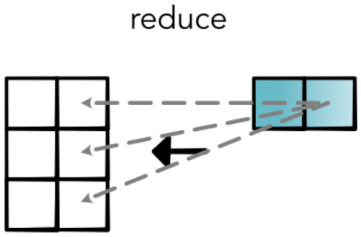
将 $g'_x(x,y) \times d_{out}$ 和 $g'_y(x,y) \times d_{out}$ 计算后返回到前面的张量上。我们只需要计算每个标量位置的导数，并应用一个 mul 映射：



reduce

给定一个 tensor，reduce 会应用一个聚集 (aggregation) 操作，为了简单起见，我们分析一下加法 reduction。当计算 $x_1 + x_2 + ... + x_n$ 时，对 x_i 求导得到的结果都是 1。因此，backward 中任

何位置计算得到的导数都是 d_{out}，这意味着计算梯度时我们仅仅需要去发送 d_{out} 给每个位置，如下图所示：



乘法 reduction 则需要计算导数，实现中我们再进行细说。

5.4.2 完成 task2.4

需要注意的是 sigmoid 函数的 backward() 和 ReLU 函数的 backward() 实现方式有较大不同：

```
class Sigmoid(Function):
  def forward(ctx, a):
    b = sigmoid_map(a)
    ctx.save_for_backward(b)
    return b
  @staticmethod
  def backward(ctx, grad_output):
      b = ctx.saved_values
      return mul_zip(grad_output, mul_zip(b, add_zip(tensor([1.0]),
          neg_map(b))))
```

ReLU 函数的实现方法：

```
class ReLU(Function):
  @staticmethod
  def backward(ctx, grad_output):
      a = ctx.saved_values
      return relu_back_zip(a, grad_output)
```

Add 和 Mul 的实现方法参考如下，这里需要用到 expand 函数，我们下一章再详细讲解该函数。

```
class Add(Function):
  @staticmethod
  def forward(ctx, t1, t2):
      ctx.save_for_backward(t1, t2)
      return add_zip(t1, t2)
  @staticmethod
  def backward(ctx, grad_output):
```

```
        a, b = ctx.saved_values
        grad_a, grad_b = a.expand(grad_output), b.expand(grad_output)
        return grad_a, grad_b
```

只要我们能通过 2.4 的全部测试，就说明我们的实现大致是正确的。

5.5 训练与测试

我们需要实现 project/run_tensor.py 里面的功能。

NetWork 类的 forward() 函数实现较为简单，和以前基本一样：

```
class Network(minitorch.Module):
  def forward(self, x):
    middle = self.layer1.forward(x).relu()
    end = self.layer2.forward(middle).relu()
    return self.layer3.forward(end).sigmoid()
```

Linear 的 forward() 函数需要多次用到 view 函数：

```
def forward(self, x):
  x = x.view(*x.shape, 1)
  w = self.weights.value.view(1, *self.weights.value.shape)
  x = x * w
  a = x.sum(1)
  a = a.view(x.shape[0], self.out_size)
  a = a + self.bias.value.view(1, *self.bias.value.shape)
  return a
```

当我学习率设置的只有 0.5 的时候，训练几乎损失率不变，而当我设置学习率为 1.0 时，损失率下降就比较正常了。

我们下一章再介绍 Linear 的 forward() 函数的实现原理。

6. Tensors 架构详解

本章是上一章的补充,介绍张量部分的基础代码架构和实现。本章主要包含了四个文件:*tensor_data.py*, *tensor_ops.py*, *tensor.py*, *tensor_functions.py*, 同时也会包括 *run_tensor.py* 的代码。上一章详细介绍过的内容本章不会再赘述, 本章主要介绍基本原理和基础代码。

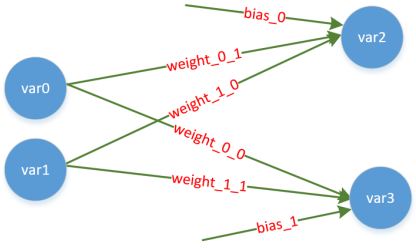
6.1 整体功能与基本布局

6.1.1 基本概念的回顾

先进行一些基本的回顾。

Variable 是变量的基类，变量有两种，Scalar 和 Tensor。对 Variable 使用 backward() 后会调用全局函数 backpropagate()，该函数会根据输入输出关系，首先将所有的 Variable 按照拓扑顺序来排序，然后从后往前依次处理——对于叶节点，使用 accumulate_derivative 来积累导数；对于其他节点，将计算的导数保存/累加到 derivs 字典里，供其他节点访问和使用。

Module 是模型类，比如 Linear 就继承自 Module。Parameter 类是网络参数类（比如权重 weight 和偏置 bias），包含了参数名和参数值。Scalar 对象和 Tensor 对象都可以作为一个参数，在 Linear 模型里，使用 Scalar 实现的 Linear 中每个元素都代表一个参数（下图中每个 weight 和 bias 都是一个参数），而 Tensor 实现的 Linear 中整个权重矩阵作为一个 Tensor 参数（下图中所有 weight 构成一个 (2,2) 的参数，所有 bias 构成一个 (1,2) 的参数）：



6.1.2 张量类和方法的布局与结构

我们先把 miniTorch 工程中的全部类和外部函数列一下：

```
# tensors.py
class Tensor
# tensor_data.py
class IndexingError(RuntimeError)
def index_to_position(index, strides)
def to_index(ordinal, shape, out_index)
def broadcast_index(big_index, big_shape, shape, out_index)
def shape_broadcast(shape1, shape2)
def strides_from_shape(shape)
class TensorData
# tensor_functions.py
class Function(FunctionBase)
def make_tensor_backend(tensor_ops, is_cuda=False)
def zeros(shape, backend=TensorFunctions)
def rand(shape, backend=TensorFunctions, requires_grad=False)
def _tensor(ls, shape=None, backend=TensorFunctions, requires_grad=
    False)
def tensor(ls, backend=TensorFunctions, requires_grad=False)
def grad_central_difference(f, *vals, arg=0, epsilon=1e-6, ind=None)
def grad_check(f, *vals)
# tensor_ops.py
def tensor_map(fn)
def map(fn)
def tensor_zip(fn)
def zip(fn)
def tensor_reduce(fn)
def reduce(fn, start=0.0)
class TensorOps
```

可知，除了 IndexingError 报错类以及 TensorOps 这个收录了三种 Tensor 操作的类，主要一共有三个类：Tensor、TensorData、Function。

函数按功能分为几类：

- 构建张量的函数：zeros、rand、_tensor、tensor
- 张量的 storage、strides 和 shape 的索引、转换之类的函数：tensor_data.py 的所有全局函数
- 通过中心差分来验证梯度的函数：grad_central_difference、grad_check

- 张量函数：map、zip 和 reduce
- 张量具体函数：make_tensor_backend 里定义的函数

6.2 数据的存储与坐标索引变换

6.2.1 坐标变换函数

我们先看与基本数据存储有关的函数。

strides_from_shape 函数是程序基础代码给出的，举两个例子即可很清楚地知道这个转换关系：

$$shape(2,3) \Longrightarrow strides(3,1) \tag{6.2.1}$$

$$shape(3,2,4) \Longrightarrow strides(8,4,1) \tag{6.2.2}$$

broadcast_index 使我们自己实现的函数。比如 $shape1(3,1,2)$ 扩充为了 $shape2(2,3,3,2)$，我们想在广播后的某个元素（例如 (1,1,2,1)）找到在广播前的对应项，则：

```
broadcast_index((1,1,2,1), shape2, shape1, out_index)
#输出： out_index(1,0,1)
```

shape_broadcast 用来得到广播后的 shape，其实就是扩展到更大的维度或 shape，例如：

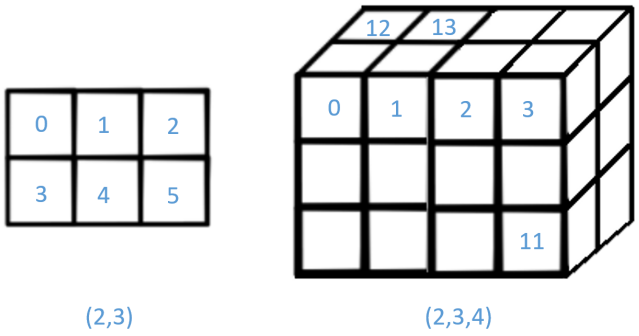
$$shape1(2,1) \quad shape2(3,1,5) \Longrightarrow out(3,2,5) \tag{6.2.3}$$

index_to_position 将坐标计算到对应的 storage 索引上。

to_index 按照顺序将序数变为索引，例如：

$$shape(2,3) \quad i = 4 \quad out = (1,1) \tag{6.2.4}$$

$$shape(2,3,4) \quad i = 13 \quad out = (1,0,1) \tag{6.2.5}$$



6.2.2 TensorData 类

TensorData 类的初始化函数很简单明了，不再细讲。

其余，主要有四个函数需要注意：is_contiguous、index、indices 和 permute。set 和 get 函数是通过 index 计算坐标然后进行数据读写的。

indices 通过 yield 来定义一个迭代器，依次得到每个元素的索引。

is_contiguous 根据 strides[i]>strides[i+1] 来判断是否为连续存储。

index 函数首先将输入转换为 array 类型，然后检查 index 的每一项是否合规（是否小于对于维度的 shape 值、是否不为负数），如果合规，则调用 index_to_position 来索引到 storage 位置上。

permute 函数通过置换 shape 和 strides 来生成新的 shape 表示形式。

6.3 张量与标量的函数调用对比

尽管我们已经熟悉了 Variable 类和 Scalar 类的各个函数调用，但是 Tensor 类显得更复杂一些。我们现在分析一下它们的区别。

Variable 类中有一个 history 成员（为 None 表示为常量），该成员用来记录生成该 Variable 的函数以及函数上下文 (Context)。Context 类中有两个用来得到存储的上下文的函数：

```
# 得到生成Scalar对象的函数的输入
saved_values
# saved_values的别名（其实程序里暂时并未使用）
saved_tensors
```

FunctionBase 实现了两个函数，一是 apply，用来计算并生成 Variable；二是 chain_rule，用来执行链式法则。

ScalarFunction 继承自 FunctionBase，其子类需要实现 forward() 和 backward() 函数。

Function 继承自 FunctionBase，之后在 Backend 类内定义了 Function 的内部类子类。

ScalarFunction 和 Function 都实现了 data 函数，但是目前好像并没有什么函数去调用 data 函数。

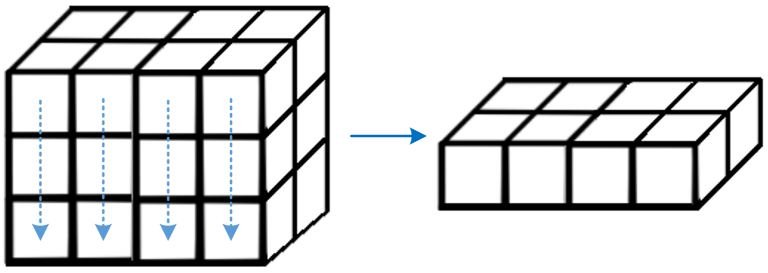
6.4 张量基本操作与构造

6.4.1 张量的基本操作

本节在前面已经叙述过了，主要是 map、zip 和 reduce 这三个函数。

map 和 zip 函数较为简单，我们还是关注一下 reduce 函数，我们需要实现的部分是 tensor_reduce() 函数的计算。我们先看 reduce() 函数，该函数需要生成 out 的存储。

比如对于 shape(2,3,4)，我们想 reduce dim 1，也就是：



所以会创建一个 out_shape 为 (2,1,4)，然后通过 zeros 函数创建一个元素全为 0 的 Tensor 对象，之后把存储的值全部设定为 start（reduce 为 add 时，start=0.0；reduce 为 mul 时，start=1.0）。

```
out = a.zeros(tuple(out_shape))
out._tensor._storage[:] = start
```

对于 out 来说，每个元素都是输入的 dim 1 上的对应元素的运算（和或者乘积）。我们要把输入的 dim1 上的元素都访问并且计算（tensor_reduce 函数）：

```
# 这里的 j 会取值为 [0,1,2]
for j in range(a_shape[reduce_dim]):
  # 找到 out_index 在 a 的位置
  a_index = out_index.copy()
  a_index[reduce_dim] = j
  pos_a = index_to_position(a_index, a_strides)
  # reduce操作
  out[o_index] = fn(a_storage[pos_a], out[o_index])
```

6.4.2 张量的构造

在 tensor_function 函数里，构造张量主要有 zeros()、rand()、_tensor() 和 tensor() 函数，这两个函数都调用了 Tensor.make 函数。

提一句，Variable 里有 requires_grad_ 函数，该函数会构造 History（没有 History 意味着是一个常量），而其实不管该函数的传入参数是什么，都会创建历史。（我怀疑是由于 rand 一般都是用来生成最初的参数的，所以都是叶子，也就不需要给 last_fn，但是叶子也不能是常量，否则无法更新参数。我个人认为这是一个历史遗留问题，也很好解决）。

tensor() 函数定义了两个内部函数：shape 通过递归的方法来得到形状，flatten 把 list 或者 tuple 对象展开成一维。它们都需要判断是否是 tuple 或者 list 类型，来区分线性结构和单个数据：

```
# 判断 ls 是否是 list 或者 tuple 类型
isinstance(ls, (list, tuple))
```

Tensor.make 函数就是调用了 Tensor 构造器来构造一个 Tensor。

6.5 张量类

Tensor 类继承自 Variable 类。

6.5.1 基础函数

成员函数 shape、size 和 dims 都是返回张量数据的相关属性。

type 函数主要是为了赋值 backend。

_ensure_tensor 函数将数字转换为 Tensor 对象。

grad 函数得到 derivative 值。

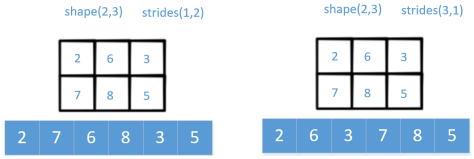
6.5.2 计算函数

以下函数都是直接或间接调用 backend 的内部函数来实现的，都需要调用内部函数的 apply 方法来调用 forward() 生成新的 Tensor 对象。我们本节并不细讲 backward() 的含义，只关注与 Tensor 的创建和解读（shape、strides）有关的内容。

contiguous 函数调用 backend.Copy，来返回一个连续的张量。Copy 函数其实就是调用了：

```
tensor_ops.map(operators.id)
```

之所以能够变为连续，是因为 to_index 是将序数根据 shape 转换为坐标，我们以下图为例：



右边是调用 contiguous 以后生成的 Tensor，在使用 map 时，以 storage[2] 的位置为例，在 storage 的索引为 2 的位置在 shape(2,3) 的 index 为 (0,2)（由 to_index 函数计算得到），转换到左图上为 storage[4]。

view 函数就是调用 backend.View 函数，其实就是根据 shape 来解释存储的索引。它会判断是否为连续，只有连续的张量才能进行 View。

permute 调用 backend.Permute，同时改变 shape 和 strides。

6.5.3 expand 函数

这里我们重点将一下 expand 函数。expand 是一个很重要的函数，可以用来允许广播后的反向传播。当输入 forward 的 tensor 的 shape 与 backward 输出的 shape 不同时，就需要调用该函数。expand 只会在 Add 和 Mul 函数中被调用，expand 的参数 other 就是 backward tensor。

expand 首先判断 other 的 shape 和自身的 shape 是否一致。如果 backward 的 tensor 比自身的小，则进行广播。如果广播后的 shape 仍然不同，则需要 reduce 额外的维度。

我们用下面的程序来说明：

```
import minitorch
import minitorch.tensor as tensor
a = tensor([1,2])
b = tensor([[3],[4]])
c = a * b
print(c)
```

此时打印出来的 c 张量为：

```
[[3.00 6.00]
 [4.00 8.00]]
```

然后调用：

```
c = c.sum()
print(c)
```

打印出来：

```
[21.00]
```

调用相乘时，其实就是：

$$\begin{bmatrix}1 & 2\\1 & 2\end{bmatrix}\times\begin{bmatrix}3 & 3\\4 & 4\end{bmatrix}=\begin{bmatrix}3 & 6\\4 & 8\end{bmatrix} \tag{6.5.1}$$

在这个过程中，expand 都会执行广播和 reduce。而如果 a 和 b 是如下设置，则 expand 并不会执行广播和 reduce：

```
a = tensor([[1,2],[1,2]])
b = tensor([[3,3],[4,4]])
```

我们回到最初情况，调用 sum() 以后，生成 c，对 sum() 调用 backward() 会生成 grad_output 为：

```
[[1.00  1.00]
 [1.00  1.00]]
```

之后根据链式法则，对 Mul 调用 backward()，此时：

```
def backward(ctx, grad_output):
  a, b = ctx.saved_values
  grad_a, grad_b = a.expand(b * grad_output), b.expand(a * grad_output
      )
  return grad_a, grad_b
```

得到的中间结果：

```
b * grad_output:
[[3.00  3.00]
 [4.00  4.00]]
a * grad_output:
[[1.00  2.00]
 [1.00  2.00]]
```

然后需要变回到 a 和 b 本来的维度，此时就需要 reduce（严格来说，是 _add_reduce，因为这涉本质上还是多元复合函数的微分）。我们可以用标量来重新写作一下上面的过程：

```
a1 = Scalar(1)
a2 = Scalar(2)
b1 = Scalar(3)
b2 = Scalar(4)
c1 = a1 * b1
c2 = a2 * b1
c3 = a1 * b2
```

```
c4 = a2 * b2
out = c1 + c2 + c3 + c4
```

6.6 张量函数

backend.View 函数会调用 Tensor.make 函数。

我们介绍一下 backend.Permute 函数。假设原来的 shape1(2,3,4)，forward() 传入的 order 是 (2,0,1)，则会交换为 (4,2,3)。backward() 函数中，首先读取 (2,0,1)，然后先生成一个序列数组 [0,0,0]，之后按顺序遍历并赋值，得到 [1,2,0]，即新的顺序。

backend.MatMul 函数因为里面调用的 matrix_multiply() 函数并没有实体，所以我们暂时不管该函数（暂时我们也用不到该函数）。

由于这些函数形式上大致与标量相同，不同的几个函数我们也都单独进行了介绍（比如 View 和 Permute），所以这里也不再赘述。

6.7 训练

见 run_tensors.py 文件，我们以训练一轮为例：

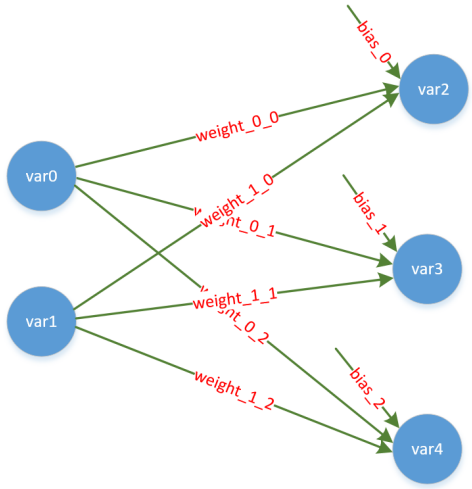
```
X = minitorch.tensor(data.X)
y = minitorch.tensor(data.y)
# 训练过程
optim.zero_grad()
# Forward
out = self.model.forward(X).view(data.N)
prob = (out * y) + (out - 1.0) * (y - 1.0)
loss = -prob.log()
(loss / data.N).sum().view(1).backward()
total_loss = loss.sum().view(1)[0]
losses.append(total_loss)
# Update
optim.step()
```

我们先来看 NetWork.forward() 函数，需要构造三层 Linear 模型。我们看一下 Linear 的 forward() 函数：

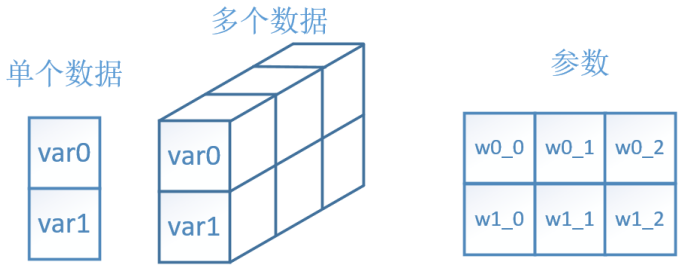
```
class Linear(minitorch.Module):
  def forward(self, x):
    x = x.view(*x.shape, 1)
    w = self.weights.value.view(1, *self.weights.value.shape)
    x = x * w
    a = x.sum(1)
```

```
    a = a.view(x.shape[0], self.out_size)
    a = a + self.bias.value.view(1, *self.bias.value.shape)
    return a
```

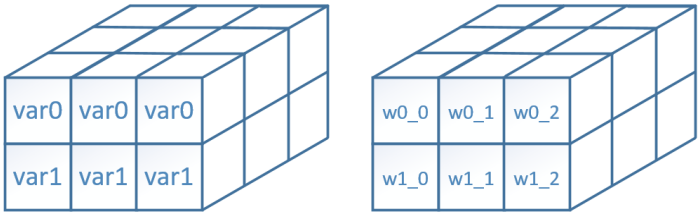
以下图为例，设 self.weights 的 shape 为 (2,3)；self.bias 的 shape 为 (3,)：



输入 x 设 shape(2,) 通过 view 函数 shape 变为 (2,1)，w 变为 shape 为 (1,2,3)，为什么要这么操作，这是因为我们可能输入数据并不只有一组，单个数据的 shape 是 (2,)；而如果我们要同时输入 100 个数据去做训练，则输入的 x 的 shape 就变为了 (100,2)，通过 view 函数以后就会变为 (100,2,1)，我们设同时输入的数据量为 3：



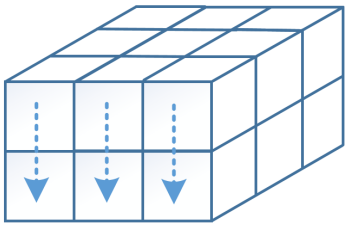
在做乘法时，根据广播机制，会得到：



对于一组数据中的第一个数据，计算得到：

```
var0*w0_0   var0*w0_1   var0*w0_2
var1*w1_0   var1*w1_1   var1*w1_2
```

调用 sum(1) 的加和方向为：



之后再调用 view 来转换为和 bias 一样的 shape，再与 bias 相加，得到输出。

至此，整个架构的解释就到此结束了。

7. Efficiency-提高效率

miniTorch 中最难理解的内容都已经讲完了，本章我们学习 *Module-3*，重点在于利用张量编写快速代码，首先在标准 *CPU* 上编写，然后使用 *GPU* 来完成。本章要不要细讲我是犹豫了一段时间的，一是由于我本人能力也有限，很难写出很高质量的实现代码，二是实现的方案可能有很多种，我怕会陷入思维局限中，但最终决定为了完整性还是好好实现一下。

7.1 并行化

我们前两章关于 Tensor 其实只是为了把网络结构更快地写代码，但并不会提高程序的运行效率。本章我们会关注利用 tensor 来实现更快的代码，首先会在标准 CPU 上实现，然后会使用 GPU。

7.1.1 并行化与 Numba

我们需要足够熟悉 Numbda 的使用方法 [15]。

Notice 7.1 (Numba 与自动并行化) 对 jit() 设置并行选项，可以尝试将代码自动并行化（类似 openMP，只支持 CPU）。并行化过程是自动的，无需用户修改代码（与 Numba 的 vectorize() 或 guvectorize() 机制不同，这些机制需要用户自己创建并行核）。使用也较为简单，在函数前加上：

```
@njit(parallel=True)
```

我们可以再次利用这样一个事实，即我们已经围绕一组核心的、通用的操作构建了我们的基础设施（比如加减运算），这些操作在整个代码库中都会使用。

高阶映射，即 map 在以前我们是通过循环来实现的，但是 map 可以快速并且并行地实现，因为每个单独的计算都不会相互影响，我们需要 Numba 工具。首次创建函数时，它会将原始 Python 代码转换为更快的数值操作。在开发数学运算符时，我们已经看到了这个库的一个示例：

```
def neg(x):
  return -x
```

如果我们想要改进我们的 map 实现，我们可以将其更改为如下所示：

```
from numba import njit
  def map(fn):
      # 改动 1: 把函数从Python搬到JIT版本
      fn = njit()(fn)
      def _map(out, input):
          for i in range(len(out)):
              out[i] = fn(input[i])
      # 改动 2: 内部的_map必须也是JIT版本
      return njit()(_map)
  # 注意，当外部map被调用时，所有JIT都会发生
  neg_map = map(neg)
```

调用上述函数时，它将运行利用该结构的快速 low-level 代码，而不是运行缓慢的 Python 代码。这种方法在启动时需要一点开销，但可以让事情变得更快。

此外，如果我们知道循环可以并行进行，我们可以通过一个小的改变进一步加快它：

```
from numba import njit, prange
def map(fn):
    fn = njit()(fn)
    def _map(out, input):
        # 改动3: 通过 prange 并行地运行循环
        for i in prange(len(out)):
            out[i] = fn(input[i])
    return njit(parallel=True)(_map)
```

这方面的优点在于，上面的代码基本上与 Python 代码相同，没有并行化。你可以在两者之间来回切换，而不需要太多改变。

你必须小心一点以确保循环实际上可以并行化。简而言之，这意味着步骤不能相互依赖，每次迭代不能写入相同的输出值。例如，在实现 reduce 时，必须小心混合并行和非并行循环。

我们的任务是通过 prange 来并行化，可以获得一些很大的成功。为了帮助去调试这个代码，我们创建了一个并行分析脚本，运行即可执行 Numba 诊断：

```
  python project/parallel_check.py
```

我们的任务中需要确保代码实现了 docstrings 中指定的优化。

7.1.2 完成 task3.1

这几个函数的实现方式有很多，我们不会花太长时间进行说明，因为我个人的思路和能力也很难优化到最好（miniTorch 的 guide 也说明了，一般人很难优化到赶上 Torch 的地步）。

map 函数

有人会倾向于先遍历并检查输入的 shape、stride 和输出的 shape、stride 是否相同，相同则采用简单的操作方式，不同则进行广播处理。

我们使用 tensor_data.py 文件里定义的 MAX_DIMS 这个常量（默认为 32，即最大的数据维度是 32 维，已经完全够用了）。

我们可以创建最大维度，也可以按照 out_shape 和 in_shape 来创建，都能通过测试：

```
def _map(out, out_shape, out_strides, in_storage, in_shape, in_strides
    ):
  # TODO: Implement for Task 3.1.
  for i in prange(len(out)):
      # 按形状创建维度
      out_index = np.zeros(len(out_shape), np.int32)
      in_index = np.zeros(len(in_shape), np.int32)
      # 或者直接创建最大维度
      # out_index = np.zeros(MAX_DIMS, np.int32)
      # in_index = np.zeros(MAX_DIMS, np.int32)
      to_index(i, out_shape, out_index)
      broadcast_index(out_index, out_shape, in_shape, in_index)
      data = in_storage[index_to_position(in_index, in_strides)]
      map_data = fn(data)
      out[index_to_position(out_index, out_strides)] = map_data
```

但是如果像在以前的函数中这么创建，就会报错：

```
  out_index = np.array(out_shape)
  in_index = np.array(in_shape)
# 报错：Rejected as the implementation raised a specific error:
# TypingError: array(int32, 1d, C) not allowed in a homogeneous
    sequence
```

需要在 tensor_data.py 文件里修改 to_index 函数：

```
# TODO: Implement for Task 2.1.
# 注意下面这一行，如果没有的话就会在最后一行的除法赋值中报错。
ordinal = ordinal + 0
for d in range(len(shape) - 1, -1, -1):
    out_index[d] = ordinal % shape[d]
    ordinal = ordinal // shape[d]
```

至于为什么会报错，可能是线程冲突的原因（与内存和拷贝有关），暂时不去关注该问题。

测试程序在 test_tensor_general.py 文件里实现：

```
  def test_one_args(fn, backend, data)
```

通过测试即可。

zip 函数

由于 zip 函数和 map 函数比较类似，所以就不再强调如何实现的了，直接给出源码：

```
# TODO: Implement for Task 3.1.
for i in prange(len(out)):
  a_index = np.zeros(MAX_DIMS, np.int32)
  b_index = np.zeros(MAX_DIMS, np.int32)
  o_index = np.zeros(MAX_DIMS, np.int32)
  to_index(i, out_shape, o_index)
  broadcast_index(o_index, out_shape, a_shape, a_index)
  broadcast_index(o_index, out_shape, b_shape, b_index)
  a_data = a_storage[index_to_position(a_index, a_strides)]
  b_data = b_storage[index_to_position(b_index, b_strides)]
  map_data = fn(a_data, b_data)
  out[index_to_position(o_index, out_strides)] = map_data
```

reduce 函数

reduce 函数的实现可以参照前面实现的内容，也不再详述：

```
# TODO: Implement for Task 3.1.
n = len(out)
out_dims = len(a_shape)
for i in prange(n):
    out_index = np.zeros(out_dims)
    to_index(i,out_shape,out_index)
    out_idx = index_to_position(out_index,out_strides)

    reduce_dim_size = a_shape[reduce_dim]

    for j in range(reduce_dim_size):
        idx_a = out_index.copy()
        idx_a[reduce_dim] = j
        pos_a = index_to_position(idx_a, a_strides)
        out[out_idx] = fn(out[out_idx],a_storage[pos_a])
```

通过测试函数即可说明正确完成了该函数：

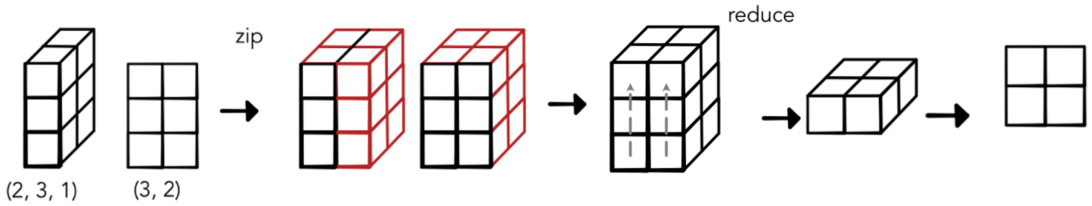
```
  def test_reduce(fn, backend, data)
```

7.2 矩阵相乘

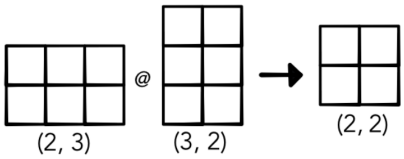
7.2.1 Fusing 操作与矩阵

使用矩阵相乘会比用张量广播进行计算的效率更高，矩阵相乘本质上是一种 Fusing(融合) 操作，就是把多个步骤融合起来，关于如何自动进行算子的融合有很多正在进行的工作，但是对于非常常见的运算符，直接编写这些运算加速它们是值得的。

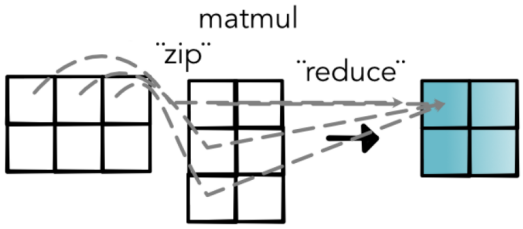
我们前面是采样广播 zip 和 reduce 来进行的矩阵相乘，考虑一个 tensor 的 shape 为 (2,3,1) 和一个 tensor 的 shape 为 (3,2)，首先我们先 zip 它们，然后 reduce，最后就能得到 shape 为 (2,2) 的 tensor：



我们可以通过编写一个特殊的张量函数（@ 运算符）来实现这一点，它由一个直接产生输出的正向函数和一个产生所需梯度的反向函数构成：

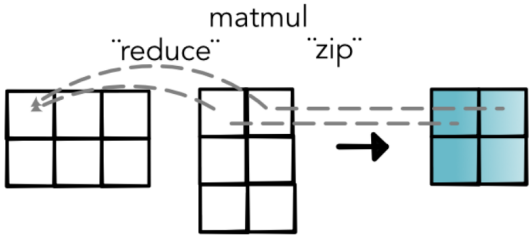


这允许我们能够专门实现矩阵相乘。我们的步骤：(1) 浏览 out 的 indices，(2) 查看哪些 indices 被 reduce 了，(3) 查看哪些是 zip 的一部分。



考虑到矩阵乘法操作的重要性，值得花时间思考如何加快每个步骤。一旦我们知道了输出 indices，我们就可以非常有效地遍历输入 indices 并累加它们的总和。

一个快速矩阵乘法运算可以用来计算前向 forward()。要反向计算，我们需要反转我们计算的操作。特别是，这需要使用可选的输入矩阵把 grad out 进行 zip。我们可以通过追踪 forward() 箭头看到 backward()：



优化这段代码也有点烦人，但同样幸运的是，为了计算向后的步骤，我们可以重用向前的优

化。事实上，我们可以简单地使用矩阵微积分中的下列恒等式，它告诉我们，使用 backward 可以计算为转置和另一个矩阵乘法。

$$f(M,N) = MN \tag{7.2.1}$$

$$g'_M(f(M,N)) = grad_{out}N^T \tag{7.2.2}$$

$$g'_N(f(M,N)) = M^T grad_{out} \tag{7.2.3}$$

矩阵微积分并不是一个很容易的知识（如果您在课程中上过国内的教材就能深刻地感受到这一点，我曾经不止一次晕倒在学习 Jacobian 矩阵的路上，国内的资料更是少得可怜），在 [16] 中可以找到非常好的解释和描述，以后有空我也会把这个教程再重新学习和整理一遍（之前根据张贤达教授的《矩阵分析与应用》编写的一本小册子其实并没有做到很通俗易懂），但 [16] 仅涉及向量函数对向量求导，所以也无法实现我们现在的目的。这里我们先根据上面的公式就可以实现我们的代码目标。

7.2.2 完成 task3.2

Backend.MatMul 类已经实现了矩阵乘法前向计算和反向传播，我们现在只需要注意如何实现矩阵乘法的逻辑。

如果我们想自己构建测试，我们必须得如此构建测试代码，让构造运算的算法为快速的张量算法，否则并不会调用我们新实现的加速代码（以前的代码并没有实现矩阵算法）。

```
# TensorBackend = minitorch.make_tensor_backend(minitorch.TensorOps)
FastTensorBackend = minitorch.make_tensor_backend(minitorch.FastOps)
a = tensor([[1,2],[1,2]], backend = FastTensorBackend)
b = tensor([[3,3],[4,4]], backend = FastTensorBackend)
c = a@b
print(c)
```

如何调用矩阵相乘

在开始写我们的程序之前，我们得先明白 matrix_multiply 函数做了些什么。

首先会进行判断，确保两个要被相乘的矩阵都是 3 维以上的，如果只是两个二维矩阵相乘，就扩展为 3 维：

```
    both_2d = 0
    if len(a.shape) == 2:
        a = a.contiguous().view(1, a.shape[0], a.shape[1])
        both_2d += 1
    if len(b.shape) == 2:
        b = b.contiguous().view(1, b.shape[0], b.shape[1])
        both_2d += 1
```

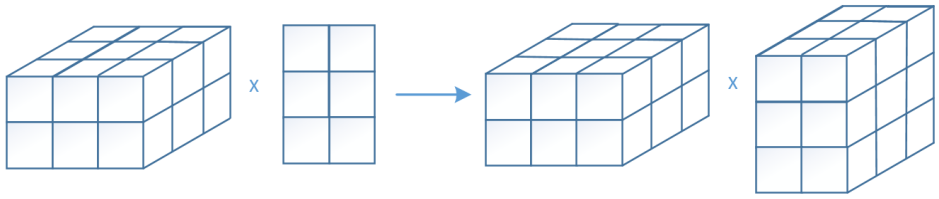
如果 a 和 b 都是二维的，则进行记录：

```
both_2d = both_2d == 2
```

然后我们要计算一下相乘以后的矩阵应该是什么样子：

```
ls = list(shape_broadcast(a.shape[:-2], b.shape[:-2]))
ls.append(a.shape[-2])
ls.append(b.shape[-1])
assert a.shape[-1] == b.shape[-2]
out = a.zeros(tuple(ls))
```

对于一个二维矩阵来说，相乘以后，得到的 shape 是由 b 的列数（b.shape[-1]）和 a 的行数（a.shape[-2]）构成，也就是 (a.shape[-2],b.shape[-1])，其他维度的维度数只是需要被广播的部分，可能是不同的 batch_size。



矩阵相乘完以后，如果进行了维度扩增，就恢复为 2 维矩阵：

```
if both_2d:
  out = out.view(out.shape[1], out.shape[2])
```

注意只有两个矩阵都是 2 维时，才需要恢复。只要有一个矩阵维度大于 2，则最终输出的结果也一定是维度大于 2 的。

矩阵相乘

源码给出的下面两个变量我没有用到，也不知道怎么用，所以就无所谓了：

```
a_batch_stride = a_strides[0] if a_shape[0] > 1 else 0
b_batch_stride = b_strides[0] if b_shape[0] > 1 else 0
```

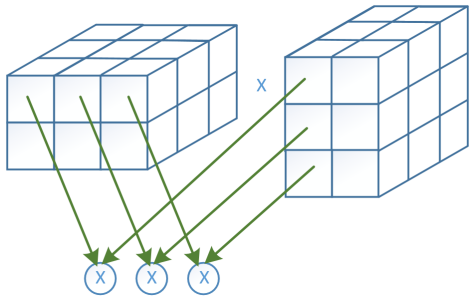
我们让加速程序遍历整个输出的 storage：

```
n = len(out)
for i in prange(n):
  # 把i计算到输出的索引
  out_index = out_shape.copy()
  temp_i = i + 0
  to_index(temp_i,out_shape,out_index)
  # 计算它在storage上的位置。对于连续张量，i等于out_pos
  out_pos = index_to_position(out_index, out_strides)
```

然后在这个大循环的内部，要进行矩阵相乘的小循环。就是把前面矩阵的行元素乘上后面矩阵的列元素，然后再相加。我们需要遍历的数量是前一个矩阵的列数：

```
  last_dim = a_shape[-1]
  for j in range(last_dim):
      temp_j = j + 0
```

也就是前一个矩阵的第 temp_j 列的元素乘以后一个矩阵的第 temp_j 行的元素：



通过广播机制就能找到对应元素，然后相乘即可：

```
    a_index = a_shape.copy()
    a_tmp_index = out_index.copy()
    a_tmp_index[-1] = temp_j
    broadcast_index(a_tmp_index, out_shape, a_shape, a_index)
    a_pos = index_to_position(a_index, a_strides)

    b_index = b_shape.copy()
    b_tmp_index = out_index.copy()
    b_tmp_index[-2] = temp_j
    broadcast_index(b_tmp_index, out_shape, b_shape, b_index)
    b_pos = index_to_position(b_index, b_strides)

    out[out_pos] += (a_storage[a_pos] * b_storage[b_pos])
```

矩阵相乘的反向传播

该函数实现在 tensor_functions.py 文件的 class MatMul 里。

矩阵相乘的求导根据前面描述的公式即可实现，就是取转置，然后再乘以 grad_output。注意取转置时就是交换最后和倒数第二个维度：

```
def transpose(a):
  order = list(range(a.dims))
  order[-2], order[-1] = order[-1], order[-2]
  return a._new(a._tensor.permute(*order))
```

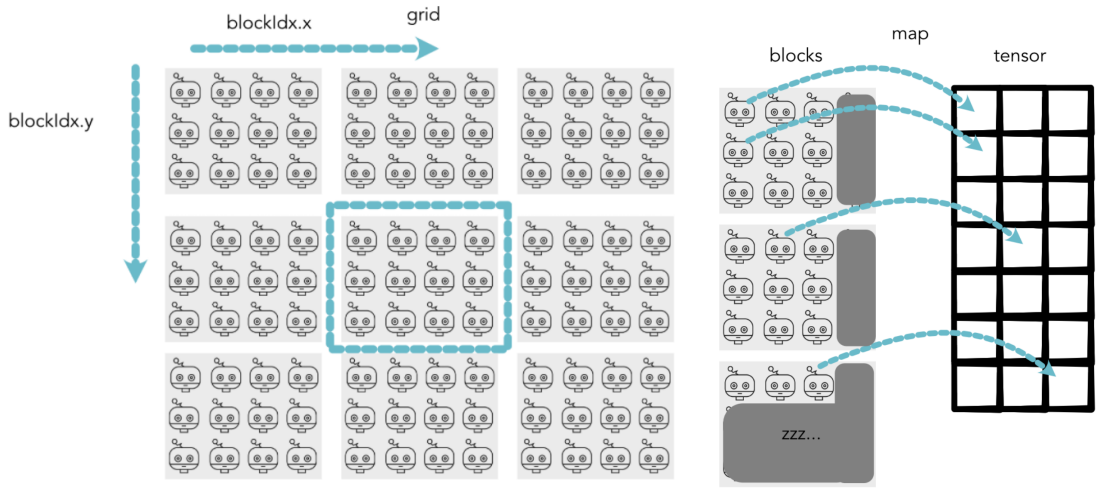
7.3 CUDA 运算

如果我们有专门的硬件，例如 GPU，我们甚至可以做得比并行更好。此任务要求您构建后端操作的 GPU 实现。当然，我们很难达到 PyTorch 的水平，但如果你很聪明，是可以让这些计算非常快的（目标是任务 3.1 的 2 倍）。

7.3.1 CUDA 编程

GPU 计算对深度学习的重要性巨大：它使运行许多即使在几年前也难以解决的模型成为可能。编写 GPU 代码比 CPU 并行化代码需要更多的工作，GPU 的编程模型与 CPU 稍有不同，这需要一些时间才能完全理解。不过幸运的是，有一个很好的 Numba 库扩展，允许我们直接用 Python 为 GPU 编码。最常用的 GPU 编程模型是 CUDA，是 NVIDIA 设备的 C++ 专有扩展。相比于之前学习 C++ 上的 CUDA 编程，Python 编程的难度已经很低了。

在 CUDA 中，线程 (threads) 聚集到 block 中，block 聚集到一起，成为 grid。每一个 block 都有明确的尺寸和 shape，也都有它们自己的共享内存：



计算全局位置的方法：

```
  x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
  y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
```

为 CUDA 编写代码时，必须同时使用相同的代码对所有线程进行编码。每个线程在 lockstep 中运行相同的函数，我们把线程程序都写作一个函数：

```
# Helper function to call in CUDA
@cuda.jit(device=True)
def times(a, b):
  return a * b
# Main cuda launcher
@cuda.jit()
def my_func(in, out):
  # Create some local memory
```

```
local = cuda.local.array(5)
# Find my position.
x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
# Compute some information
local[1] = 10
# Guard some of the threads.
if x < out.shape[0] and y < out.shape[1]:
  # Compute some global value
  out[x, y] = times(in[x, y], local[1])
```

请注意，我们不能直接调用上述函数：我们需要启动它，并说明如何设置 block 和 grid。以下是使用 Numba 的方法：

```
threadsperblock = (4, 3)
blockspergrid = (1, 3)
my_func[blockspergrid, threadsperblock](in, out)
```

这将建立一个类似于前面提到的 map 函数的 block 和 grid 结构。my_func 中的代码对结构中的所有线程同时运行。但是，您必须小心一点，因为有些线程可能会计算超出结构内存的值，所以上述代码最后一部分先判断 x 和 y 是否在结构内存内，然后再执行计算。

7.3.2 完成 task3.3

这些函数的实现和前面并没有太大不同，只是需要调用：

```
out_index = cuda.local.array(MAX_DIMS, numba.int32)
in_index = cuda.local.array(MAX_DIMS, numba.int32)
```

来生成数组，并且位置计算是根据线程计算的：

```
pos = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
```

据 guide 所述，reduce 是一个特别具有挑战性的功能。miniTorch 中提供了一个简单的练习（求和函数）来帮助我们理解。我们这里直接给出 _reduce 函数：

```
# TODO: Implement for Task 3.3.
out_dims = len(out_shape)
a_dims = len(a_shape)
out_index = cuda.local.array(MAX_DIMS, numba.int32)
a_index = cuda.local.array(MAX_DIMS, numba.int32)
pos = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
if out_size > pos:
    to_index(pos, out_shape, out_index)
    reduce_dim_size = a_shape[reduce_dim]
    # 注意初始化out
```

```
        out[pos] = reduce_value
        for j in range(reduce_dim_size):
            to_index(pos, out_shape, a_index)
            a_index[reduce_dim] = j
            pos_a = index_to_position(a_index, a_strides)
            out[pos] = fn(out[pos], a_storage[pos_a])
```

通过全部测试即可。

7.4 CUDA 矩阵相乘

最后，我们可以将这两种方法结合起来，实现 CUDA matmul。这个操作可能是所有深度学习中最重要的，也是快速构建模型的核心。同样，我们首先要努力做到准确，但是，你做得越快越好。

对于许多深度学习任务来说，高效地实现矩阵乘法和约简是非常重要的。按照指南来实现这些功能。我们需要通过将大型矩阵运算与简单运算进行比较，来证明这些运算会加快大型矩阵运算的速度。

7.4.1 完成 task3.4

矩阵相乘的难度较大，我也是花了很长时间去调试。一开始，我只能通过 test_mul_practice1，然后我修改了索引等表示方式以后，可以通过 practice2 和 practice3 了。之后还需要扩展到 batch 上，也就是 practice5 和 practice6。在构建矩阵相乘的过程中，也对这些概念越来越熟悉了。当我成功通过了全部测试的时候，激动难以言表。

本小节我将介绍如何分析和实现这个过程。

构建测试

虽然有测试程序，但有时候我们想自己构建一个更简单清楚的测试，那么我们的测试方式如下：

```
import random
dim_c = 32
x = [[random.random() for i in range(dim_c)] for j in range(dim_c)]
y = [[random.random() for i in range(dim_c)] for j in range(dim_c)]
# 在cuda方法里相乘
CudaTensorBackend = minitorch.make_tensor_backend(minitorch.CudaOps)
a = tensor(x, backend = CudaTensorBackend)
b = tensor(y, backend = CudaTensorBackend)
c = minitorch.mm_practice(a, b)
# 使用前面测试通过的矩阵快速相乘程序，用来对照
```

```
z = minitorch.tensor(x, backend=FastTensorBackend) @ minitorch.tensor(
    y, backend=FastTensorBackend)
# 遍历，对比结果是否一致
for i in range(dim_c):
    for j in range(dim_c):
        if minitorch.operators.is_close(z[i, j], c._storage[dim_c * i
            + j]):
            pass
        else:
            print("("+str(i) + "," + str(j)+")␣"+ str(c._storage[dim_c
                * i + j]) + ",␣"+ str(z[i, j]))
```

如果有计算不一致的值就会打印出来，这样容易让我们查看和分析。

matrix_multiply 代码分析

该函数前面部分和 fast_ops 的一样，都是生成一个输出数组 out。

区别在于设定的多线程：

```
  blockspergrid = (
    (out.shape[1] + (THREADS_PER_BLOCK - 1)) // THREADS_PER_BLOCK,
    (out.shape[2] + (THREADS_PER_BLOCK - 1)) // THREADS_PER_BLOCK,
    out.shape[0],
  )
  threadsperblock = (THREADS_PER_BLOCK, THREADS_PER_BLOCK, 1)
```

我们要注意的是这里的 grid 是三维的，所以待会计算索引的时候也要按三维的去计算（否则 cuda 计算时，后面的 batch[0] 后面的 batch 我们都索引不到）。

但我们从这里可以看到，矩阵相乘的程序会被默认为都是三维的，shape[0] 表示 batch 数，shape[1] 是矩阵的行数，shape[2] 是矩阵的列数。

由于这是 miniTorch 教程的源代码，所以就不给它人为修改了，但其实我认为应该这么写（用来做更高维度的矩阵相乘）：

```
  blockspergrid = (
    (out.shape[-2] + (THREADS_PER_BLOCK - 1)) // THREADS_PER_BLOCK,
    (out.shape[-1] + (THREADS_PER_BLOCK - 1)) // THREADS_PER_BLOCK,
    (len(out._tensor._storage) + THREADS_PER_BLOCK * THREADS_PER_BLOCK
        - 1) // (THREADS_PER_BLOCK * THREADS_PER_BLOCK) ,
  )
  threadsperblock = (THREADS_PER_BLOCK, THREADS_PER_BLOCK, 1)
```

这里之所以要这么写：

```
  (out.shape[-2] + (THREADS_PER_BLOCK - 1)) // THREADS_PER_BLOCK
```

是因为，比如我们默认 THREADS_PER_BLOCK 是 32，那么如果我们行线程维度数是 32，那么只需要一个 block，如果是 33，就得需要两个 block 了。

tensor_matrix_multiply 代码实现

在上一节的描述下，我们给出最终代码：

```
# TODO: Implement for Task 3.4.
pos_1 = cuda.blockDim.x * cuda.blockIdx.x + cuda.threadIdx.x
pos_2 = cuda.blockDim.y * cuda.blockIdx.y + cuda.threadIdx.y
# 每行有多少个线程数
index_per_grid_x = cuda.blockDim.x * cuda.gridDim.x
# 每个gridDim.z方向有多少个线程数（行线程数乘列线程数）
index_per_grid = cuda.blockDim.x * cuda.gridDim.x * cuda.blockDim.y
    * cuda.gridDim.y
pos = index_per_grid * cuda.blockIdx.z + index_per_grid_x * pos_2 +
    pos_1
if out_size > pos:
  out_index = cuda.local.array(MAX_DIMS, numba.int32)
  a_index = cuda.local.array(MAX_DIMS, numba.int32)
  b_index = cuda.local.array(MAX_DIMS, numba.int32)
  to_index(pos, out_shape, out_index)
  broadcast_index(out_index, out_shape, a_shape, a_index)
  broadcast_index(out_index, out_shape, b_shape, b_index)
  for j in range(a_shape[-1]):
    a_index[len(a_shape) - 1] = j
    pos_a = index_to_position(a_index, a_strides)

    b_index[len(b_shape) - 2] = j
    pos_b = index_to_position(b_index, b_strides)

    out[pos] += (a_storage[pos_a] * b_storage[pos_b])
```

如果这里的线程索引的描述您看不懂的话，可以去网络上找找相关资料来辅助理解。因为篇幅原因，不会花太多时间介绍。

7.5 训练

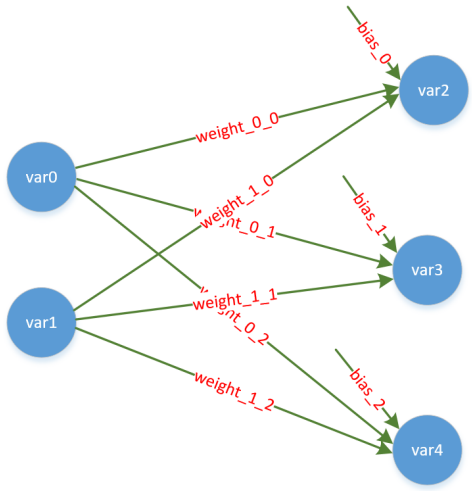
当写到这一节时，我就感受到胜利在望了。从刚接触到这个项目，去研究怎么搭建环境、怎么调试和分析，花了不少的精力和时间，度过了很多加班熬夜的夜晚。但所有付出都是值得的，当我可以熟练地构建计算图、熟练地做矩阵相乘的测试时，那种成就感也是油然而生的。虽然 MiniTorch

并不是一个非常复杂的工业级自动微分机，但它的基本思想和方法跟工业级神经网络框架是一致的。

如果代码正常工作，现在应该可以在 project/run_fast_tensor.py 中继续使用 tensor 训练网络。此代码与张量的基本训练设置相同，但现在使用快速张量代码。

7.5.1 完成 task3-5

对于线性单层结构来说，前向传播示意图如下：



对应的程序也很简单：

```
class Linear(minitorch.Module):
  def forward(self, x):
      # TODO: Implement for Task 3.5.
      return x @ self.weights.value + self.bias.value
```

网络也大致不变：

```
class Network(minitorch.Module):
  def forward(self, x):
      # TODO: Implement for Task 3.5.
      middle = self.layer1.forward(x).relu()
      end = self.layer2.forward(middle).relu()
      return self.layer3.forward(end).sigmoid()
```

7.5.2 训练的代码解释

在每个 epoch 中，都会把样本顺序进行打乱：

```
  c = list(zip(data.X, data.y))
  random.shuffle(c)
  X_shuf, y_shuf = zip(*c)
```

首先先把样本特征和类别组成对，然后打乱，再把特征和类别分别取出。shuffle 函数就是把数据打乱。

单个数据的 shape 是 (1,2)，一个 batch 的数据的 shape 就是 (batch,2) 构成一个张量，然后送入到网络中。

单个数据的输出是一个标量，所以一个 batch 的数据输出以后，经过 view(y.shape[0]) 就变为了和 y 一个形状，之后计算损失值，并执行反向传播。

8. 网络 Networks

我们现在拥有深度学习 *lib* 库，它拥有像 *pyTorch* 这样一个真正的工业系统的大部分特征。为了充分利用这项工作，本模块完全基于我们实现的软件框架。我们将建立一个图像识别系统，通过在 *MNIST* 上构建 *LeNet* 网络来实现这一点：用于数字识别的经典卷积神经网络（*CNN*），以及用于 *NLP* 情绪分类的 *1D conv*。本来这一章我并不打算写，因为它并不再属于基本的网络框架部分了，而是一些具体功能（类似于 *LSTM* 等自定义结构），但考虑到课程完整性，还是把它写完了。其实我们现在具备的知识已经可以去学习 *PyTorch* 源码了，而没有必要再费尽心力去自己造一些效率并不会很高的轮子了。

8.1 准备工作

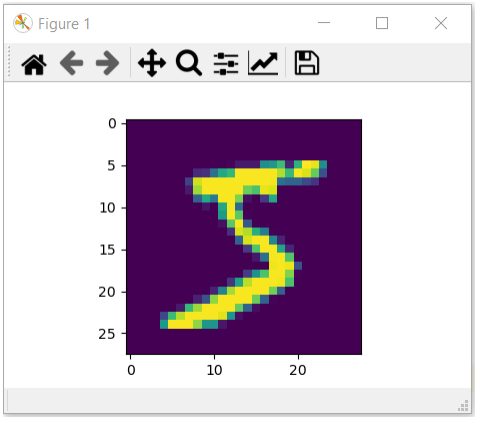
我们需要下载并安装 MNIST 库：

```
pip install python-mnist
cd project/ ; mnist_get_data.sh
```

它将在您的模块中添加一个 data/directory。您可以尝试以下代码来测试，我把数据放在了统一的目录里（SciData）方便管理。显示其中的第一张图像的代码：

```
from mnist import MNIST
import numpy as np
import matplotlib.pyplot as plt
mndata = MNIST("C:/SciData/DeepleranData/MNIST/raw")
images, labels = mndata.load_training()
im = np.array(images[0])
im = im.reshape(28, 28)
plt.imshow(im)
plt.show()
```

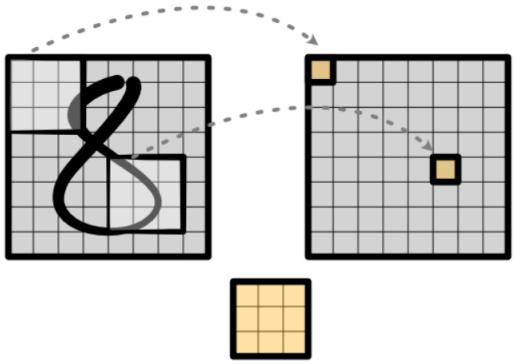
显示如下：



8.2 一维卷积

到目前为止，我们解决分类问题的主要方法是首先将输入送入到线性层，该线性层对输入应用许多不同的线性分隔符 (linear seperators)，然后应用 ReLU 函数将其转换为新的隐藏表示 (hidden representation)。

上述方法的一个主要问题是，它基于原始输入特征的绝对位置，这使得我们无法在输入的不同部分使用相同的学习参数。现在，我们可以实现一个滑动窗口（即卷积），在输入的不同局部区域使用相同的学习参数集，如下所示：



这不是直接变换整个输入图像，而是在图像的每个部分应用相同的卷积来产生新的表示。从某种意义上说，它是滑动的，实际上，窗口在哪个区域上进行处理，滑动整个图像以产生输出。

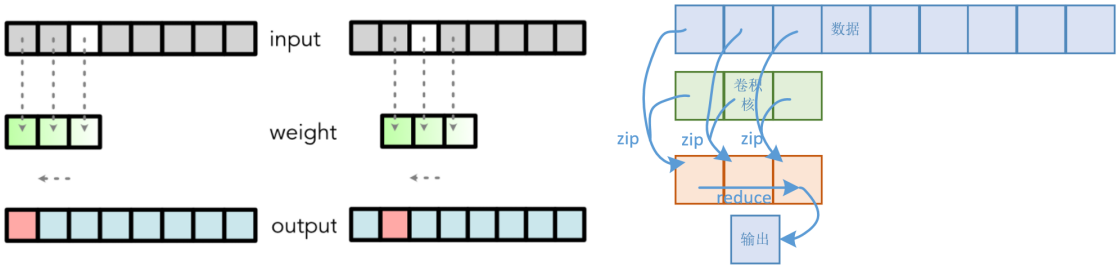
我们将主要使用图像上的卷积来了解如何学习这些局部应用的参数，但为了直观地了解卷积是如何工作的，我们将从 1D 中的输入序列开始。

下面我会根据 guide 来讲解一维卷积。我不知道是我个人的理解问题还是我个人先验知识阻碍了我的认知，guide 的内容一开始读起来让我感觉稀里糊涂的，我也是花了很长时间才理解图示，这里我尽量讲解地更清晰明白一些。

8.2.1 一维卷积原理

我们的简单的 1D 卷积采用长度为 T 的输入向量和长度为 K 的权重（或内核 (kernel)）向量来生成长度为 T 的输出向量。它通过沿输入滑动权重、使用部分输入来 zip 权重、将 zip 的结果

reduce 到一个值，然后将该值保存在输出中（注意 guide 给的示意图有点迷惑性，关于它从哪个元素开始卷积、以及边缘到底算在哪里，需要看下面的程序）：



如果滑动窗口越过输入的边缘，为了使事情更简单，我们只假设边缘外的输入值为 0。

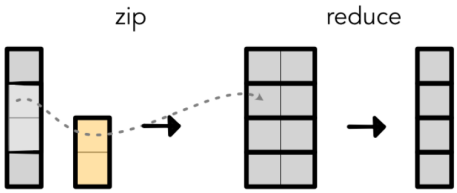
考虑卷积的另一种方法是展开输入。现在假设我们有一个名为 unroll 的函数，它可以获取一个输入张量，并生成一个新的输出张量（可以看到，边缘被添加到后面去了，第 6 个输出的末尾添加了两个 0）：

```
input = minitorch.tensor([1, 2, 3, 4, 5, 6])
K = 3
input = unroll(input, K)
print(input)

[[1, 2, 3], [2, 3, 4], [3, 4, 5],
 [4, 5, 6], [5, 6, 0], [6, 0, 0]]
```

然后我们应用矩阵乘法得到点积：

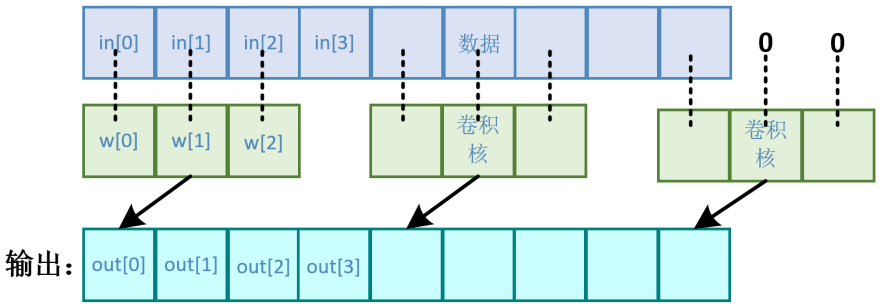
```
weight = minitorch.tensor([5, 2, 3])
output = (input @ weight.view(K, 1)).view(T)
print(output)

[18, ..., 30]
```

我们可以将卷积视为以下两个独立操作的融合（请注意，我们在实践中并不是这样实现的！）：



给定一个输入和权重，它会有效地展开输入，并将矩阵与权重相乘。这项技术非常有用，因为它允许您将鼠标悬停在输入句子的局部部分上。可以将权重视为在原始输入中捕获一个模式（或许多不同的模式）。

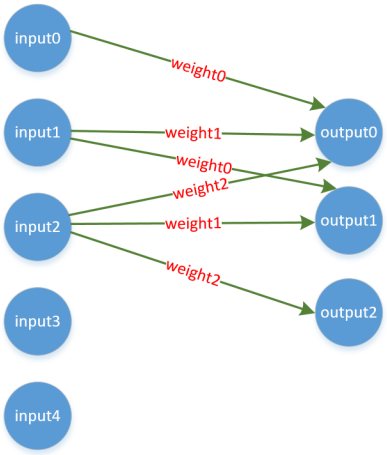
我们再给出一个更详细的示意图：



和模型中的其他操作一样，我们需要能够计算一维卷积的 backward 操作。注意，我们可以通过矩阵乘法通过每个 cell 的流动进行推理。回到上一个示例，原始输入中的第三个单元格（即 input[2]）仅用于计算输出中的三个不同单元格：output[0]、output[1] 和 output[2]。

```
output[0] = weight[0] * input[0] + \
  weight[1] * input[1]  + weight[2] * input[2]
output[1] = weight[0] * input[1] + \
  weight[1] * input[2]  + weight[2] * input[3]
output[2] = weight[0] * input[2] + \
  weight[1] * input[3]  + weight[2] * input[4]
```

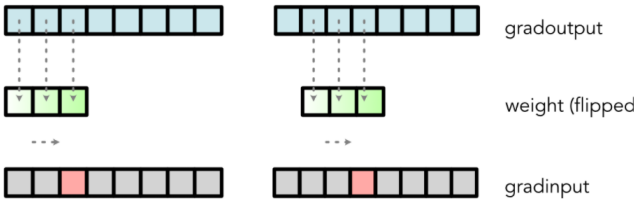
注意这里为了简单，都是默认卷积核权重是固定的。我们用一个图来表示一下：



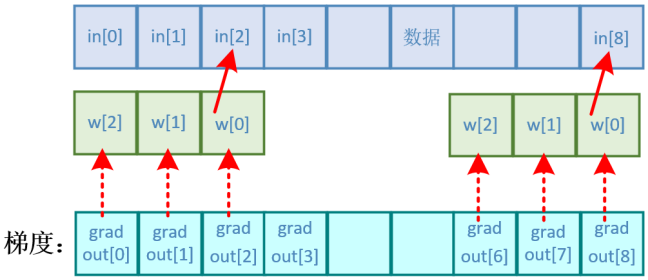
因此，梯度计算为：

```
grad_input[2] = weight[0] * grad_output[2] + weight[1] * grad_output
    [1]  + weight[2] * grad_output[0]
```

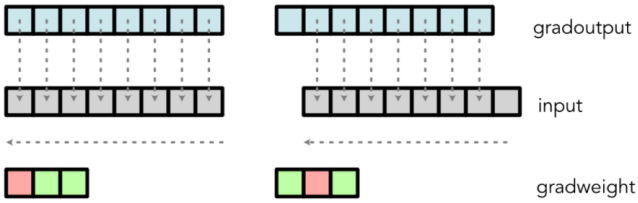
从视觉上看，这意味着卷积的 backward 是一个固定在相反一侧的卷积，权重相反：



我再来画个示意图：



可以通过相似的方法为权重计算梯度：

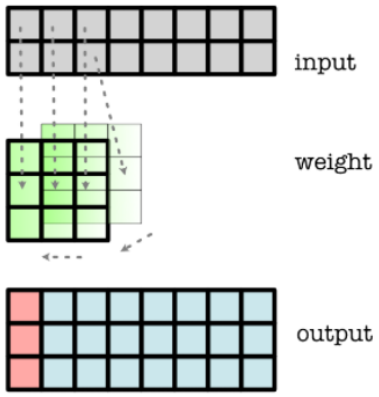


我们可以看到，weight[0] 对所有的输入（input[0] 到 input[n-1]）都会相乘，而 weight[1] 并不会与 input[0] 相乘。即：

```
gradweight[0] = gradoutput[0]*input[0] + gradoutput[1]*input[1]
    +...+ gradoutput[8]*input[8]
```

上面的图示意味着，和矩阵相乘类似，实现一个快速的卷积可以用来执行 forward 和 backward 函数。

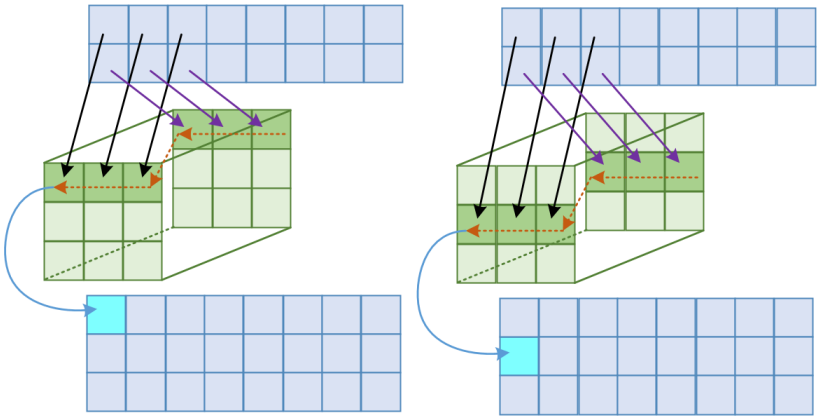
上述方法可以扩展为通过通道 (channals) 同时处理多个输入特征和多个权重。类似于矩阵乘法，我们取 (in_channels, T) 输入和 (out_channels, in_channels, K) 权重来产生一个 (out_channels, T) 输出。下面的这个例子中，in_channels=2，out_channels=3，K=3 并且 T=8。



代码方面（阅读起来可能有点困难，guide 中建议只去理解上面的图表（其实我觉得这个图更难理解，看起来乱七八糟的））：

```
input = minitorch.rand(in_channels, T)
input = unroll(input, K).permute(0,2,1) #Shape:in_channels x K x T
weight = minitorch.rand(out_channels, in_channels, K)
output = weight.view(out_channels, in_channels * K) @ input.view(
    in_channels * K, T)
```

为了更好地说明，我重新画一张图：



一维卷积有各种各样的应用：它可以应用于 NLP，作为一种将模型应用于多个相邻单词的方法；它可以用于语音识别，作为识别重要声音的一种方法；它可以用于异常检测，以发现触发警报的模式；你可以想象，应用于序列子集的微型神经网络在任何地方都是有用的。

8.2.2 完成 task4.1

我也没有花太多时间去研究怎么优化和加快计算，只是简单地实现和分析一下，由于不算是网络最基本的结构，所以这里暂时不再解释，以后我可能会根据 pyTorch 或者 tensorflow 源码来解释一下工业级神经网络框架的实现机制。

8.3 二维卷积

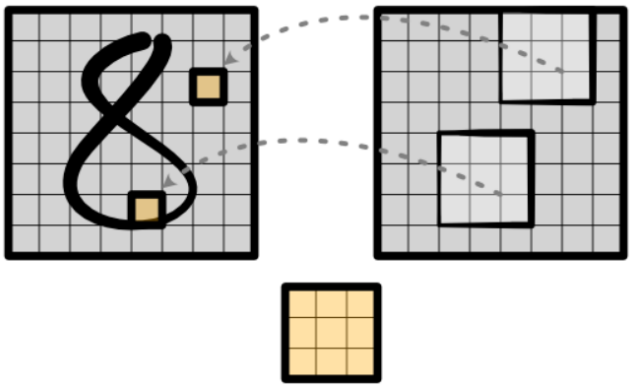
8.3.1 二维卷积讲解

我们简单的二维卷积取一个 (H,W) 的输出（高 height 和宽 width），以及一个 (KH,KW) 的权重（注意这个 KH 和 KW 并不是 K*H 和 K*W，这只是一个表示符号，一般 KH=KW=3），来产生一个 (H,W) 的输出。操作几乎相同：我们遍历输入矩阵中每个可能展开的矩形，然后乘以权重。假设我们有一个类似的矩阵展开函数，这相当于计算：

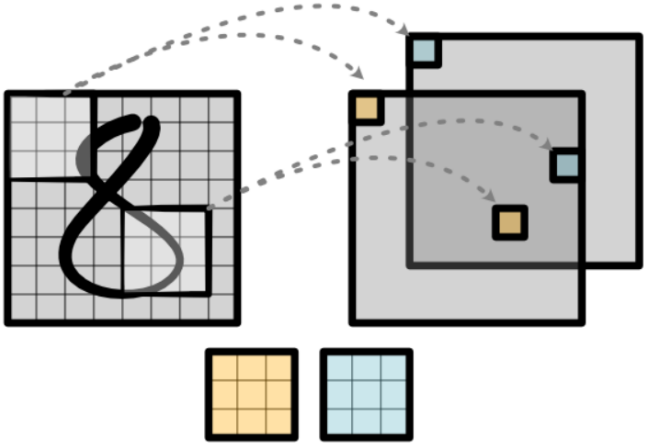
```
output = (unrolled_input.view(H, W, KH * KW) @ weight.view(KH * KW,
    1)).view(H, W)
```

另一种思考方法是将权重作为线性层应用于输入图像中的每个矩形。

关键的是，正如一维卷积定位在左侧一样，二维卷积定位在左上角。为了 backward() 计算，我们计算右下角的反向卷积：



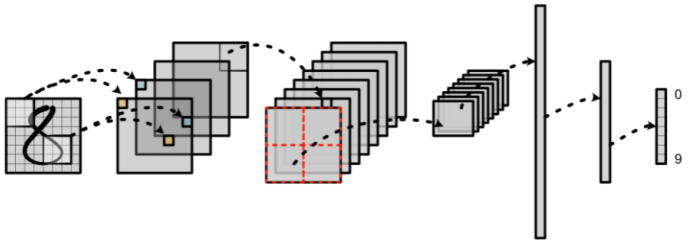
最后，我们可以通过同时对多个输入特征应用多个权重来再次使事情复杂化，这为我们提供了 Torch 中使用的标准 2D 卷积。我们采用一个 (in_channels,H,W) 输入和一个 (out_channels,in_channels,KW,KH) 权重矩阵来产生一个 (out_channels,H,W) 输出：



输出大致是这个形式：

```
output = unrolled_input.view(H, W, in_channels * KH * KW) \
@ weight.view(in_channels * KH * KW, out_channels)
```

二维卷积是图像识别系统的主要运算符。它允许我们将图像处理为局部特征表示。这也是卷积神经网络管道中的关键步骤。它将输入图像转换为隐藏特征 (hidden features)，这些特征通过网络的每个阶段传播：



8.3.2 完成任务 4.2

该部分内容同一维卷积，暂不叙述，有需要可以参考源码的实现。

8.4 池化

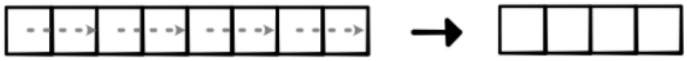
8.4.1 池化的基本概念

在前面的模块中，我们发现在某些维度上减少张量的形状是有用的。例如，在 NLP 示例中，我们对句子长度进行求和，以便根据求和的单词表示进行分类。关键是，这个操作并没有消除单词的重要性（即它们仍然接收梯度信息），但它确实允许我们基于固定大小的表示做出单个分类决策。

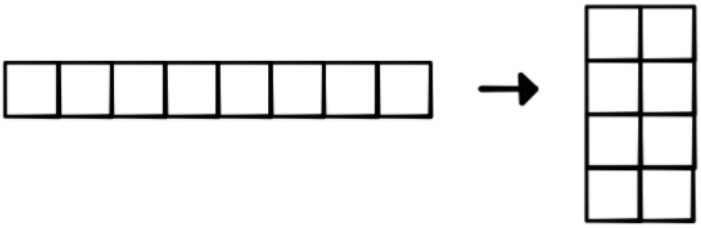
我们现在要实现的这种类型的网络在神经网络文献中被非正式地称为池化。当我们减少部分输入以使用更小的大小时，我们说我们已经将输入表示集合在一起。

在长度或其他维度上应用缩减是合并的一种形式。对于连续的情况，我们可以称之为 X-over-time pooling（其中 X 可能是总和、平均值、最大值等）：

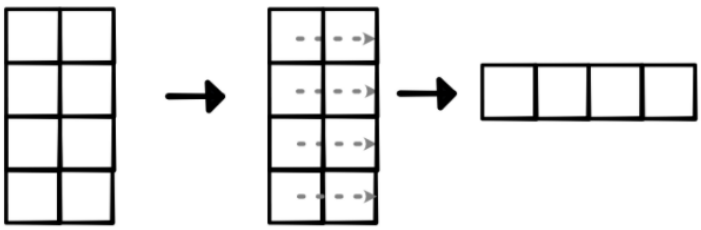
池化的另一种常见形式是仅在维度内局部池化。例如，我们可以将相邻元素集中在一起，以减少该维度的 l 长度，如下所示。这在输入序列很长的语音识别等领域很常见：



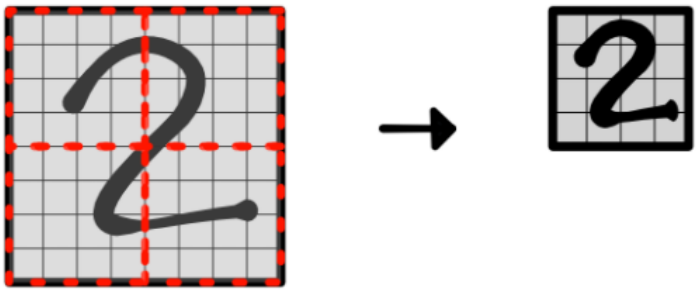
在实践中，为了避免将池操作实现为循环，我们可以操纵输入张量的 shape 和 strides，以便直接池化它。假设输入张量是连续的，我们可以通过应用视图简单地添加一个额外的维度。例如，我们可以将 (8,) 输入视为 (4,2)：



一旦我们有了这种形式的输入，我们就可以在第二维度上进行压缩，得到一个 (4,1) 张量，可以看作是 (4,)。只要池化的维度可以被池化常数（即每个池化操作的输入大小）整除，这个过程就会产生正确的池化结果，如下所示。如果不能整除，我们可以填充输入张量，或者沿着池化方向添加填充。



您将在二维图像中实现这种类型的池化。您需要在 1D 池中概括上述想法，创建一个具有两个额外维度的 shape 来 reduce：



池化的好处是，对池化的结果应用较小的卷积可以覆盖原始输入图像中的较大区域。理想情况下，这些后续层可以学习输入图像的更高级属性。

8.4.2 完成任务 4.3

池化主要会去操作最后两个维度，把这两个维度的内容进行排列，然后进行 reduce。我们来介绍一下过程，根据测试示例（见 test_nn.py 文件的 test_avg 函数），我们构造的是一个 (1,1,4,4) 的张量，但是由于前两个维度都没有被变换，所以我们只关注后两个维度。我们把 tile 和 avgpool2d 两个函数的过程合起来看。设随机初始化的张量为：

```
[[0.71  0.12  0.55  0.63]
 [0.45  0.09  0.90  0.12]
 [0.71  0.95  0.05  0.62]
 [0.15  0.51  0.78  0.52]]
```

在调用下面的函数以后，

```
input = input.view(batch, channel, tile_h, kh, tile_w, kw)
```

变成了这种排列 (4,1,2,2)：

```
[[[
    [0.71  0.12]
    [0.55  0.63]]]
  [[
    [0.45  0.09]
    [0.90  0.12]]]
  [[
    [0.71  0.95]
    [0.05  0.62]]]
  [[
    [0.15  0.51]
    [0.78  0.52]]]
```

我们把需要被 reduce 的局部区域放在最后两个维度上，使得 shape 变为 (4,2,1,2)。

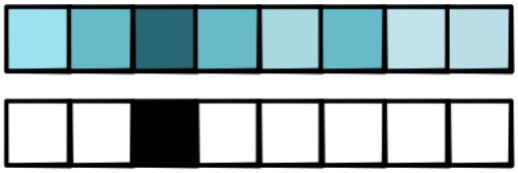
我们的最终目标是变为 (4,2,2)，其中最后一个维度的大小是 2，这个维度是我们要去 reduce 的维度。如果池化局部区域大小为 (2,2)，则最后一维的大小就是 4。

8.5 Softmax and Dropout

本节要实现 max 函数、softmax、log softmax 以及 drop 和最大池化运算。

8.5.1 基本知识

sigmoid 只能用于二分类问题，当我们有 K 个输出类别时，我们需要多分类。我们假设输出是一个 K 维向量，我们想要这个值最高的作为结果，argmax 函数给出一个 K 维向量，目标位置为 1，其他位置都是 0。



softmax 函数表示如下：

$$softmax(\mathbf{x}) = \frac{\exp \mathbf{x}}{\sum_i \exp x_i} \tag{8.5.1}$$

由于 softmax 函数需要对输入分数求幂，因此在实践中，它在数值上可能不稳定。因此，通常使用数值技巧来计算 softmax 函数的对数：

$$logsoftmax(\mathbf{x}) = \mathbf{x} - \log \sum_i \exp x_i = \mathbf{x} - \log\Big(\sum_i \exp(x_i - m)\Big) - m \tag{8.5.2}$$

说到 max，我们可以在代码库中添加一个 max 操作符。我们可以计算向量的最大值（或通常的张量）作为一个约化，它返回输入中单个得分最高的元素。直观地说，我们可以考虑输入的微小变化对返回值的影响。忽略关系，只有得分最高的元素才会有非零导数，其导数为 1。因此，max reduction 的梯度是一个 one-hot 向量，1 代表得分最高的元素，即 argmax 函数。

以下是二进制分类函数如何与多类分类函数相关联的总结：

表 8.1: 二进制分类函数与多类别函数

二进制	多类别
ReLU	Max
Step	Argmax
Sigmoid	Softmax

也就是说，最大池化中，局部区域内最大值元素导数为 1，其他元素都是 0，这样就可以做反向传播了。

8.5.2 完成任务 4.4

大部分实现的内容比较简单，这里只说一下 dropout：

```
def dropout(input, rate, ignore=False):
  # TODO: Implement for Task 4.4.
  if not ignore:
      bit_tensor = rand(input.shape, input.backend) > rate
      input = bit_tensor * input
  return input
```

其实就是将一定比率的神经元乘以 0，这只在比较大规模的神经网络下才好用，规模比较小的时候则会非常不稳定。

8.6 cuda 上的卷积

该任务是一个额外任务，建立一个文件 cuda_conv.py 来实现 CUDA 上的 conv1d 和 conv2d 函数。

由于本人时间和兴趣关系，没有去做该任务。

8.7 训练图像分类器

现在需要看 project/run_sentiment.py 和 project/run_mnist_multiclass.py 这两个脚本。我们需要实现 Conv1D、Conv2D 和 Network 这几个类中的函数。

任务目标：

- 在 Sentiment (SST2) 上训练一个模型，模型应该有超过 75% 的准确率。
- 训练一个模型用于手写数字辨识。

8.7.1 网络函数

我们只看 run_mnist_multiclass.py 类，可以看到每次输入到网络中之前，张量的 batch 数为：

```
X_train[example_num : example_num + BATCH]
```

输入到网络中的 shape 变为：

```
x.view(BATCH, 1, H, W)
```

我们在第一层卷积定义 4 个卷积核，第二层定义 8 个卷积核，然后通过两个线性层。至于每层卷积以后的大小应该怎么计算，这里就不再赘述了：

```
class Network(minitorch.Module):
  def __init__(self):
    super().__init__()
    # For vis
    self.mid = None
```

```
        self.out = None
        # TODO: Implement for Task 4.5.
        self.layer1 = Conv2d(1, 4, 3, 3)
        self.layer2 = Conv2d(4, 8, 3, 3)
        self.layer3 = Linear(392, 64)
        self.layer4 = Linear(64, 10)
    def forward(self, x):
        # TODO: Implement for Task 4.5.
        self.mid = self.layer1(x).relu()
        self.out = self.layer2(self.mid).relu()
        pool = minitorch.avgpool2d(self.out, (4, 4)).view(BATCH, 392)
        x3 = self.layer3(pool).relu()
        if self.mode == "train":
            x3 = minitorch.dropout(pool, 0.25)
        x4 = self.layer4(x3)
        return minitorch.logsoftmax(x4, 1)
```

在上面的程序里，我们已经把前面实现过的内容都综合使用了一遍，例如 ReLU、卷积、dropout 和 pooling，如果能正常开始训练，说明我们的系统是比较完备的。当然，它的速度可能会很慢，我一个 epoch 都要训练好久好久，但至少已经具备了雏形。

8.8 结语

历时一个多月，终于在业余时间完成了这个小项目，期间，在写程序中出现了很多问题，但是在不断地调试和分析中也将这些问题克服了。其中一个非常值得记录的问题就是当初 Sum 函数的 backward() 我可能不小心删了点内容，结果导致 backward() 返回的 shape 和前面输入的 shape 不匹配，导致报错，我跟踪了很久的代码，才最终发现（这是意料之外的，而我一直以为是 View 函数出了问题）。

总之，这个用了一个多月的业余时间攻克的小项目就到此结束了，在这期间也是收获良多。最终，感谢 Alexander Rush 教授能够提供这么好的学习教程，虽然没有在线视频课公开课，但还是可以通过各种网络资源（比如 Cornell 大学学生的 Github 作业）来学习和理解该课程。这也是我第一次将深度学习和链式法则、雅克比矩阵结合起来。

我将这一个月的思考和学习整理成了这一本电子书，希望能够帮助更多想去学习这门课程但是自身水平比较弱的同学；还有比如像我这种，主要靠 C++ 工作，对 Python 很多机制都不熟悉的人，当初 Fundamental 章节的 map 函数的实现就花了我很长时间，在 C++ 中，函数内定义函数并不是一个常见的方法，需要时则会使用 Lambda 表达式去实现，而 Python 中则可以很容易地实现。

最终，再次感谢 Alexander Rush 教授以及 Python 视界公众号、机器之心公众号。我也是在 Python 视界公众号 2021 年 12 月 15 日推送的文章中了解到的该项目。

Bibliography

[1] https://github.com/minitorch

[2] https://minitorch.github.io/index.html

[3] https://minitorch.github.io/setup.html

[4] https://github.com/psf/black

[5] https://docs.pytest.org/en/stable/

[6] https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html

[7] https://minitorch.github.io/modules.html

[8] https://github.com/streamlit/streamlit

[9] https://minitorch.github.io/derivative.html

[10] https://minitorch.github.io/scalar.html

[11] https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types/

[12] https://minitorch.github.io/backpropagate.html

[13] https://en.wikipedia.org/wiki/Topological_sorting

[14] https://minitorch.github.io/tensordata.html

[15] https://numba.pydata.org/numba-doc/latest/user/parallel.html

[16] https://explained.ai/matrix-calculus/index.html

[17] Automatic Differentiation in Machine Learning: a Survey (2018)

[18] https://github.com/mihirjain-iitgn/MiniTorch

[19] https://github.com/Dearkano/MiniTorch

[20] http://rush-nlp.com/