MINITORCH 纯 python 实现的深度学习框架 LaTeX

神经网络大作业

汪福运 | 191300051[a]

[a]南京大学　人工智能学院　江苏南京　210013

关键词：自动求导；GPU 加速；多层感知机；对比试验

目录

1. 代码说明

***A.* 基本介绍.** 本代码借鉴了Cornell CS5781的minitorch，并根据神经网络课程大作业的要求，实现了一个基于纯 python(不依赖于 numpy 等科学计算库) 的 pytorch 风格的深度学习框架，支持并行计算，GPU 加速，模组嵌套，自动求导，张量的索引、广播、运算等特性，实现了包括全连接，卷积，池化，dropout 等网络模组，实现了多种损失函数，优化器及初始化方法等特性。

***B.* *package* 的安装.** 进入文件夹，执行命令

```
python -m pip install -r requirements.txt
python -m pip install -r requirements.extra.txt
python -m pip install -Ue .
```

成功后，即可本地使用 minitorch 进行程序的编写。如

***C.* 代码文件与相应功能.**

1. autodiff.py 实现自动求导
2. cuda_ops.py 实现 CUDA 的算子。
3. fast_conv.py fast_ops.py 利用 JIT 库实现 CPU 的并行计算加速。
4. module.py 实现网络 module 的嵌套，parameter。
5. nn.py 实现了 nn.conv2d, nn.linear, nn.maxpool2d, nn.averagepool2d,dropout 等网络模组，实现 cross_entropy, bce_loss, mse_loss 等损失函数。
6. optim.py 实现了 SGD, ADAM 优化器，实现了动态学习率调整和 weight decay 的正则选项。
7. operators.py 基本的标量运算函数。
8. tensor_data.py 实现了 tensor 的底层存储、索引转换、广播。
9. tensor_ops.py 无加速情况下的 tensor 的算子。

10. tensor_functions.py 在用户指定 backend(cuda 或者 cpu) 后，生成 tensor 的运算支持函数。每个函数都包含前向反向传播两部分，是实现自动求导的重要部件。

11. tensor.py 代码的核心 tensor 类。主要进行操作符的重载和定义 tensor 需具有的功能函数。

***D.* 报告内容概述.** 在下面的三节，我将框架代码实现过程中的难点分为三部分: 自动求导树的构建与反向传播2，张量的存储、索引、广播与运算3，并行 CPU 与 GPU 算符4进行介绍。

在后面，我根据课程作业的要求，分别探讨了参数初始化、损失函数、正则化、学习率调整带来的影响，在实验的过程中，我们还发现恰当的数据预处理在同等条件下，能够加快神经网络的训练，并提高泛化能力。

最后，我们在原有模型的基础上增加了卷积层和 dropout 层，显著降低了模型的参数量，并在数据受限的情况下仍能得到良好的性能。

2. 神经网络：自动求导树的构建与反向传播

自动求导树的节点为核心类 Variable。用户指定创建的 Variable 为叶子节点。通过 Variable 通过运算符进行运算得到的其他 Variable 为中间节点，它们需要存储生成自己的运算符 (last_fn)、父节点的索引 (inputs) 以及为了方便反向传播需要存储的其他变量 (context)。

当我们从根节点 (常常为 loss), 执行 backward 之后，我们迭代地向父节点传递梯度，并保留叶子结点的梯度值，并删除中间节点。(与 pytorch 一致，除非 pytorch 中显示要求 retain graph)。

一个简单的例子如图1 图中 x, y 为叶子节点，variable4 为中间节点，out 为根节点。当对于 out 进行反

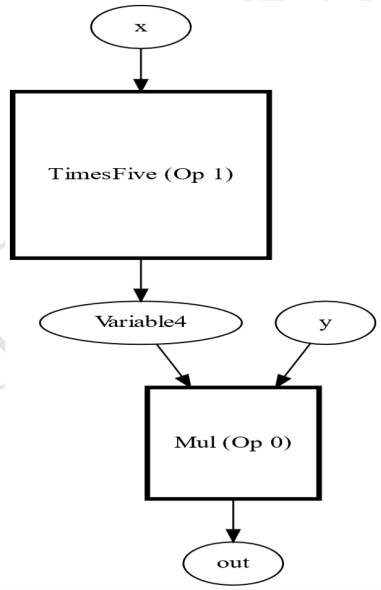


Fig. 1. 自动求导树实例

向传播时，会首先将梯度传播给 y 和 variabel4，即 y 的梯度为：variable4, variable 4 的梯度为 y。注意到 variable5 为中间节点，故需进一步反向传播，得到 x 的梯度为 $5y$。反向传播完成后，中间节点 variable4 被释放。

隐去部分细节，整个反向传播的过程可由下列代码表示

```
1 def backpropagate(variable, deriv):
2     variable_queue=topological_sort(variable=variable)#根据自动求导树，对variable进行拓扑
                                                        排序，然后顺序遍历进行梯度传播。
3     v_d_dict=dict(zip(list(map(lambda x:x.name,variable_queue)),[0 for i in range(len(
                                                        variable_queue))]))
```

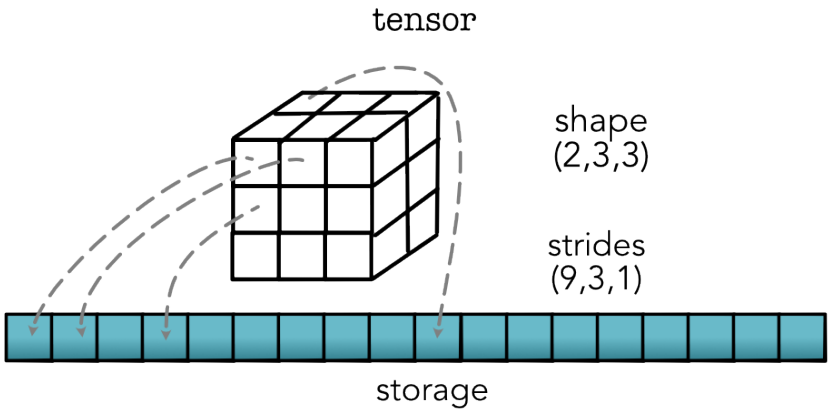


Fig. 2. 张量的索引转换

```
    v_d_dict[variable.name]=deriv
    for variable in variable_queue:
        if variable.is_leaf():#遍历到叶子节点之前由于拓扑排序，其梯度积累已经完成。
            variable.accumulate_derivative(v_d_dict[variable.name])
        else:
            for father_variable,father_deriv in variable.history.backprop_step(v_d_dict[
                                                    variable.name]):
                v_d_dict[father_variable.name]+=father_deriv
```

3. 神经网络：张量的存储、索引、广播与运算

在神经网络的运算中，将会涉及到大量地重复计算，使用循环来为每个元素执行同样的操作，会导致代码的重复和运算效率的显著下降。因此，以张量的形式进行计算一方面能够简化编程的复杂度，实现代码复用率，另外一方面能够提高计算效率，并支持更加高效的并行算法。

***A.* 张量的存储与索引.** 尽管在数学意义上，张量是可以是任意高维的数组，但是在程序上实现这种存储模式是十分困难且不利的。例如对于高维的张量进行 reshape，则可能设计到非常多的内存存取操作，这是极为耗时的。因此，我们需要明晰三个概念，shape，strides,storage。

Storage 是保存张量的核心数据的地方。无论张量的维度或形状如何，它始终是长度大小数字的一维阵列。保持一维存储允许我们将具有不同形状的张量指向相同类型的底层数据。

Strides 是一个元组，提供从用户索引到一维存储中位置的映射。

Shape 用户心中张量的实际形状。

也就是说，无论张量的维度是怎样的，张量实际在内存中是一个一维数组的形状。假设索引为 $(s_1, s_2, s_3 \dots)$ 当我们希望访问该张量的某个索引的数据时，当中的运算过程

$$\text{storage}[s_1 * \text{index}_1 + s_2 * \text{index}_2 + s_3 * \text{index}_3 ...]$$

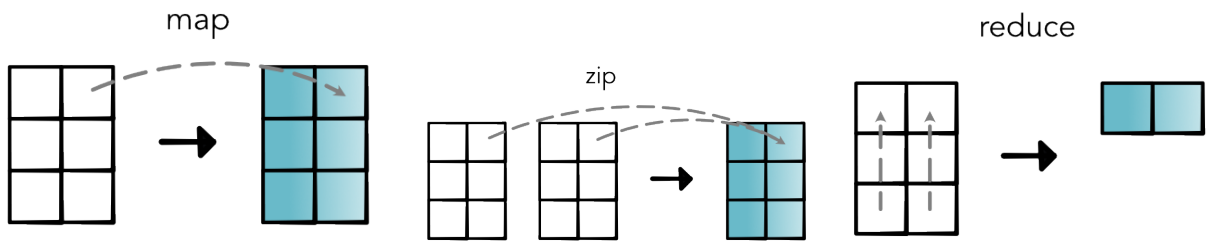
一个简单的例子如图2所示

***B.* 张量的广播.** 在张量的计算中，我们常常要处理两个维度并不完全匹配的向量的运算，如向量的数乘，需要对向量当中的每个元素都与指定的标量进行相乘，这就应用到了广播的思想。

关于广播有三条基本的规则：

1. 任何一的维度都可以被广播为 n 维。
2. 可以在张量形状的左侧增加任意多个一的维度来使形状匹配。
3. 只能在左侧增加一的维度。

例如 tensor1.view(1, 2, 3, 1) + tensor2.view(7, 2, 1, 5) 得到的结果的形状为 (7,2,3,5)。



(a) map 操作　(b) zip 操作　(c) reduce 操作

Fig. 3. tensor 运算示意图

***C.* 张量的运算.** 张量的基本运算主要包括三种，示意图如3所示：

1. map 对一个输入张量的每个元素进行同样的重复操作，且每个元素的操作与其他位置的元素的取值无关。如 sigmoid,negative 等。
2. zip 对两个输入张量的对应位置的两个元素进行操作。如 add，sub 等。
3. reduce 对张量的维度进行缩减，归纳运算。如 sum,mean 等。

4. 神经网络：并行 CPU 与 GPU 算符

在本节，我将以 map 为例，首先介绍 map 的原始实现，然后说明我如何实现 cpu 的并行与 GPU 加速。

A. MAP 的基本实现

```
1
2  def tensor_map(fn)
3      def _map(out, out_shape, out_strides, in_storage, in_shape, in_strides):
4          out_index=np.array(out_shape)
5          in_index=np.array(in_shape)
6          for i in range(len(out)):
7              to_index(i,out_shape,out_index) # 获得out的索引
8              broadcast_index(out_index,out_shape,in_shape,in_index) # 将out的索引利用广播
                                                                      机制找到对应的in的索引
9              out[index_to_position(out_index,out_strides)]=fn(in_storage[index_to_position
                                                                      (in_index,in_strides)]) # 对in
                                                                      的索引位置的数据进行fn函数的转
                                                                      化存储到out对应位置。
```

B. MAP 的 *JIT* 加速

```
1  def tensor_map(fn)
2      def _map(out, out_shape, out_strides, in_storage, in_shape, in_strides):
3          for i in prange(len(out)):# 此处使用prange 替代range，多线程自动分配执行不同的循
                                                                      环
4              out_index=out_shape.copy()
5              in_index=in_shape.copy()
6              ordinal=i+0 #进行数据的拷贝，保证并行不会修改公共变量
7              to_index(ordinal,out_shape,out_index)
8              broadcast_index(out_index,out_shape,in_shape,in_index)
9              out[index_to_position(out_index,out_strides)]=fn(in_storage[index_to_position
                                                                      (in_index,in_strides)])
```

```
    return njit(parallel=True)(_map)
```

C. MAP 的 CUDA 加速.

C.1. CUDA 编程的基本介绍. 如图4所示，我们可以将一个线程视为一个机器人，每个机器人能够负责一小块内存的读取与计算。

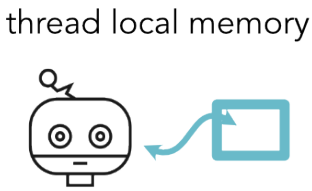


Fig. 4. 线程比喻

而正如图5所示，GPU 拥有海量的线程，每个线程属于某一个 block，block 的堆叠构成了 grid。这样，我们只需要知道一个线程在 grid 当中的 blokcId 以及在自己所在 Block 当中的 threadId 就可以访问和使用该线程，利用这种数字关系，对于某些具有重复或者规律性的任务，能够极大地提高运算性能。

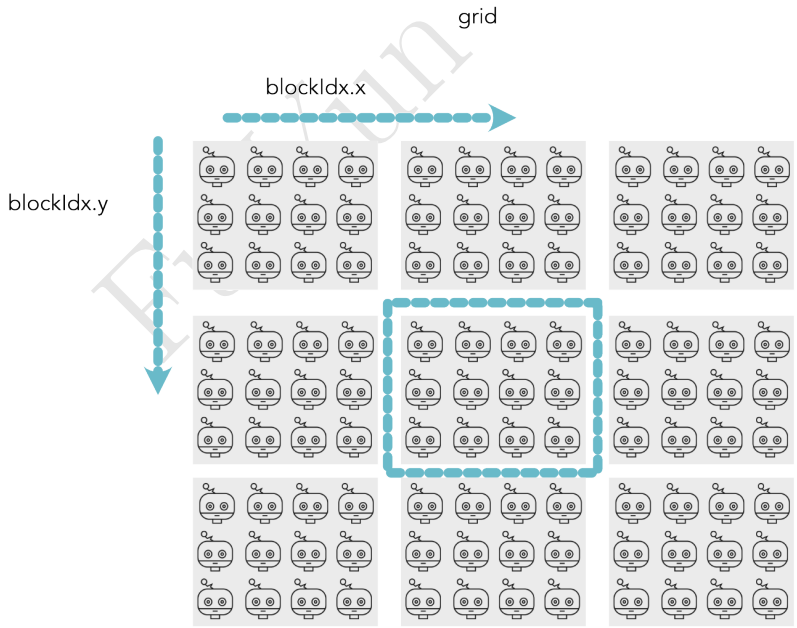


Fig. 5. GPU 的线程组织

C.2. 编程实现

```
def tensor_map(fn):
    def _map(out, out_shape, out_strides, out_size, in_storage, in_shape, in_strides):
        assert len(out) == out_size
```

```
        i = cuda.blockDim.x*cuda.blockIdx.x+cuda.threadIdx.x
        if i >= out_size:
            return
        out_index = cuda.local.array(MAX_DIMS, numba.int32) # 每个线程自己保留一部分本地内存
        in_index = cuda.local.array(MAX_DIMS, numba.int32)
        to_index(i, out_shape, out_index)
        broadcast_index(out_index, out_shape, in_shape, in_index)
        out[index_to_position(out_index, out_strides)] = fn(
            in_storage[index_to_position(in_index, in_strides)]) #每个线程处理一个元素的运算

    return cuda.jit()(_map)


def map(fn):
    f = tensor_map(cuda.jit(device=True)(fn))

    def ret(a, out=None):
        if out is None:
            out = a.zeros(a.shape)

        threadsperblock = 32 # 配置每个block的线程数量
        blockspergrid = (out.size + (threadsperblock - 1)) // threadsperblock # 根据tensor的大小确定每个grid的block数目确保可以全部都能运算。
        f[blockspergrid, threadsperblock](*out.tuple(), out.size, *a.tuple()) # 调用CUDA的kernel函数进行执行
        return out

    return ret
```

5. 构建多层感知机

***A.* 模型.** 本实验使用的模型非常的简单，如果将当中的 minitorch 改为 torch.nn 则与 PyTorch 并无区别。

```
class MLP(minitorch.Module):
    def __init__(self):
        super().__init__()
        self.l1=Linear(28*28,512)
        self.l2=Linear(512,256)
        self.l3=Linear(256,C)


    def forward(self, x):
        x=x.view(x.size//(28*28),28*28)
        x=self.l1(x).relu()
        x=self.l2(x).relu()
        x=self.l3(x)
        return x
```

注意到在最后的输出层，我并没有使用任何的激活函数，这样方便在后面测试不同的损失函数带来的影响。

***B.* 优化器.**

B.1. 优化器基本原理

```
class SGD():
```

```
    def __init__(self,parameters,lr=0.01,weight_decay_l2=None):
        保留参数
    def step(self):
        # 参数更新
    def zero_grad(self):
        # 参数的梯度置0
```

优化器主要提供三个 API，主要功能是存储 model.parameters() 的引用。当执行 step 时，则让每个可导的参数都减去 learning rate 乘上其对应的梯度。当执行 zero grad 时清楚每个参数上积累的梯度。

***B.2.* 实现正则化的技巧.** 尽管从数学意义上，加入参数的正则化对应了在 loss 上面加入参数的二范数的惩罚项，即

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{ce} + \lambda \|\boldsymbol{w}\|_2$$

但实际上我们并不需要这么做，只需要在每次更新梯度时同时减掉 $2 * \text{learning rate} * \lambda * p.value$ 。具体的反映在代码上为

```
            for p in self.parameters:
                if p.value.grad is not None:
                    p.update(p.value - self.lr * (p.value.grad)-self.weight_decay_l2*2*p.
                                                                    value)
```

***C.* 损失函数.** 共实现了三种常用的损失函数

1. SoftMax+ 交叉熵损失函数. 将模型的输出通过 softmax 在进行交叉熵求损失。与 pytorch 种的 F.cross_entropy 一致。
2. Sigmoid+BCE. 将模型的输出通过 sigmoid 再使用 Binary cross entropy 进行运算，与 pytorch 种的 BCEwithlogits 一致。
3. MSE. 使用回归问题常用的损失来直接用于分类问题。注意我们这里最后一层并没有使用 sigmoid 的因此不会出现课件中出现的梯度消失的情况。

***D.* 初始化.** 共实现了三种初始化方式

1. 随机初始化，认为指定 r，在 $[-r, r]$ 之间均匀采样。
2. He 初始化，指定 r 为 $\sqrt{\frac{6}{M_{l-1}}}$
3. 全零初始化：经过多次实验验证，发现全零初始化确实非常容易因为权值对称的问题导致梯度爆炸或者梯度消失的问题。即使正确训练，其最终的泛化性能也不理想。

***E.* 参数初始化与数据预处理的关系探究.**

***E.1.* 实验过程中出现的参数初始化问题.** 当使用原数据进行训练时，我设置 $r = 0.005$，learning rate= 0.01，将 r 或者 learning rate 扩大十倍都会出现梯度爆炸的问题。

但是，当我对图像数据做了 Z-score 的预处理后，在没有预处理情况下可以稳定运行的参数却出现了梯度消失的问题。我将 r 和 learning rate 同时扩大了十倍，才能够快速的收敛得到预期的结果。

***E.2.* 可能的解释.** 因为原始数据是在 0-255 之间的整数，因此原始输入的值的变化可能很大，因此将参数设置为较小的值能够得到好的结果。参数这是过大可能会导致输出随着输入剧烈波动或者产生溢出。经过预处理后，输入的波动变小，因此需要将参数初始化为较大的值，否则输出的波动可能非常小导致梯度消失。

6. 多层感知机对比试验

***A.* 实验配置.** 虽然我们对运算符进行了并行计算的加速，但整体的加速仍然是在 python 语言上进行的。因此实验的数据规模较小。具体的，训练集样本数:500, 测试集样本数:200, 批的大小: 50 。

***B.* 实验比较.**

***B.1.* 预处理带来的影响.** 实验结果如图6所示，可以看到加入归一化后模型的收敛速度明显加快，且验证集准确率也有了一定的提升。需要注意的是这里是否归一化的情况下设置的学习率与初始化方式并不相同，具体原因请查看节E。

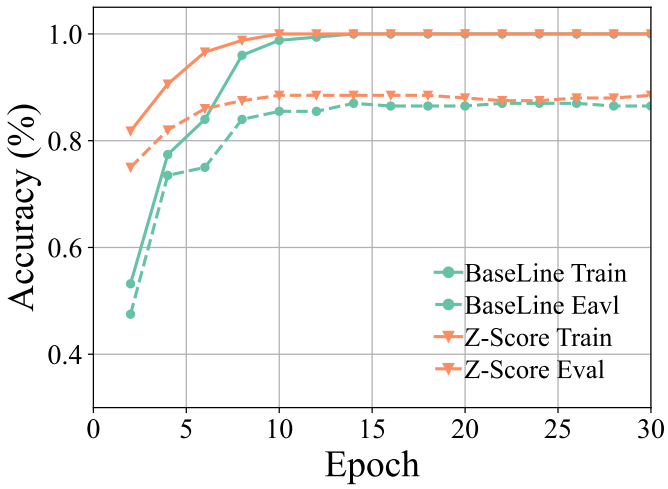


Fig. 6. 是否进行 Z-score(归一化) 处理的对比结果

***B.2.* 学习率调整的影响：对比 *ADAM* 和 *SGD*.** 在一般情况下使用 ADAM 和 SGD 所得到的结果如图7所示，可以看到由于动态学习率的调整，ADAM 训练集准确率的的提高要快于 SGD, 但最后的测试集准确率却更低。可能是由于 ADAM 中动量的使用导致了过拟合。

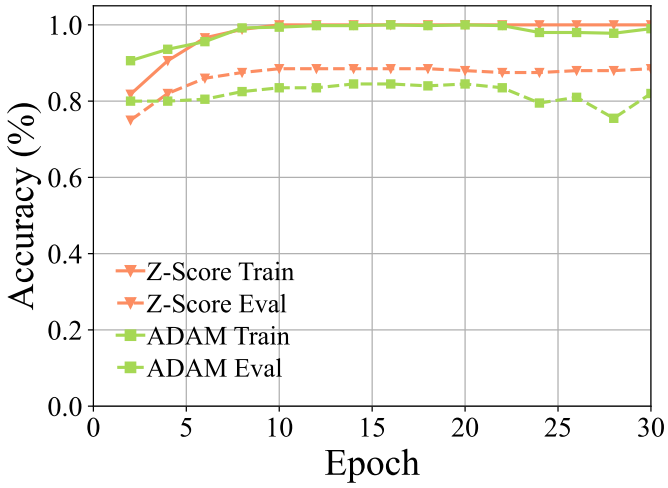


Fig. 7. ADAM 和 SGD 对比

***B.3.* 初始化带来的影响.** 使用三种不同的初始化方式训练模型得到的结果如图8所示。可以看到使用 He 初始化的显著快于 Z-score 情况下我们随机初始化的结果，证明了 He 初始化的有效性。另外一方面，注意到图中的黄色线段使用全零初始化由于权值对称导致了梯度消失的问题，模型准确率保持不变。

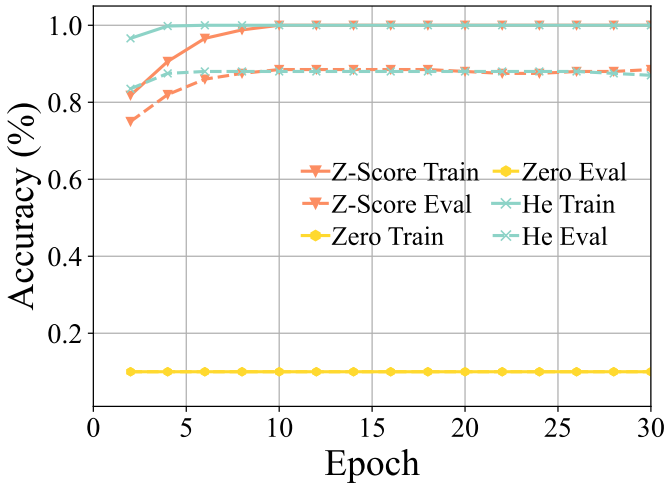


Fig. 8. 初始化带来的影响

***B.4.* 正则化的影响：使用 *L2* 范数.** 使用 L2 范数的结果如图9所示。图9中的蓝色线段使用了较大的惩罚系数，导致模型出现了欠拟合的情况，训练结果不佳。而使用了恰当惩罚系数的紫色曲线，能够取得略优于原始情况的结果。

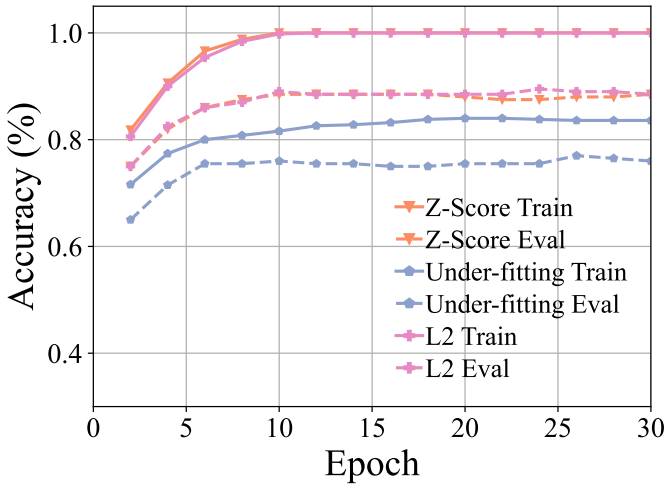


Fig. 9. 是否使用 L2 范数的对比结果

***B.5.* 损失函数的影响：探究学习率和优化器对损失函数的影响.** 损失函数带来的影响如图10所示。通过实验我发现，损失函数的使用对于优化器和学习率是较为敏感的。例如，MSE 的损失函数在使用 SGD 达到了三种损失函数的最优结果 (意料之外)，拥有最快的收敛速度和最好的泛化性能。但是，当使用 ADAM 优化器时，其表现如图10所示，又显著弱于另外两种损失函数。

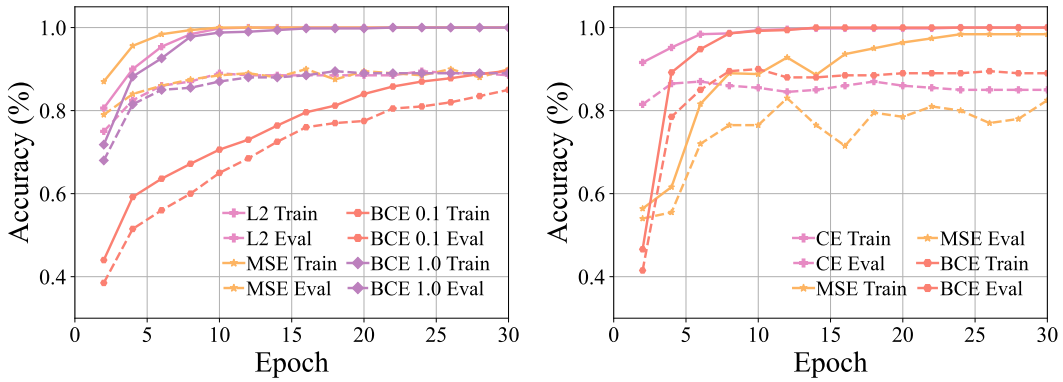
此外，BCE 损失在使用 SGD 的情况下，对于学习率较为敏感，当其学习率与另外两种损失函数相同时，其学习速率如图中红色曲线，非常缓慢，只有我们设置了较大的学习率后，它才能取得与另外两种损失函数相似的结果。一种可能的原因是因为使用了 sigmoid 函数，(1-sigmoid)sigmoid 的值会将梯度缩小。但是，当我们使用了 ADAM 的优化器时，BCE 取得了最优的泛化性能。7 轮训练后，其验证集准确率到达 90%。

C. 最终的最优结果.

最终最优的结果有两组

【随机初始化 +MSE 损失函数 +SGD 恒定学习率 +L2 正则化】得到了 90% 的准确率。

【随机初始化 +BCE 损失函数 +ADAM 动态调整学习率 +L2 正则化】得到了 90% 的准确率。



(a) 当使用 SGD 优化　　(b) 当使用 ADAM 优化

Fig. 10. 使用不同的损失函数的对比

进入 project 运行 run_mnist_multiclass.py 即可。

7. 卷积神经网络

***A.* 卷积神经网络的结构.** 28x28 的图像输入首先通过两个卷积层，并执行等长度的 padding。于是 28x28 的图象会转变为 4x28x28 的特征图，然后转变为 8x28x28 的特征图。然后通过 4x4 的 maxpooling 的池化操作得到 8x7x7 的特征图。然后将 8x7x7 的特征图展平为 392 的向量。然后通过全连接层的得到长度为 128 的向量。通过 dropout 层，以 0.25 的概率将特征向量进行掩码，再进行 4/3 的数乘操作 (保证与测试阶段的期望输出与训练时的期望输出相等), 最后再通过全连接层转变为长度为 10 的向量，每一个维度表征一个类别的置信度的大小。

```
class Network(minitorch.Module):
    def __init__(self):
        super().__init__()
        self.mid = Conv2d(1,4,3,3)#36
        self.out = Conv2d(4,8,3,3)#216
        self.l1=Linear(392,128)#392 * 128
        self.l2=Linear(128,C)# 128 *10
    def forward(self, x):
        x=self.mid(x).relu()
        x=self.out(x).relu()
        x=minitorch.nn.maxpool2d(x,[4,4])
        x=x.view(x.size//392,392)
        x=self.l1(x).relu()
        if self.training:
            x=minitorch.nn.dropout(x,0.25)*4/3
        x=self.l2(x)
        return x
```

8. 实验对比

同样使用 500 张训练样本，ADAM 优化器，交叉熵损失进行训练，卷积神经网络与多层感知机的实验结果如图11所示。注意到，卷积神经网络的训练效率与多层感知机结果相仿，但是其测试集准确率略优于多层感知机的结果。

同时注意到，我们的卷积神经网络的参数量为 36+216+392*128+128*10+4+8+128+10=51858，而多层感知机的参数量为 28*28*512+512*256+256*10+512+256+10=535818。我们的卷积神经网络相较于多层感知机参数量降低了十余倍，仍能取得更优的结果，展现了卷积操作在图像处理当中的优越性。

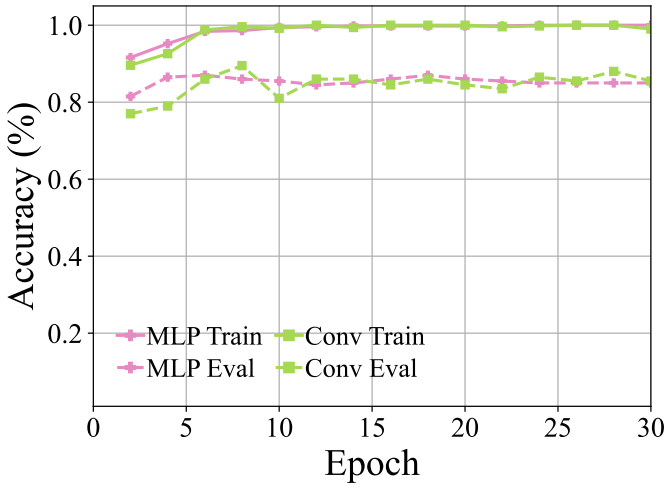


Fig. 11. 卷积神经网络与多层感知机对比