作业 4 PyTorch 深度学习

郝千越 2018011153 无 85

目录

任务一：使用 PyTorch 搭建神经网络

1 环境配置

本实验涉及深度学习模型的训练，故使用课程提供的服务器以加快训练速度，首先在服务器上配置相关环境。

Anaconda 是目前比较热门的集成环境管理工具，可以支持不同 python 版本的独立虚拟环境。使用镜像源安装 Miniconda（只包括环境管理工具，不包括预装 package 的轻量级版本），创建虚拟环境 env0 并使用“conda activate env0”进入该虚拟环境。在该虚拟环境中使用“conda install”或“pip install”命令补充安装常用的 package。

使用“nvidia-smi”命令查看 GPU 硬件型号以及驱动版本：

```
Sun Dec 13 18:59:35 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.66       Driver Version: 450.66       CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  TITAN RTX           Off  | 00000000:1A:00.0 Off |                  N/A |
| 54%   74C    P2    77W / 280W |  22488MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  TITAN RTX           Off  | 00000000:1B:00.0 Off |                  N/A |
| 98%   88C    P2   137W / 280W |  19529MiB / 24220MiB |     70%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  TITAN RTX           Off  | 00000000:3D:00.0 Off |                  N/A |
| 41%   41C    P2    66W / 280W |  17966MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  TITAN RTX           Off  | 00000000:3E:00.0 Off |                  N/A |
| 41%   56C    P2    78W / 280W |  17966MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   4  TITAN RTX           Off  | 00000000:88:00.0 Off |                  N/A |
| 41%   46C    P2    64W / 280W |  17966MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   5  TITAN RTX           Off  | 00000000:89:00.0 Off |                  N/A |
| 41%   52C    P2    67W / 280W |  17966MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   6  TITAN RTX           Off  | 00000000:B1:00.0 Off |                  N/A |
| 41%   50C    P2    63W / 280W |  17966MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   7  TITAN RTX           Off  | 00000000:B2:00.0 Off |                  N/A |
| 41%   48C    P2    61W / 280W |  17966MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

图 1 查看 GPU 相关信息

可以看到服务器搭载了 8 块显存 24GB 的 Nvidia TITAN RTX 的 GPU，安装的驱动版本为 450.66，CUDA 版本为 11.0。根据这一信息安装对应的 CUDA 版本 Pytorch，完成后可用命令“torch.cuda.is_available()”查看安装及支持是否成功，返回 True 证明安装已经成功：

```
>>> import torch
>>> torch.cuda.is_available()
True
```

图 2 确认 CUDA 版本 Pytorch 安装成功

另外在环境中安装 jupyter，便于进行 Notebook 的调试，后使用“conda list”命令查看当前环境中已经安装的 package：

```
# packages in environment at /home/NMDAhaoqianyue/anaconda3/envs/env0:
#
# Name                    Version                   Build  Channel
_libgcc_mutex             0.1                 conda_forge    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
_openmp_mutex             4.5                       1_llvm   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
argon2-cffi               20.1.0          py36h1d69622_2    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
async_generator           1.10                       py_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
attrs                     20.3.0             pyhd3deb0d_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
backcall                  0.2.0              pyh9f0ad1d_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
backports                 1.0                        py_2   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
backports.functools_lru_cache 1.6.1                    py_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
blas                      1.0                         mkl    https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
bleach                    3.2.1              pyh9f0ad1d_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
ca-certificates           2020.12.5           ha878542_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
certifi                   2020.12.5       py36h5fab9bb_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
cffi                      1.14.4          py36hc120d54_1    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
cudatoolkit               10.1.243            h036e899_6    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
cycler                    0.10.0                     py_2   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
decorator                 4.4.2                      py_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
defusedxml                0.6.0                      py_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
entrypoints               0.3             pyhd8ed1ab_1003   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
fontconfig                2.13.1            h7e3eb15_1002   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
freetype                  2.10.4              h7ca028e_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
icu                       67.1                he1b5a44_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
importlib-metadata        3.1.1              pyhd8ed1ab_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
importlib_metadata        3.1.1                hd8ed1ab_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
ipykernel                 5.4.2           py36he448a4c_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
ipython                   7.12.0          py36h5ca1d4c_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
ipython_genutils          0.2.0                      py_1   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
ipywidgets                7.5.1              pyh9f0ad1d_1   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jedi                      0.10.2                  py36_2    https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
jinja2                    2.11.2             pyh9f0ad1d_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jpeg                      9d                  h36c2ea0_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jsonschema                3.2.0                      py_2   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jupyter                   1.0.0                      py_2   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jupyter_client            6.1.7                      py_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jupyter_console           6.2.0                      py_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jupyter_core              4.7.0           py36h5fab9bb_0    https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
jupyterlab_pygments       0.1.2              pyh9f0ad1d_0   https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
```

图 3 查看当前环境中的 package（部分）

2 数据集概述

Cifar-10 包括十个类别，共 60000 张 32*32 的彩色图片，其中 50000 张为训练集样本，10000 为测试集样本。其中部分图片如下图所示：

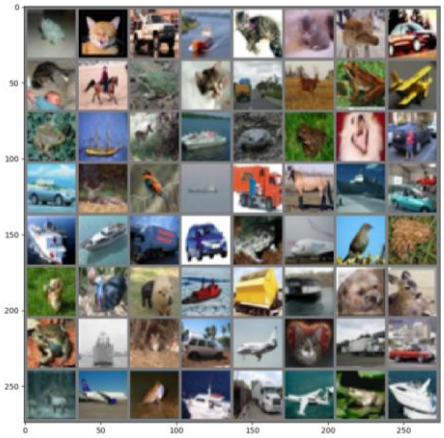


图 4 Cifar 部分样本示例

3 深度学习框架 PyTorch 的基本使用

3.1 数据加载

PyTorch 中提供了丰富的数据预处理、载入工具。简述如下：

torch.utils.data 中包含了 dataset 对象。通过查看相关源代码可以得知，该对象初始化时生成两个 List 成员，分别记录了所有数据的文件路径（并不是将所有数据载入内存，否则很容易造成内存溢出）和对应的分类标签。该对象中还包括两个基本函数：__getitem()__函数根据给定索引将对应的数据读入并返回，同时返回数据标签；__len__()函数返回数据集长度。默认函数仅支持 jpg、png 等图像格式的数据载入，通过继承 dataset 对象并重载上述两个函数，可以实现自定义数据集对象，处理需要的数据集。同时，还可以指定 transform 参数，在载入数据后进行预处理。

torchvision.transforms 提供了数据预处理、数据增强的方法。如：尺寸调整、裁剪、旋转、镜像、归一化等等。其中 Compose 对象可以将多个处理方法包装成一个 transforms 对象，传入 dataset 对象中对数据进行处理

torchvision.datasets 中提供了常用的计算机视觉数据集，包括 MNIST、Cifar、COCO、VOC 、 ImageNet 等 。 通 过 torchvision.datasets.CIFAR10(root='./data',train=True, download=True,transform=transform)可以返回 dataset 对象。该函数调用时会在指定目录下搜索是否已经存在 Cifar-10 数据集，如果存在则载入并返回对应的 dataset 对象；如果不存在则先下载 Cifar-10 数据集，再返回对应的 dataset 对象。

torch.utils.data.random_split 可以按照指定比例将一个 dataset 对象划分为两部分，由此实现训练集、验证集的随机划分。

torch.utils.data.DataLoader 接收 dataset 对象，根据指定的 batch_size 和是否随机打乱参数，将一个 batch 的数据包装成，在训练或测试中使用。该函数支持多线程读取数据，大大加速了数据载入。

3.2 网络结构定义

PyTorch 中提供了便利地网络定义方式：

torch.nn 提供了常用神经网络的各种层，如 BN 层、卷积层、池化层、全连接层等。

torch.nn.functional 中提供了常用的各种激活函数、softmax 函数、池化函数。

构建一个以 nn.Module 为基类的类即可自定义网络，指定 forward 方法即可定义网络前向传播的方式。同时，nn.Squential 可以将多个模块包装成一个模块，并自定定义 forward 方法为顺序通过，由此可以方便地实现网络中模块的复用（这种情况在较大规模的网络如 GoogleNet、ResNet 中非常常见）

本实验中首先使用以下较为简单的网络进行训练，探究多种超参数变化对网络表现的影响，再使用较复杂的网络进行训练以获得较好的性能表现。

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3,out_channels=6,kernel_size=5)
        self.pool = nn.MaxPool2d(kernel_size=2,stride=2)
        self.conv2 = nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5)
        self.fc1 = nn.Linear(in_features=16*5*5,out_features=120)
        self.fc2 = nn.Linear(in_features=120,out_features=84)
        self.fc3 = nn.Linear(in_features=84,out_features=10)

    def forward(self, x):
        x = self.pool(F.leaky_relu(self.conv1(x),0.2))
        x = self.pool(F.leaky_relu(self.conv2(x),0.2))
        x = x.view(-1, 16 * 5 * 5)
        x = F.leaky_relu(self.fc1(x),0.2)
        x = F.leaky_relu(self.fc2(x),0.2)
        x = self.fc3(x)
        return x
```

该网络首先经过卷积核边长为 5 的卷积层将输入通道数由 3 扩展至 6；由 relu 函数激活并通过 2*2 最大池化；再经过卷积核边长为 5 的卷积层将输入通道数由 6 扩展至 16；由 relu 函数激活并通过 2*2 最大池化；最后依次通过三个全连接层，特征数由 400-120-84-10 减少，每层之间由 relu 函数激活。

3.3 损失函数

Pytorch 内部 torch.nn 中有多种常用的损失函数，例如：

- **CrossEntropyLoss**：这一损失函数常用在多分类任务中。根据官方文档对于该函数的定义，函数内部首先通过 softmax 函数对输出结果进行处理，再对所得各类别的概率计算交叉熵，总计算式如下：

$$\text{loss}(x, \text{class}) = -\log\left(\frac{\exp(x[\text{class}])}{\sum_j \exp(x[j])}\right) = -x[\text{class}] + \log\left(\sum_j \exp(x[j])\right)$$

- **L1Loss**：计算网络输出和真实值的平均绝对误差；
- **MSELoss**：计算网络输出和真实值的均方误差；
- **NLLLoss**：经过 softmax 函数后，计算网络输出和真实值的负对数误差；
- **SmoothL1Loss**：类似 L1Loss，但当网络输出值与真实值的绝对误差小于 1 时，返回 0.5 倍绝对误差的平方；绝对误差大于 1 时返回绝对误差减 0.5。相比于 MSELoss，该损失函数对于离群点的敏感程度较低。

3.4 优化方法

Pytorch 内部 torch.optim 中有多种常用的损失函数，其中部分方法可以自适应学习率从而达到较好的训练效果，例如：

- **SGD**：这一方法根据每次取出的小批量样本求出损失函数的梯度，根据指定学习率更新梯度。该方法如果不指定动量参数（momentum）则默认不使用动量法，因此学习率为固定值，不利于在训练前期较快更新参数和训练后期微调参数。使用动量参数，能够在每次更新时保留部分上次更新的“惯性”，有利于使得训练脱离局部极值点从而得到全局最优的位置。
- **Adagrad**：缩放每个参数反比于其所有梯度历史平方值总和的平方根，从而使得梯度大的参数学习率下降快，梯度小的参数学习率下降慢，实现自适应学习率的效果；
- **RMSprop**：类似 torch.optim.Adagrad，但在累计所有梯度历史平方值总的平方根时用数衰减平均以丢弃遥远过去的历史；
- **Adam**：通过一阶矩、二阶矩来估计参数更新幅度，可以实现自适应学习率，在训练初期快速更新，而在训练后期减小更新幅度。

另外，Pytorch 中还提供了 torch.optim.lr_scheduler 库，内部有多种自定义学习率衰减方式的函数，可以根据需求，手动调节学习率在训练过程中的变化情况。

4 模型表现探究

本实验中探究分为以下两步：

- 首先使用 3.2 中所述的简单网络结构（以便加快探究速度）探究不同网络结构、优化方法、超参数对于训练结果的影响。此时由于为了获得尽可能多的训练样本，将全部 50000 张训练集图像用作训练，用 10000 张测试集图像兼做验证集，得到训练过程中的模型表现。
- 再使用较为成熟且复杂的神经网络模型，固定超参数进行训练，得到尽可能好的测试结果。此时严格按照训练集、验证集的要求，按照 4:1 划分全部训练样本用于训练和验证。最后使用训练完成的模型在测试集上测试得到最终结果，见第 5 节。

4.1 网络结构探究

首先训练 3.2 中所述的简单网络结构，使用 batch_size=4，SGD 优化器，lr=0.001，momentum=0.9。训练过程中各项指标变化如下：

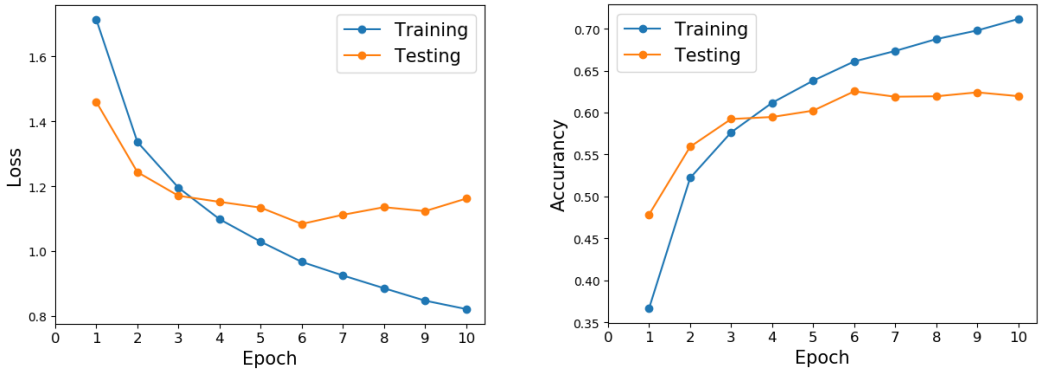


图 5 简单网络训练过程

可以看到训练过程中开始时 Loss 下降较快、Accuracy 提升较快，而训练后期 Loss 下降和 Accuracy 提升减缓。最终 Loss 和 Accuracy 基本保持不变，训练集 Loss 小于测试集而 Accuracy 高于测试集。训练完成时测试集最高准确率为 62.5%。

下面探究不同网络结构对训练过程以及模型表现的影响，使用 github 上的开源工具 thop.profile 统计各种网络下每一张图片样本所需的计算量：

表 1 不同网络结构的影响

网络编号	参数数目	运算量（FLOPs）	Top-1 准确率
1	62006	658025	62.5%
2（加通道）	114174	1226792	66.7%
3（减通道）	36246	354056	57.3%
4（加卷积）	114646	822024	62.1%
5（换激活）	62006	658025	65.3%

网络 1、2、3、4 具体为：

- 网络 1：3.2 中所述的简单网络
- 网络 2：原始网络中卷积层数目不变，但通道数由原来的 3-6-16 变为 3-8-32
- 网络 3：原始网络中卷积层数目不变，但通道数由原来的 3-6-16 变为 3-4-8
- 网络 4：原始网络中 conv1、conv2 通道数目不变，在 conv2 后再增加一个输入通道数为 16，输出通道数为 32，卷积核边长为 3，padding=1 的卷积层
- 网络 5：原始网络中结构不变，激活函数改为 leak-relu 函数，负半轴斜率设为 0.2

使用上述相同训练设置，将不同网络训练过程中的 Training Loss 和 Testing Loss 曲线画出如下：

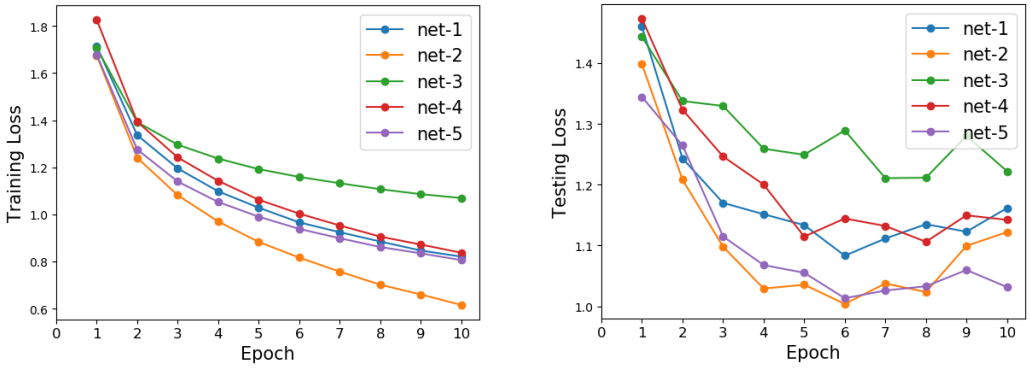


图 6 不同网络结构对比

可以看到增加通道数目后 Loss 下降明显快于其他，分类表现也优于其他；减少通道数目后 Loss 下降明显变缓，分类表现也有所下降。由此可以发现更多的通道数能够更加充分地提取特征，从而获得更好的分类效果。

同时，将激活函数从 Relu 函数改为负半轴也有一定斜率的 Leaky-Relu 后，模型分类表现也有所提高。推测其原因如下：Relu 函数在负半轴全部取值为 0，形成了一定数量的“dead neures”使得网络稀疏化，从而造成网络表现下降。改为 Leaky-Relu 后，所有神经元被充分利用，网络表现提升。

4.2 优化方法探究

下面探究不同优化方法下模型训练过程中的收敛速度，分别选取 SGD、SGD+ momentum、Adam 几种优化方法对原始网络进行训练，取 batch_size=4，训练过程中的 Training Loss 和 Testing Loss 曲线如下：

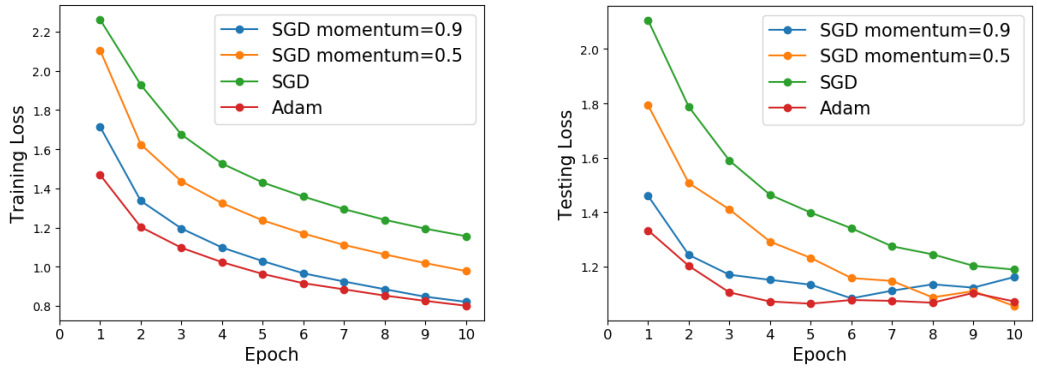


图 7 不同优化方法对比

可以发现 Loss 下降并收敛的速度大致为

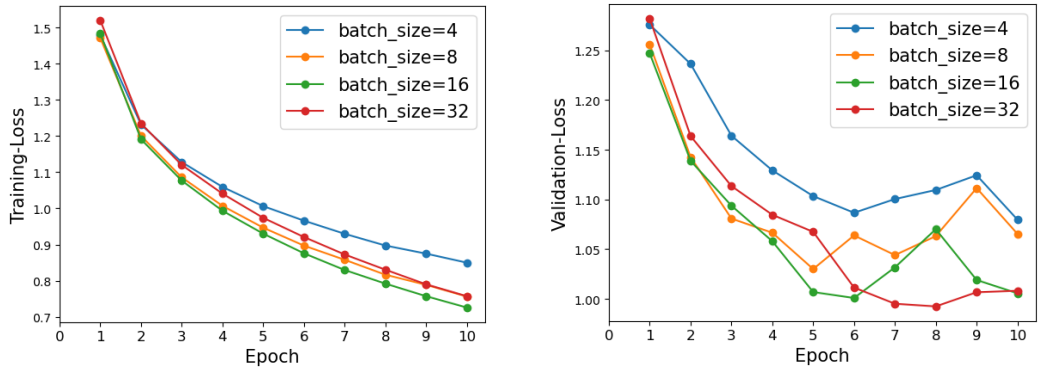
$$\text{Adam} > \text{SGD momentum} = 0.9 > \text{SGD momentum} = 0.5 > \text{SGD}$$

因此，使用较为高级的自适应学习率算法的 Adam 在优化中最为有效，使用了较大动量值的 SGD 方法一定程度上能够跳出局部极值，也比较有效。而最为基本的固定学习率 SGD 方法的表现较差。

4.3 超参数探究

下面进行 batch_size 和 learning_rate 两个超参数对模型表现的探究。

首先使用 Adam 优化器，learning_rate 固定为 0.001，分别设置 batch_size 为 4,8,16,32 进行训练，训练过程中各项指标变化如下：



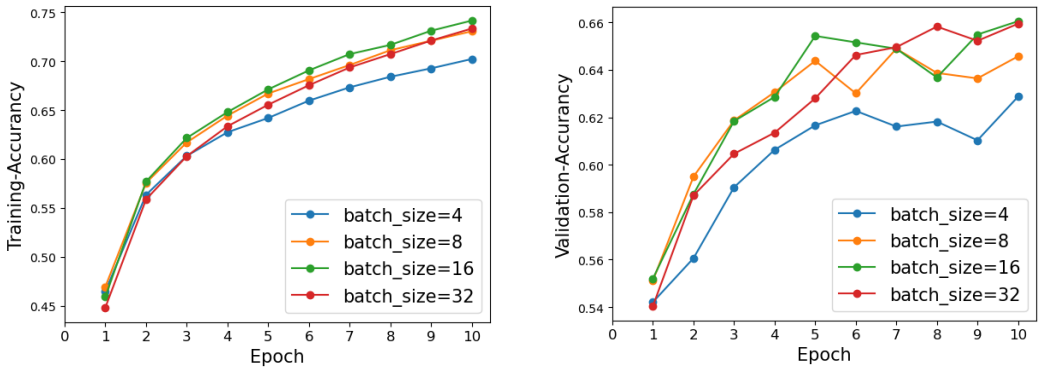


图 8 不同 batch_size 对比

可以看出 batch_size=4 时，各项指标均明显差于 batch_size 较大时的情形。而 batch_size 为 16 或 32 时，模型个表现总是较好的。由此可以发现，Minibatch 的规模增大时，网络每次可以接受到较多的样本信息，从而得到的梯度方向也更加接近全局梯度，故收敛较快，模型表现也相对较好。因此，在内存条件允许时，应当尽量使用较大的 batch_size，获得较好的模型表现。

再使用 Adam 优化器，batch_size 固定为 16，分别设置 learning_rate 为 0.01,0.005,0.001,0.0005 进行训练，训练过程中各项指标变化如下：

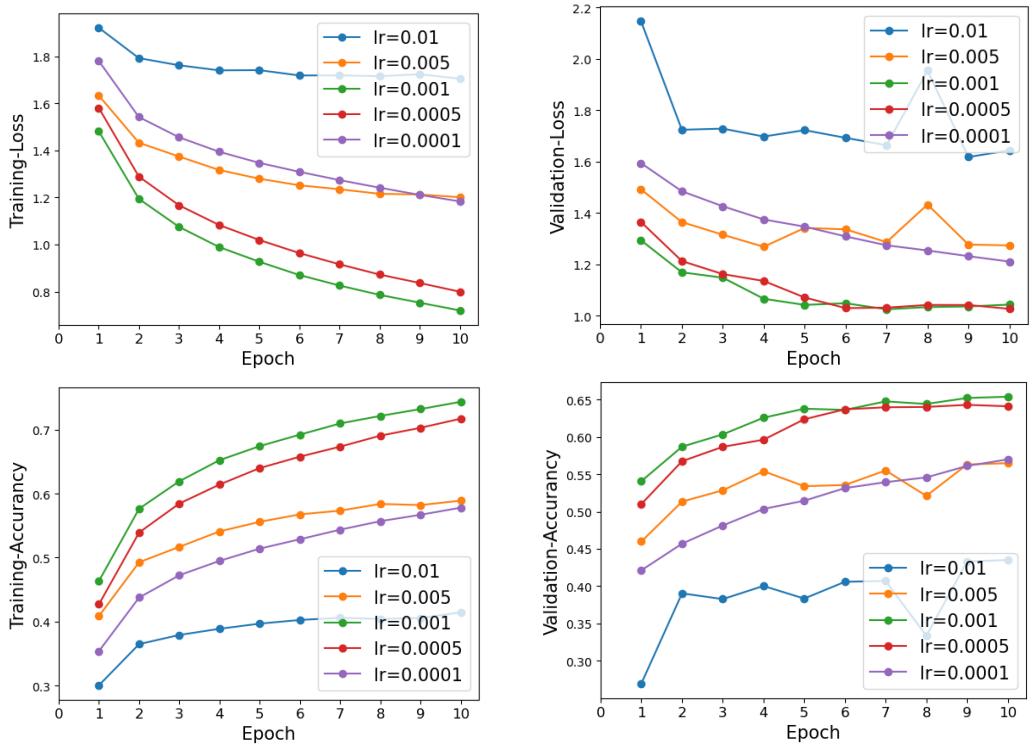


图 9 不同 learning_rate 对比

可以明显看出，learning_rate 较大（0.01,0.005）时，模型表现差且振荡较严重，这是由于学习率太大使得模型参数变化过于剧烈从而不容易收敛在最优点；learning_rate 太小（0.0001）时，模型收敛过慢，训练所需要的时间更长；而 learning_rate 适中（0.001,0.0005）时，模型能较快、较好地收敛。相比之下，learning_rate 取 0.001 时模型表现最好。事实上，不同优化器对应的最佳学习率差别较大，Adam 优化器常使用 0.001 附近的值作为学习率进行训练。

4.4 深度可分离卷积网络

传统卷积网络所需要的参数量较大，每组卷积核参数数目为

$$L_W \times L_H \times in\_channels \times out\_channels$$

Google 公司在 2017 年提出了 MobileNet，其中引入了“深度可分离卷积”的思想。该方法将一个卷积核组分为两个步骤，首先对每个通道分别做卷积，通道数目不变；再用 1×1 的卷积核对各通道做卷积，扩展/缩减通道数目。这样处理后，上述卷积核组参数数目减小到

$$L_W \times L_H \times in\_channels + in\_channels \times out\_channels$$

大大降低了参数数目和计算量，使得卷积网络更容易部署到移动设备上，MobileNet 也因而得名。利用 PyTorch 中 Conv2d 中 groups 参数可以实现这一分组卷积功能，将 3.2 中的网络改为 MobileNet 中深度可分离卷积的结构，其参数量、计算量对比如下：

表 2 轻量级网络与传统网络对比

网络	总参数	卷积部分参数	计算量（FLOPs）
原网络	62006	3083	658025
MobileNet	59504	581	165688
减少量	—[1]	81.2%	74.8%

使用 batch_size=4，SGD 优化器，lr=0.001，momentum=0.9 的优化方法训练，训练过程中 Loss 及 Accurancy 的变化如下：

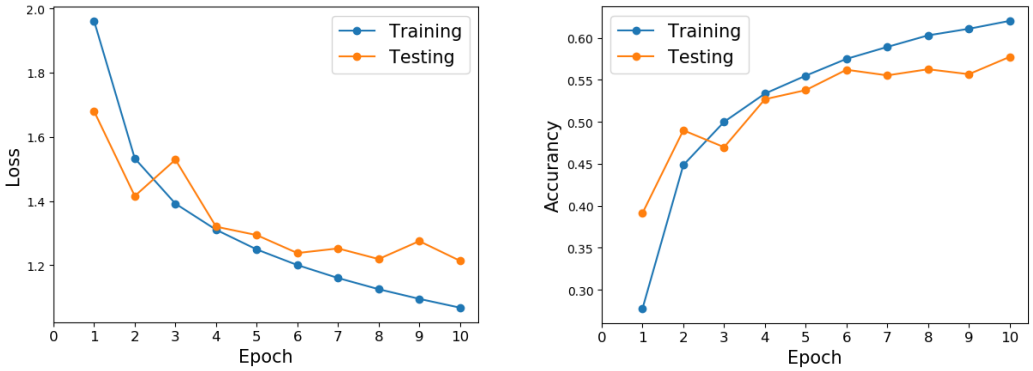


图 10 轻量级网络训练过程

对比 4.1 中原网络的训练，训练后网络分类准确率为 57.7%，比原网络略有下降。在很多应用场景中，网络准确率略微下降对于应用几乎没有影响，而使用轻量级网络带来的计算速度巨大提升有着重大的意义。

5 深度模型应用与最终结果

5.1 模型训练与测试

经过上面的探究，大致可以得到使用较大的 batch_size、Adam 优化器、learning_rate=0.001 能够获得较好的模型表现。下面使用较为成熟的深度卷积神经网络——ResNet 进行最终的模型训练。所提交的 cifar10_cnn.ipynb 文件中为本模型训练结果，前面探究中使用的网络模型附在 Net.py 中。

随着计算机视觉任务的复杂程度不断提升，浅层的神经网络由于参数量有限，难以进行深层次的特征提取，从而表现能力有限。深层神经网络在理论上拥有更大的参数量和更好的表出能力，但由于训练过程中存在不可忽视的梯度消失等问题，深层的神经网络很难得到有效训练。针对这一问题，KaiMing He 提出了在层之间由 shortcut 路径的“残差网络”[2]，使得网络层的学习目标转变为与该层输入的残差。

[1] 该网络深度较小，绝大部分参数来自全连接层，故总参数减少量不明显

[2] Deep Residual Learning for Image Recognition CVPR 2016 Kaiming He et al.

其结构为：

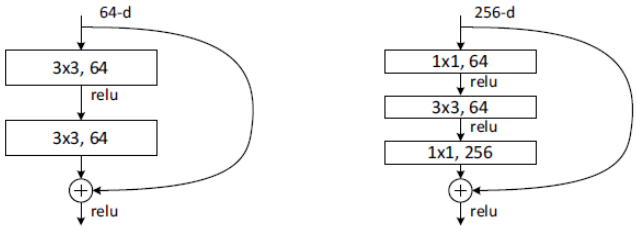


图 11 残差 block 结构

其中包括两个卷积层的残差块将输入值与经过两个 3*3 卷积核的值相加后经过激活函数；包括三个卷积层的残差块首先将输入值利用 1*1 卷积核减少通道数后经过 3*3 卷积核运算，再经过 1*1 卷积核提高通道数与输入匹配后相加经过激活函数。

实际上，PyTorch 中的 torchvision.model.resnet18()支持直接创建 ResNet18 模型，并通过 pretrained 参数选择是否下载并使用预训练模型。此处按照要求，手工编写 ResNet18 的模型结构并进行训练。训练中使用 model = nn.DataParallel(model)，实现在多张 GPU 上并行计算，加快训练速度。训练过程中各项指标变化如下：

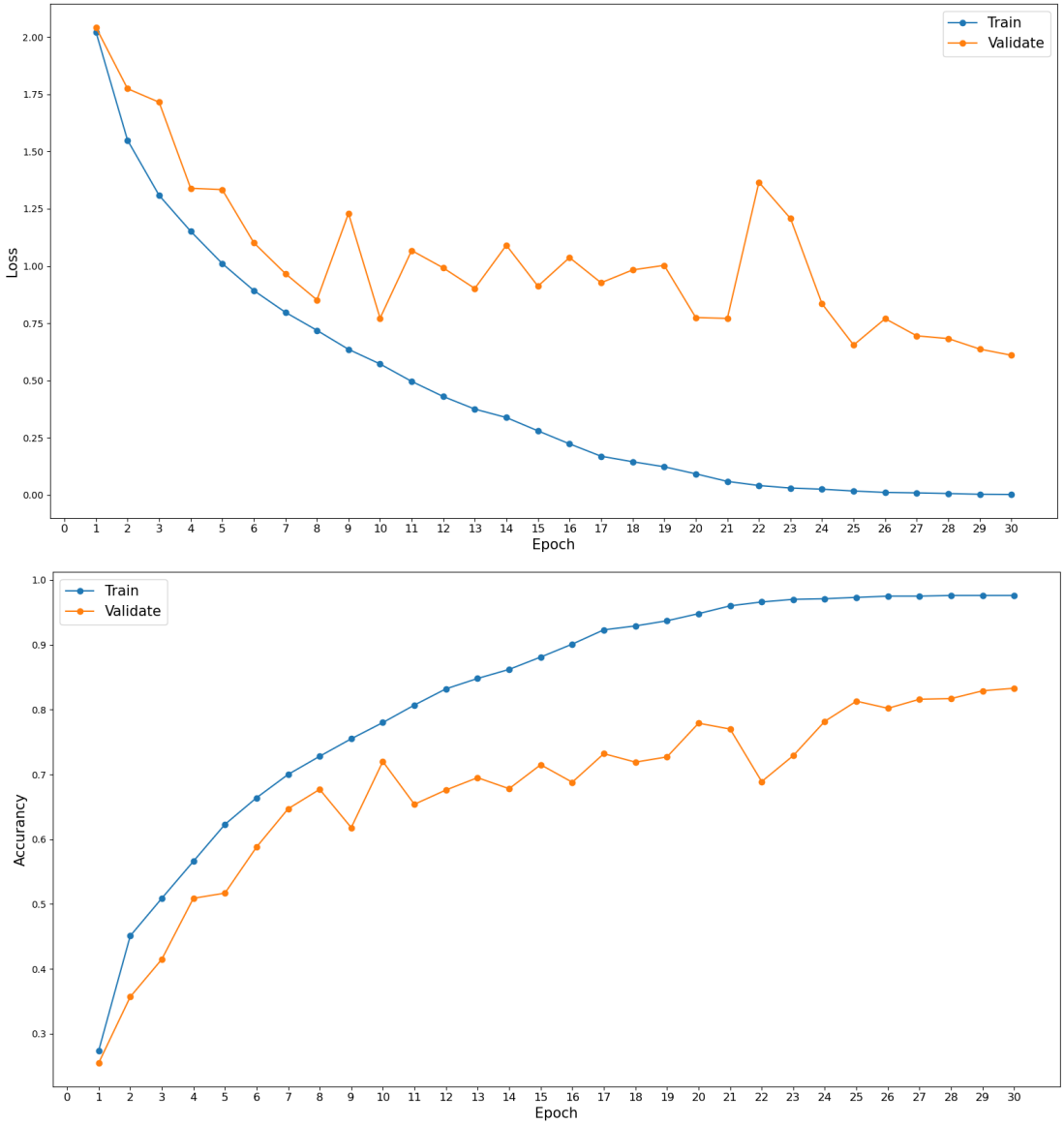


图 12 ResNet18 训练过程

最终测试结果如下：

表 3 ResNet18 最终结果

类别	准确率[3]
plane	0.867
car	0.951
bird	0.816
cat	0.72
deer	0.831
dog	0.751
frog	0.896
horse	0.859
ship	0.926
truck	0.889
overall	**0.851**

5.2 过拟合问题的探究

从训练过程中可以看出，模型出现了一定程度的过拟合，因此尝试在最后一个全连接层加入不同概率的 Dropout，其训练过程验证准确率未加入时对比如下：

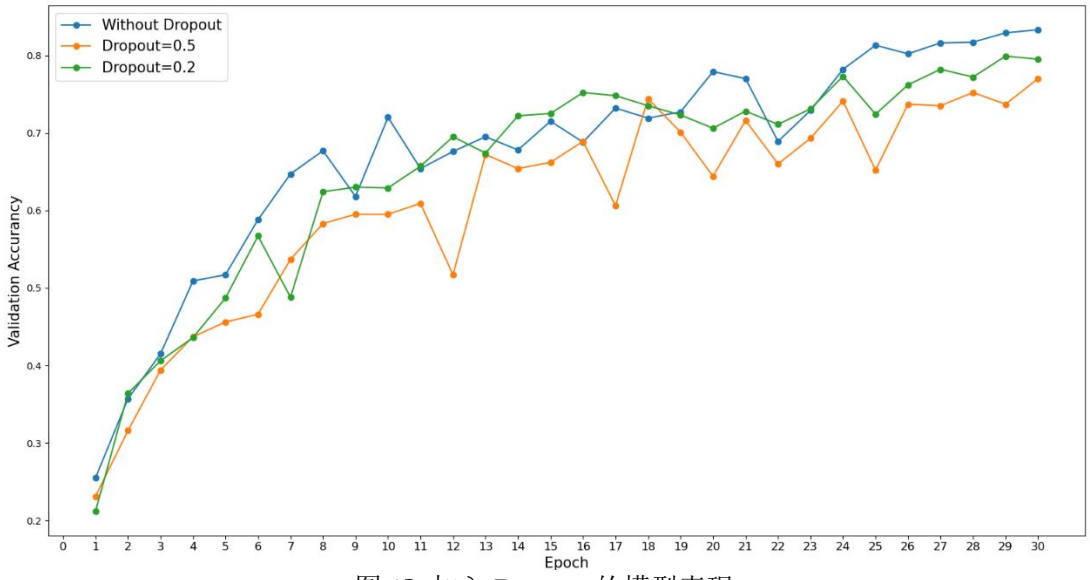


图 13 加入 Dropout 的模型表现

最终测试结果为：

表 4 ResNet18 最终结果

类别	准确率（无 Dropout）	准确率（Dropout=0.5）	准确率（Dropout=0.2）
plane	0.867	0.769	0.887
car	0.951	0.675	0.844
bird	0.816	0.588	0.883
cat	0.72	0.726	0.616
deer	0.831	0.822	0.726
dog	0.751	0.701	0.638
frog	0.896	0.857	0.801
horse	0.859	0.838	0.919
ship	0.926	0.908	0.936
truck	0.889	0.967	0.904
overall	**0.851**	**0.785**	**0.815**

[3] 本文中分类别准确率均指查准率=正确识别数量/该类别样本总数

可以发现，加入 Dropout 后，尽管理论上能够减轻过拟合的问题，但模型表现整体有所下降。推测可能是因此，在一些情况下，并非特别严重的过拟合实际上是可以接受的，加入 Dropout 机制后可能会对模型表现略有影响。

同时发现，当 Dropout 概率被设置为 0.5 时，模型表现下降较多；而 Dropout 概率置为 0.2 时则下降较小。因此可以推测，当模型过拟合严重，确实需要引入 Dropout 机制时，设置合适的 Dropout 概率也是必要的。概率设置太大会使得模型过度稀疏，影响最终表现。

6 总结

本实验中在计算机视觉较常用的小规模数据集 Cifar-10 上训练了多种结构的神经网络，同时探究了不同的损失函数、优化方法、训练超参数对于模型表现的影响。最后在 ResNet18 上得到较好的最终训练结果。分析过程及最终结果，模型尚存在以下问题和可能的改进：

- 观察最终各类别上的分类准确率，类别之间存在较大的差异。因此可以得知，不同类别的物体特征获取、识别的难度不尽相同。通过随机裁剪、翻转、缩放等方式进行数据增强可以获得模型在不同样本上更好的鲁棒性，有可能提升模型表现。
- 较小规模的网络的表出能力有限而难以取得较好的结果，较大规模的网络训练代价又较大。因此选择合适规模的网络是必要的，事实上，神经网络的表出能力也是机器学习理论研究的前沿方向之一。同时，例如深度可分离卷积等方法也提供了神经轻量化的方法。

任务 2：使用 PyTorch 实现 DTP

1 原理介绍

普通神经网络训练过程使用 BP 算法，即用全局损失函数对网络中所有参数求导，对参数进行梯度下降，使得网络不断优化。然而，BP 算法所依赖的偏导数求解要求网络运算以及损失函数必须是连续可微的，这使得这一算法在一些离散或随机的网络中难以被应用。因此，研究者[4]根据神经细胞的作用机理，设计了一种新的神经网络训练方法，即 Difference target propagation (DTP)方法。

该方法中神经网络前向传播过程与 BP 算法中相同，而更新参数的过程则完全不同。更新参数时，网络首先利用反向编码器计算各层应当出现的目标值，再利用目标值与实际输出值之差构建逐层的局部损失函数。用局部损失函数仅对该层参数求导进行梯度下降，如此更新各层梯度，同时更新反向编码器。算法具体流程如下：

Algorithm 1 Training deep neural networks via difference target propagation

Compute unit values for all layers:
for $i = 1$ to M **do**
  $\mathbf{h}_i \leftarrow f_i(\mathbf{h}_{i-1})$
end for
Making the first target: $\hat{\mathbf{h}}_{M-1} \leftarrow \mathbf{h}_{M-1} - \hat{\eta}\frac{\partial L}{\partial \mathbf{h}_{M-1}}$, ($L$ is the global loss)
Compute targets for lower layers:
for $i = M-1$ to 2 **do**
  $\hat{\mathbf{h}}_{i-1} \leftarrow \mathbf{h}_{i-1} - g_i(\mathbf{h}_i) + g_i(\hat{\mathbf{h}}_i)$
end for
Training feedback (inverse) mapping:
for $i = M-1$ to 2 **do**
  Update parameters for g_i using SGD with following a layer-local loss L_i^{inv}
  $L_i^{inv} = ||g_i(f_i(\mathbf{h}_{i-1} + \epsilon)) - (\mathbf{h}_{i-1} + \epsilon)||_2^2, \ \epsilon \sim N(0, \sigma)$
end for
Training feedforward mapping:
for $i = 1$ to M **do**
  Update parameters for f_i using SGD with following a layer-local loss L_i
  $L_i = ||f_i(\mathbf{h}_{i-1}) - \hat{\mathbf{h}}_i||_2^2$ if $i < M$, $L_i = L$ (the global loss) if $i = M$.
end for

该算法可以概括为如下几个主要步骤：

1) 前向计算，将每一层神经网络及其对应的激活函数看作一个广义的函数f_i，得到每个f_i的输出h_i。
2) 计算每一层的目标输出，最上层通过全局损失函数L的导数进行梯度下降得到，其余层通过反向编码函数g_i根据原目标输出h_i和更新后的目标输出 $\hat{h}_i$进行差分运算得到。
3) 计算每层反向编码函数g_i的局部损失函数L_i^{inv}，计算过程为通过f_i、g_i作用于引入了高斯噪声扰动ϵ目标输出h_{i-1}，再利用均方误差函数$MSELoss$得到。利用局部损失函数逐层更新反向编码函数g_i。
4) 计算每层正向函数f_i的局部损失函数L_i，计算过程为利用均方误差函数$MSELoss$计算原输出h_i和目标输出$\hat{h}_i$的均方误差。利用局部损失函数逐层更新正向函数f_i。

2 代码实现

用 PyTorch 实现上述过程，具体代码见提交的 notebook。编写过程中，对于 DTPNet 类，我根据自身对算法的理解，并结合编程习惯对类内成员、方法进行了重构，具体说明如下：

- **类内成员 forward_model、forward_optim、backward_model、backward_optim**，为四个字典。其中 forward_model 中存储了正向计算函数$\{f_i\}$的参数，forward_optim 中存放了对应每个$\{f_i\}$的优化器，本实验中均使用 RMSprop 优化器。backward_model 中存储了反向编码函数$\{g_i\}$的参数，backward_optim 中存放了对应每个$\{g_i\}$的优化器，本实验中均使用 RMSprop 优化器。

[4] Difference Target Propagation Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, Yoshua Bengio

- **类内成员 num_layers**=len(hidden_sizes)-1，根据传入的 list 类型参数 hidden_sizes，确定神经网络的层数。本实验编写的代码中，能够根据传入的参数动态生成网络结构，只需要更改参数 hidden_sizes，即可生成不同的网络结构。
- **类内成员 criterion** = torch.nn.MSELoss()为均方误差函数，用于计算各层的局部损失。
- **成员函数__init__(self,hidden_sizes=None)**，对象构造函数，构造对象时根据传入的 list 类型参数 hidden_sizes 生成网络结构，即生成 forward_model、forward_optim、backward_model、backward_optim 四个字典。
- **成员函数 forward(self, x)**，前向计算，根据输入进行前向计算，得到各层的中间输出和最终的结果。
- **成员函数 compute_target(self, values, global_loss)**，根据得到的各层中间输出值和全局损失函数，更新当前各层的目标输出。
- **成员函数 compute_loss(self, values, targets, global_loss)**，根据得到的各层中间输出值、全局损失函数和当前各层的目标输出，计算各前向计算函数$\{f_i\}$的局部损失函数值。
- **成员函数 compute_loss_inv(self, values)**，根据得到的各层中间输出值，计算各反向编码函数$\{g_i\}$的局部损失函数值。
- **成员函数 backward(self, values, global_loss)**，调用 compute_loss 和 compute_loss_inv 计算得到各层前向计算函数、反向编码函数的局部损失函数值，结合根据得到的各层中间输出值、全局损失函数计算出各层前向计算函数、反向编码函数参数的梯度值。利用梯度下降法逐层更新参数，实现网络的一次优化迭代。
- **成员函数 set_train(self)和 set_eval(self)**，由于类内将模型各层定义在字典中，PyTorch 原装的 train()、eval()方法无法实质性地切换训练状态与测试状态，因此无法在验证、测试时禁用梯度计算功能，造成效率低下。故手动定义这两个函数，调用时将字典中的模型层逐个置为训练状态或测试状态。

在编程实现过程中主要有以下几个问题与注意点：

- 由于本算法中需要单独对每个模型层求梯度，根据 PyTorch 特性，如直接对损失函数值使用.backward()方法，会使计算图中所有变量均被反传累计梯度，达不到希望的效果。因此使用 torch.autograd.grad()方法手动求取梯度，再更新至参数的 weight.grad 和 bias.grad 中。当所有梯度都求出后，依次使用 optimizer.step()方法进行梯度下降。同时注意加参数 retain_graph=True，否则 PyTorch 出于效率考虑，计算图经过一次反传后会被释放，无法多次求取梯度。
- 由于 MLP 的输入为一个向量，因此输入前先用 view()方法将三通道二维图像变为一维列向量。模型一个训练周期流程为：载入数据、数据整形为列向量、forward 正向传播、backward 反向传播。其中 backward 反向传播中又包括：调用 compute_target 计算目标输出、调用 compute_loss_inv 和 compute_loss 计算各层的局部损失函数值、计算梯度并逐层梯度下降优化。

3 性能分析

按照实验指导中给出的建议，使用 3072-1000-1000-1000-10 的 MLP，激活函数为 Tanh 进行 Cifar-10 上的图像分类。训练过程中各项参数变化如下：

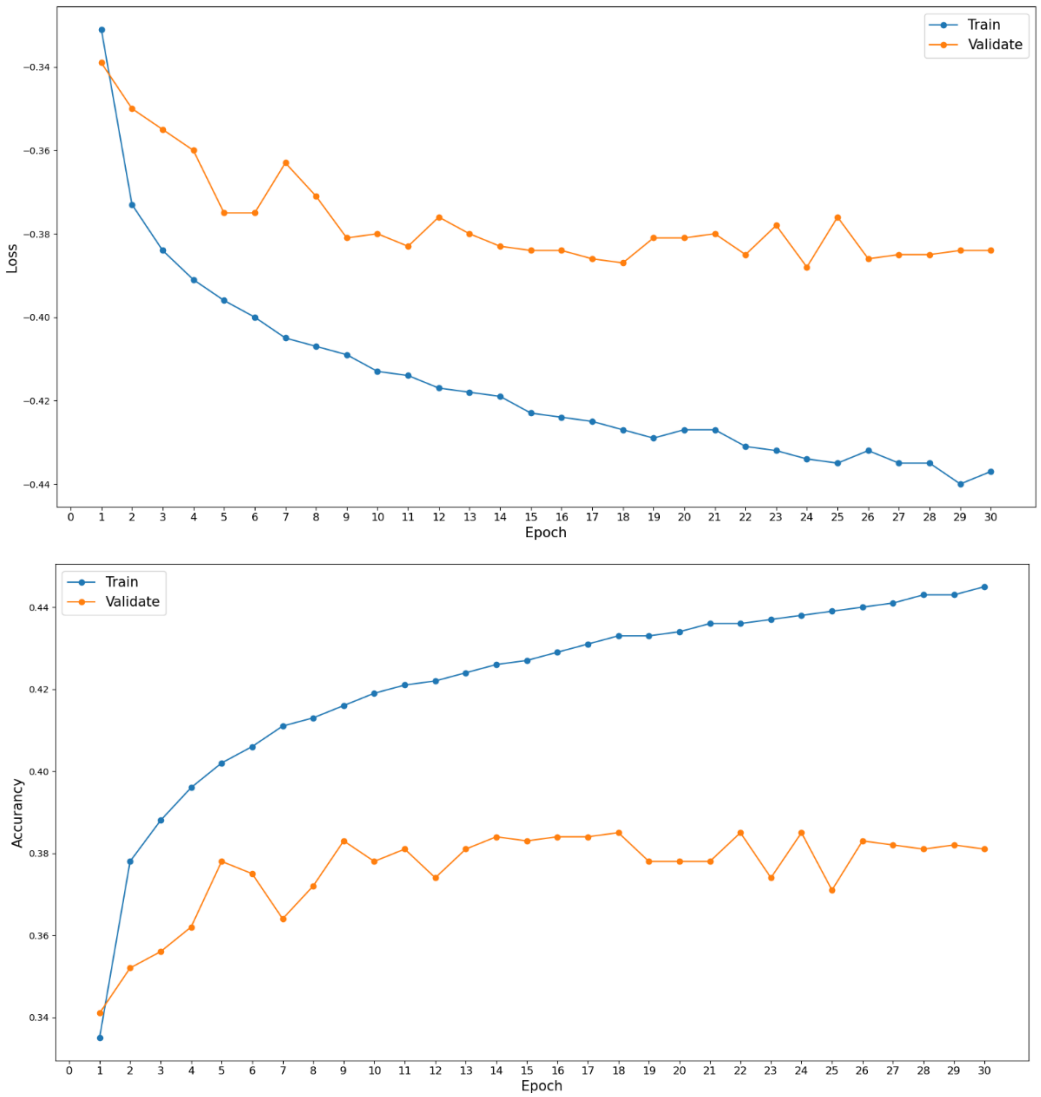


图 1 训练过程各项指标变化

由图中可以看出，该算法能够使得网络收敛，在训练 30 个 epoch 后网络在验证集上的表现基本趋于稳定。最终测试集上的分类结果如下（最终提交的 notebook 中展示的是此结果）：

表 1 最终分类结果

类别	准确率
plane	0.480
car	0.453
bird	0.246
cat	0.171
deer	0.311
dog	0.339
frog	0.464
horse	0.403
ship	0.561
truck	0.496
overall	**0.392**

为对比 DTP 算法与传统的 BP 算法差异，用同样的网络结构，使用 learning_rate=0.001 的 Adam 优化器直接进行梯度反传，相关代码为 BP.py 文件，训练过程中指标对比如下：

图 2 DTP 算法与 BP 算法对比

最终分类结果对比如下：

表 2 最终分类结果对比

类别	准确率（DTP）	准确率（BP）
plane	0.480	0.580
car	0.453	0.523
bird	0.246	0.151
cat	0.171	0.093
deer	0.311	0.357
dog	0.339	0.384
frog	0.464	0.564
horse	0.403	0.459
ship	0.561	0.502
truck	0.496	0.576
overall	**0.392**	**0.419**

由上图可以看出，两种算法均能使神经网络收敛，而收敛速度与最终模型表现均为 BP 算法略好于 DTP。这复现了原论文中所描述的“leading to results comparable to back-propagation”，即 DTP 的表现基本上可以与 BP 算法相同。

同时，由于 DTP 算法需要计算多次中间输出、局部损失，其计算代价和耗时大于 BP 算法，因此在普通的神经网络中，BP 算法更加适用；而在一些特殊的网络中，涉及离散、随机性时，DTP 算法则有较好的性能。

4 总结

本实验实现了一种不同于传统 BP 算法的神经网络训练方法——DTP 算法，同时经过对比验证，复现了原论文中的结果，证明了该算法的可行性。针对本实验有如下可能的问题与改进：

- 本实验只使用了较为简单的 MLP 网络，由于网络结构过于简单，最终分类结果较差，仅用作 DTP 算法是示意。实际上，MLP 网络输入时先将图像展开为列向量，这种操作不易获得图像上邻近位置的信息，因而分类效果较差。使用更加复杂的卷积神经网络应当可以达到更好的效果，将卷积神经网络中的一个卷积层或一个卷积块看做一个广义的前向计算函数$\{f_i\}$即可使用 DTP 算法进行训练。由于任务一中已经验证了 ResNet 的分类表现，此处不再重复实现。
- DTP 算法中涉及较多的超参数选择，如各层局部参数更新的学习率、随机高斯噪声的功率谱密度以及全局损失函数、优化器的选择。在这些超参数上进行进一步调优可能在相同的网络结构上实现更好的测试表现。

文件清单

文件名	说明
实验报告.pdf	本文件
cifar10_cnn.ipynb	任务一代码及结果
cifar10_Target_Propagation.ipynb	任务二代码及结果
Net.py	任务一中探究使用的网络模型
BP.py	任务二中对比 BP 算法
algo.png	notebook 中的图片
model.png	