CVPR 第二次作业　图像分类实验

吴天阳　*4124136039*　人工智能学院 *B2480*

1　实验目的

1. 基于两层神经网络的图像分类器；
2. 学习使用 PyTorch 深度学习框架搭建图像分类器；
3. 学习使用常用 CNN 结构和图像增强技术.

2　实验原理

2.1　全连接网络

全连接网络用于图像分类的基本流程如下：

2.1.1　输入图像

给定一幅输入图像，假设大小为 $H \times W \times C$，其中：

- H 为图像高度（像素数）;
- W 为图像宽度（像素数）;
- C 为通道数（灰度图通道数为 1，RGB 图像通道数为 3）.

将输入图像表示为一个张量 $x \in \mathbb{R}^{H \times W \times C}$.

2.1.2　特征展平

为了输入到全连接层，首先将图像展平成一个一维向量：

$$x_{\text{flat}} = \text{flatten}(x) \in \mathbb{R}^{HWC}.$$

此过程保留了图像的所有像素信息，但丢失了空间结构信息.

2.1.3　全连接层计算

全连接层通过一个权重矩阵 W 和一个偏置向量 $\boldsymbol{b}$ 对输入进行线性变换：

$$\boldsymbol{z} = W x_{\text{flat}} + \boldsymbol{b},$$

其中：

- $W \in \mathbb{R}^{N \times (HWC)}$ 是权重矩阵，N 为神经元的数量.
- $b \in \mathbb{R}^N$ 是偏置向量.
- $z \in \mathbb{R}^N$ 是线性变换的结果.

2.1.4　激活函数

在线性变换之后，通过非线性激活函数（例如 ReLU）引入非线性特性：

$$\boldsymbol{a} = \sigma(\boldsymbol{z}),$$

其中 σ 是激活函数，常用的包括：

- ReLU: $\sigma(x) = \max(0, x)$
- Sigmoid: $\sigma(x) = \dfrac{1}{1+e^{-x}}$
- Mish: $\sigma(x) = x\tanh(\text{softplus}(x)) = x\tanh(\ln(1+e^{x}))$

2.1.5 输出层和分类

输出层通常是另一个全连接层，其输出的维度等于分类任务的类别数 C_{class}：

$$\boldsymbol{y}_{\text{pred}} = \text{softmax}(W_{\text{out}}\boldsymbol{a} + \boldsymbol{b}_{\text{out}}),$$

其中：

- $W_{\text{out}} \in \mathbb{R}^{C_{\text{class}} \times N}$ 为输出层的权重.
- $\boldsymbol{b}_{\text{out}} \in \mathbb{R}^{C_{\text{class}}}$ 为输出层的偏置.
- softmax 将输出变为概率分布：$\text{softmax}(z_i) = \dfrac{e^{z_i}}{\sum_j e^{z_j}}$.

2.1.6 损失函数

使用交叉熵损失函数（Cross-Entropy Loss）来衡量预测概率分布和真实标签的差异：

$$\mathcal{L} = -\sum_{i=1}^{C_{\text{class}}} y_i \log(\hat{y}_i),$$

其中：

- y_i 是真实标签的 one-hot 编码.
- $\hat{y}_i$ 是模型预测的概率分布.

通过梯度下降或其他优化方法更新网络参数，最小化损失函数.

2.1.7 分类结果

最终的分类结果为输出概率中最大值对应的类别：

$$\text{class} = \arg\max_i \boldsymbol{y}_{\text{pred}}.$$

2.2 卷积网络

一个典型的 CNN 模型包括以下几层：

2.2.1 卷积层

卷积层通过卷积核对输入数据进行操作，提取局部特征. 卷积运算公式如下：

$$z_{i,j}^{k} = \sum_{m=1}^{M}\sum_{n=1}^{N} x_{i+m-1,j+n-1} w_{m,n}^{k} + b^{k},$$

其中：

- $x_{i,j}$ 是输入数据.
- $w_{m,n}^{k}$ 是第 k 个卷积核的权重.
- b^{k} 是偏置项.
- $z_{i,j}^{k}$ 是卷积结果.

2.2.2 池化层

池化层用于降维和减少计算量，常用的操作有最大池化和平均池化. 例如，对于最大池化：

$$z_{i,j} = \max_{p,q} x_{i+p,j+q},$$

其中 p, q 是池化窗口的范围.

2.2.3 全连接层

全连接层将前面提取的特征映射到最终的输出空间. 其计算公式为：

$$z = Wx + b,$$

其中 W 是权重矩阵，b 是偏置项.

2.3 ResNet: 深度残差网络

ResNet（Residual Network）[1]通过引入残差块（Residual Block）解决了深层神经网络的梯度消失和退化问题. 其核心思想是学习残差函数 $F(x) := H(x) - x$，其中 $H(x)$ 是目标函数，x 是输入. 网络的基本单元为残差块，数学表达如下：

2.3.1 残差块公式

$$y = F(x, \{W_i\}) + x$$

其中：

- x 是输入特征.
- $F(x, \{W_i\})$ 是通过卷积、批归一化和激活函数计算得到的残差函数：

$$F(x, \{W_i\}) = \sigma(W_2 \cdot \text{BatchNorm}(\sigma(W_1 \cdot x)))$$

- W_1 和 W_2 是卷积核权重.
- $\sigma(\cdot)$ 是激活函数（通常为 ReLU）.

2.3.2 ResNet 结构

ResNet 的总体结构可以表达为：

$$h^{(l+1)} = h^{(l)} + F(h^{(l)}, \{W^{(l)}\})$$

其中：

- $h^{(l)}$ 是第 l 层的特征.
- $F(h^{(l)}, \{W^{(l)}\})$ 是第 l 层的残差函数.

2.3.3 网络深度

ResNet-18、ResNet-34 等通过堆叠多个残差块构建，网络的层数为：

$$\text{总层数} = 2 \times \text{残差块数} + \text{输入输出层数}$$

[1] https://arxiv.org/pdf/1512.03385

2.3.4 瓶颈块（Bottleneck Block）

对于深层网络（如 ResNet-50、ResNet-101），使用瓶颈块减少参数量：

$$F(x) = W_3 \cdot \sigma(W_2 \cdot \text{BatchNorm}(\sigma(W_1 \cdot x)))$$

瓶颈块中：

- W_1: 1x1 卷积，用于降维.
- W_2: 3x3 卷积，用于特征提取.
- W_3: 1x1 卷积，用于升维.

2.4 Wide Residual Network (WideResNet)

Wide Residual Network (WideResNet)[2] 是 ResNet 的一种变体，旨在通过增加网络宽度而不是深度来提高模型性能，同时降低训练复杂度. WideResNet 的主要公式如下：

$$x_{l+1} = x_l + \mathcal{F}(x_l, \{W_{l,i}\}), \tag{2.1}$$

其中 $\mathcal{F}$ 表示残差映射（Residual Mapping），$\{W_{l,i}\}$ 表示第 l 层的可学习参数集合. WideResNet 通过引入一个宽度因子 k 来增加每层的通道数，即将 ResNet 的每层特征图数量扩展为 k 倍.

2.4.1 WideResNet 与 ResNet 的区别

WideResNet 在设计上主要与 ResNet 有以下区别：

- **宽度 vs. 深度**: ResNet 通过增加网络的深度来增强学习能力，但过深的网络可能带来梯度消失或过拟合问题. WideResNet 则通过增加每层的通道数（宽度因子 k）来提高网络的容量，同时避免了过深网络的训练问题.
- **简化的 Bottleneck 结构**: ResNet 在深层通常使用 Bottleneck 结构（$1 \times 1, 3 \times 3, 1 \times 1$ 的卷积），而 WideResNet 直接使用标准卷积层，从而简化了模型.
- **减少深度**: WideResNet 显著降低了模型深度，使得训练时间大幅减少. 例如，WideResNet-28-10 指的是一个深度为 28，宽度因子为 10 的 WideResNet.

2.4.2 CIFAR-10 上使用 WideResNet

在 WideResNet 的原始论文中，WideResNet-28-10 在 CIFAR-10 数据集上达到了接近最优的性能，其错误率低至 3.8%，显著优于传统的 ResNet.

CIFAR-10 数据集是一个由 10 类小图像组成的数据集，每张图像的尺寸为 32×32 像素. WideResNet 在 CIFAR-10 上表现出色的原因包括：

- **高效的特征表示**: 对于小尺寸图像，增加网络的宽度（而非深度）可以更有效地捕获局部和全局特征，从而提升分类性能.
- **优化效率**: CIFAR-10 数据集的样本较小，WideResNet 的浅层网络结构能够显著减少训练时间，同时保持或超越深度网络的准确率.
- **防止过拟合**: 在 CIFAR-10 这样的小型数据集上，过深的网络容易过拟合，而 WideResNet 的较浅结构通过宽度扩展减少了这种风险.

[2] https://arxiv.org/pdf/1605.07146

2.5 AutoAugmentPolicy 数据增强

AutoAugment[3] 使用概率性数据增强策略提高模型的泛化能力. 以下为 CIFAR-10 策略中的数据增强方法和数学描述.

2.5.1 数据增强方法列表

1. **Invert**: 将图像颜色取反.
$$I'(x,y) = 255 - I(x,y)$$
其中 $I(x,y)$ 是原始像素值，$I'(x,y)$ 是增强后的像素值.
2. **Contrast**: 调整对比度. 令图像的平均亮度为 μ，对比度调整公式为：
$$I'(x,y) = \mu + \alpha \cdot (I(x,y) - \mu)$$
其中 α 为增强强度.
3. **Rotate**: 图像旋转 θ 度：
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$
4. **TranslateX / TranslateY**: 在水平 / 垂直方向上平移 Δx 或 Δy 像素：
$$I'(x + \Delta x, y + \Delta y) = I(x,y)$$
5. **Sharpness**: 调整图像锐化程度，使用卷积核增强边缘信息.
6. **ShearX / ShearY**: 图像沿 X / Y 轴剪切：
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & \lambda_x \\ \lambda_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$
其中 λ_x, λ_y 为剪切强度.
7. **AutoContrast**: 自动调整对比度，使像素值覆盖整个动态范围.
8. **Equalize**: 直方图均衡化：
$$H'(v) = \left\lfloor \frac{\mathrm{CDF}(v) - \mathrm{CDF}_{\min}}{1 - \mathrm{CDF}_{\min}} \cdot (L-1) \right\rfloor$$
其中 $\mathrm{CDF}(v)$ 是像素值 v 的累计分布函数.
9. **Posterize**: 降低图像的色深：
$$I'(x,y) = \lfloor \frac{I(x,y)}{2^b} \rfloor \cdot 2^b$$
其中 b 为剩余位数.
10. **Color**: 调整颜色饱和度.
11. **Brightness**: 调整亮度：
$$I'(x,y) = I(x,y) + \beta$$
12. **Solarize**: 反转高于某阈值 T 的像素值：
$$I'(x,y) = \begin{cases} I(x,y) & \text{if } I(x,y) < T \\ 255 - I(x,y) & \text{if } I(x,y) \geqslant T \end{cases}$$

[3] http://arxiv.org/pdf/1805.09501

3 实验步骤与结果分析

3.1 在 cifar10 上用 PyTorch 训练两层神经网络分类器

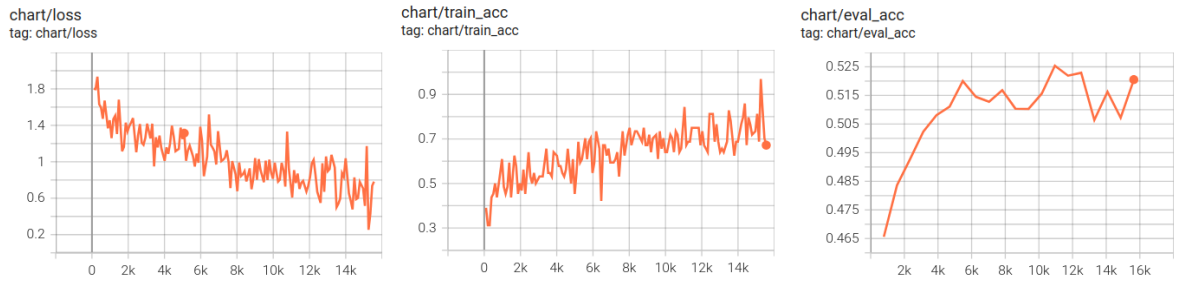
训练流程为，定义超参数、神经网络，读取数据集，划分数据集为训练集与验证集，实例化模型、优化器、损失函数，开始训练，在验证集上验证模型性能，保存模型，具体代码如下：

```
import time
from pathlib import Path

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard.writer import SummaryWriter

# Tensorbaord 日志
path_log = Path(f"./logs/{time.strftime('%Y%m%d-%H%M%S')}")
writer = SummaryWriter(path_log)

# 超参数设置
batch_size = 64
learning_rate = 0.001
num_epochs = 20
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 数据加载和预处理
transform = transforms.Compose([
  transforms.ToTensor(),        # 转换为 Tensor
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # 标准化到 [-1, 1]
])

train_dataset = datasets.CIFAR10(root='./data', train=True,
↪  transform=transform, download=True)
test_dataset = datasets.CIFAR10(root='./data', train=False,
↪  transform=transform, download=True)

train_loader = DataLoader(train_dataset, batch_size=batch_size,
↪  shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
↪  shuffle=False)

# 定义全连接神经网络
class FullyConnectedNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(FullyConnectedNN, self).__init__()
    self.fc1 = nn.Linear(input_size, hidden_size)  # 输入到隐藏层
    self.relu = nn.ReLU()                # 激活函数
    self.fc2 = nn.Linear(hidden_size, num_classes)  # 隐藏层到输出层

```

```
  def forward(self, x):
    x = x.view(x.size(0), -1)  # 展平
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    return x

# 模型实例化
input_size = 32 * 32 * 3  # CIFAR-10 图像大小 (32x32x3)
hidden_size = 256     # 隐藏层神经元数
num_classes = 10      # CIFAR-10 分类数
model = FullyConnectedNN(input_size, hidden_size, num_classes).to(device)

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
global_step = 0
best_eval_acc = 0

# 训练模型
for epoch in range(num_epochs):
  model.train()
  for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    # 前向传播
    outputs = model(data)
    loss = criterion(outputs, target)
    _, predicted = torch.max(outputs, 1)
    acc = (target == predicted).sum().item() / batch_size

    # 反向传播
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    global_step += 1

    if (batch_idx + 1) % 100 == 0:
      writer.add_scalar("chart/loss", loss.item(), global_step)
      writer.add_scalar("chart/train_acc", acc, global_step)
      print(f"Epoch [{epoch + 1}/{num_epochs}], Step [{batch_idx +
      ↪  1}/{len(train_loader)}], Loss: {loss.item():.4f}, Acc:
      ↪  {acc:.4f}")

  # 测试模型
  model.eval()
  correct = 0
  total = 0
  with torch.no_grad():
    for data, target in test_loader:
      data, target = data.to(device), target.to(device)
      outputs = model(data)
```

```
        _, predicted = torch.max(outputs.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

    eval_acc = correct / total
    print(f"Test Accuracy: {100 * eval_acc:.2f}%")
    writer.add_scalar("chart/eval_acc", eval_acc, global_step)

    if eval_acc > best_eval_acc:
      best_eval_acc = eval_acc
      # 保存最优 eval 模型
      path_save_model = f"cifar10_fc_model_best_eval.pth"
      torch.save(model.state_dict(), path_log / path_save_model)
      print(f"Best eval model ({100*eval_acc:.2f}%) saved as {path_log /
      ↪  path_save_model}")


# 保存模型
path_save_model = f"cifar10_fc_model_{global_step}.pth"
torch.save(model.state_dict(), path_log / path_save_model)
print(f"Last model saved as {path_log / path_save_model}")
print(f"Best eval accuracy {100 * best_eval_acc:.2f}%")
```

TensorBoard 日志图片如下：



(a) 损失函数图像　(b) 训练中准确率图像　(c) 训练中准确率图像

图 1: 训练 20 个 epochs 的 TensorBoard 日志图像，在验证集上的最优准确率为第 11 个 epoch 时的 52.54%

3.2 在 cifar10 上用 PyTorch 训练卷积网络分类器

与全连接神经网络不同之处在于：

1. 使用了图像增强，包括随机裁剪，随机水平翻转，色彩抖动；
2. 三个 CNN 卷积块（2D 卷积，批归一化，Mish 激活函数），每个卷积块后经过一个最大池化将图像缩小一倍，最后展平，用全连接做输出头预测类别.

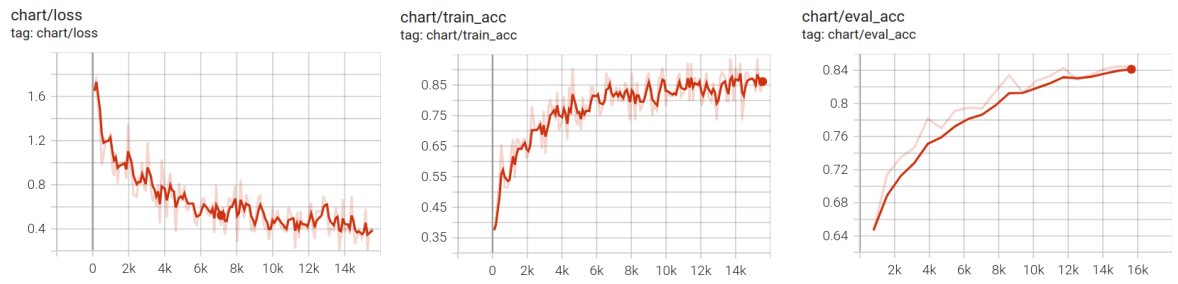
```
import time
from pathlib import Path

import torch
import torch.nn as nn
```

```python
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard.writer import SummaryWriter

# Tensorbaord 日志
path_log = Path(f"./logs/{time.strftime('%Y%m%d-%H%M%S')}")
writer = SummaryWriter(path_log)

# 检查是否有可用 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# 超参数设置
batch_size = 64
learning_rate = 0.001
num_epochs = 20

# 数据增强和预处理
transform_train = transforms.Compose([
  transforms.RandomCrop(32, padding=4),       # 随机裁剪
  transforms.RandomHorizontalFlip(),      # 随机水平翻转
  transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # 色彩抖动
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # 标准化
])

transform_test = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_dataset = datasets.CIFAR10(root='./data', train=True, transform=transform_train, download=True)
test_dataset = datasets.CIFAR10(root='./data', train=False, transform=transform_test, download=True)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

class CNN(nn.Module):
  def __init__(self, in_ch, out_ch, kernel, stride, padding):
    super().__init__()
    self.conv = nn.Conv2d(in_ch, out_ch, kernel_size=kernel, stride=stride, padding=padding)
    self.bn = nn.BatchNorm2d(out_ch)
    self.mish = nn.Mish()

```

```
  def forward(self, x):
    return self.mish(self.bn(self.conv(x)))

# 定义 CNN 模型
class Model(nn.Module):
  def __init__(self, num_classes=10):
    super().__init__()
    self.backbone = nn.Sequential(
      CNN(3, 64, 3, 1, 1),
      nn.MaxPool2d(kernel_size=2, stride=2),
      CNN(64, 128, 3, 1, 1),
      nn.MaxPool2d(kernel_size=2, stride=2),
      CNN(128, 256, 3, 1, 1),
      nn.MaxPool2d(kernel_size=2, stride=2),
    )
    self.head = nn.Sequential(
      nn.Linear(256 * 4 * 4, 512),
      nn.Mish(),
      nn.Linear(512, num_classes),
    )

  def forward(self, x):
    x = self.backbone(x)
    x = nn.Flatten()(x)
    x = self.head(x)
    return x

# 初始化模型、损失函数和优化器
model = Model().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
global_step = 0
best_eval_acc = 0

# 训练模型
for epoch in range(num_epochs):
  model.train()
  for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)

    # 前向传播
    outputs = model(data)
    loss = criterion(outputs, target)
    _, predicted = torch.max(outputs, 1)
    acc = (target == predicted).sum().item() / batch_size

    # 反向传播
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    global_step += 1
```

```
102
103     if (batch_idx + 1) % 100 == 0:
104       writer.add_scalar("chart/loss", loss.item(), global_step)
105       writer.add_scalar("chart/train_acc", acc, global_step)
106       print(f"Epoch [{epoch + 1}/{num_epochs}], Step [{batch_idx +
          ↪  1}/{len(train_loader)}], Loss: {loss.item():.4f}, Acc:
          ↪  {acc:.4f}")
107
108   # 测试模型
109   model.eval()
110   correct = 0
111   total = 0
112   with torch.no_grad():
113     for data, target in test_loader:
114       data, target = data.to(device), target.to(device)
115       outputs = model(data)
116       _, predicted = torch.max(outputs.data, 1)
117       total += target.size(0)
118       correct += (predicted == target).sum().item()
119
120   eval_acc = correct / total
121   print(f"Test Accuracy: {100 * eval_acc:.2f}%")
122   writer.add_scalar("chart/eval_acc", eval_acc, global_step)
123
124   if eval_acc > best_eval_acc:
125     best_eval_acc = eval_acc
126     # 保存最优 eval 模型
127     path_save_model = f"cifar10_fc_model_best_eval.pth"
128     torch.save(model.state_dict(), path_log / path_save_model)
129     print(f"Best eval model ({100*eval_acc:.2f}%) saved as {path_log /
        ↪  path_save_model}")
130
131 # 保存模型
132 path_save_model = f"cifar10_fc_model_{global_step}.pth"
133 torch.save(model.state_dict(), path_log / path_save_model)
134 print(f"Last model saved as {path_log / path_save_model}")
135 print(f"Best eval accuracy {100 * best_eval_acc:.2f}%")
```

TensorBoard 日志图片如下：



(a) 损失函数图像　(b) 训练中准确率图像　(c) 训练中准确率图像

图 2: 训练 20 个 epochs 的 TensorBoard 日志图像，在验证集上的最优准确率为第 19 个 epoch 时的 84.48%

3.3 在 cifar10 上用 WideResNet+AutoAugment 训练

这一次，我参考了 GitHub 上pytorch-auto-augment仓库的代码，使用了 WideResNet 和 AutoAugment 进行训练，并加了 L2 范数正则项，和余弦学习率调整，训练 120 个 epoch，在验证集上最好能达到 96.44% 的准确率.

首先演示 Augment 数据增强效果图：

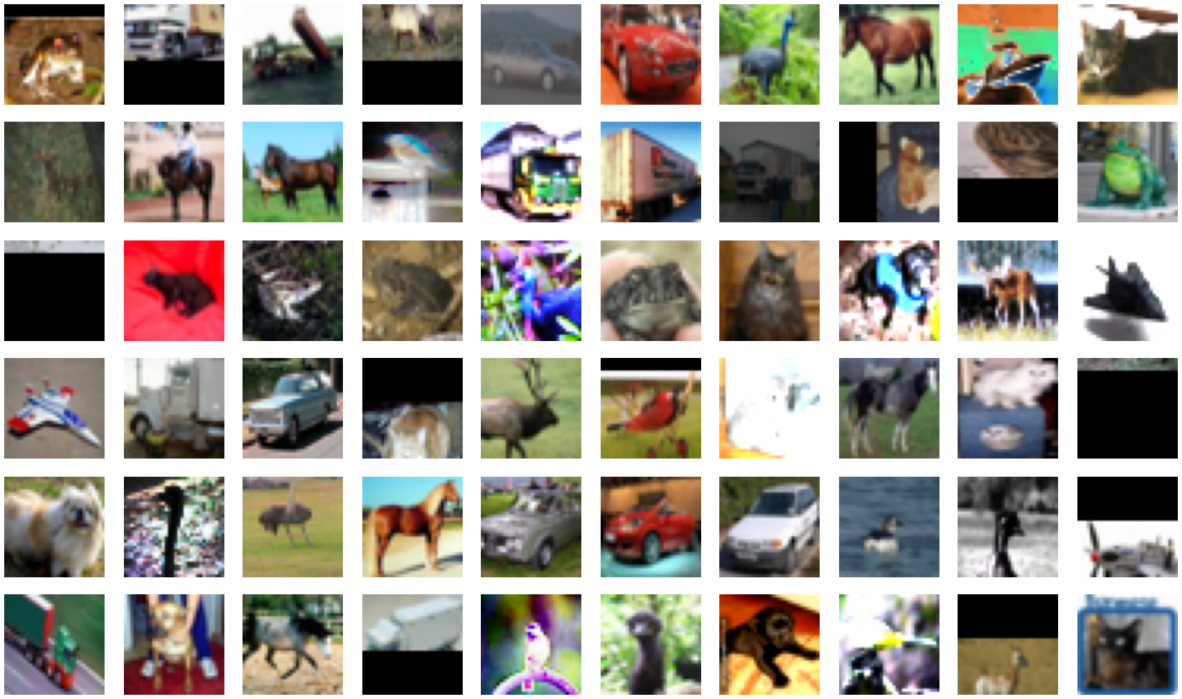


图 3: 训练集上不同图片增强后的效果

演示验证集上单张图片随机增强后的效果图：

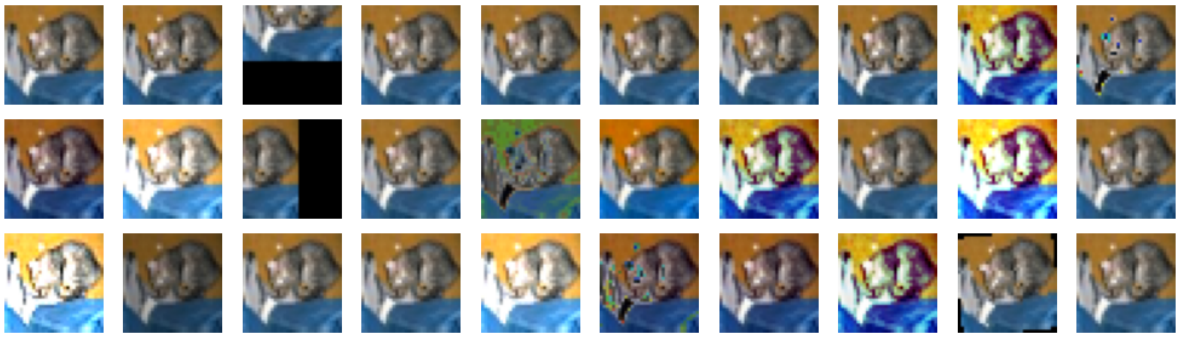


图 4: 验证集上同一张图片增强后的效果，左上角为原始图片

代码非常简单，可以直接使用如下代码构建图像变换函数，从而作用在数据集上，最后用 matplotlib 可视化变换效果即可：

```
from torchvision.transforms.autoaugment import AutoAugment,
↪  AutoAugmentPolicy

transform = AutoAugment(AutoAugmentPolicy.CIFAR10)  # 创建变换

```

```
# 作用于数据集上
train_dataset = datasets.CIFAR10(root='./data', train=True,
↪  transform=transform, download=True)

def grid_augment():
  train_iter = iter(train_dataset)
  r = 6
  c = 10
  figure, axs = plt.subplots(r, c, figsize=(10, 6))

  for i in range(r):
    for j in range(c):
      img = next(train_iter)[0]
      ax: Axes = axs[i,j]
      ax.set_axis_off()
      ax.imshow(img)
  plt.tight_layout()
  plt.savefig(path_figures / "grid_augment.png", dpi=100)
  plt.show()
```

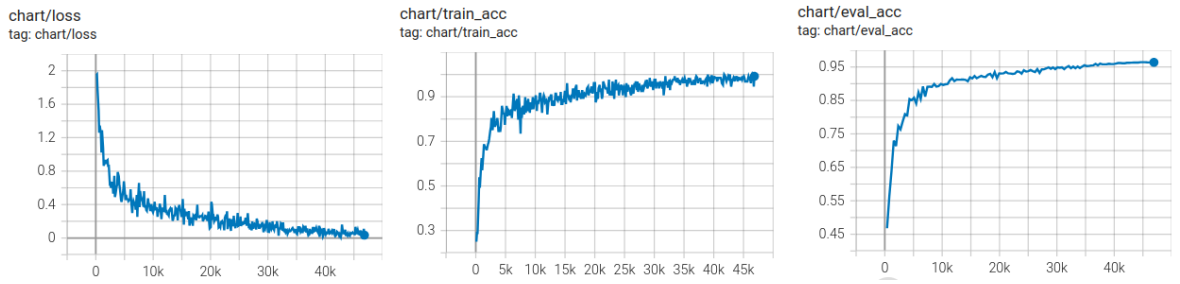
学会数据增强后，类似之前训练模型的方法，只需简单修改即可得到训练代码：

```
import time
from pathlib import Path

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard.writer import SummaryWriter
from torchvision.transforms.autoaugment import AutoAugment,
↪  AutoAugmentPolicy
from wide_resnet import WideResNet

# Tensorbaord 日志
path_log = Path(f"./logs/{time.strftime('%Y%m%d-%H%M%S')}-wide-resnet")
writer = SummaryWriter(path_log)

# 检查是否有可用 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# 超参数设置
batch_size = 128
learning_rate = 0.1
momentum = 0.9
weight_decay = 1e-4
num_epochs = 120
autoaugment = True

# 数据增强和预处理
```

```
transform_train = transforms.Compose([
  transforms.RandomCrop(32, padding=4),
  transforms.RandomHorizontalFlip(),
  *([AutoAugment(AutoAugmentPolicy.CIFAR10)] if autoaugment else []),
  transforms.ToTensor(),
  transforms.Normalize((0.4914, 0.4822, 0.4465),
                       (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.5071, 0.4867, 0.4408),
                        (0.2675, 0.2565, 0.2761)),
])

train_dataset = datasets.CIFAR10(root='./data', train=True,
↪  transform=transform_train, download=True)
test_dataset = datasets.CIFAR10(root='./data', train=False,
↪  transform=transform_test, download=True)

train_loader = DataLoader(train_dataset, batch_size=batch_size,
↪  shuffle=True, num_workers=8)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
↪  shuffle=False, num_workers=8)

# 初始化模型、损失函数和优化器
model = WideResNet(depth=28, width=10, num_classes=10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9,
↪  weight_decay=weight_decay)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
↪  len(train_loader) * num_epochs)
# scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [60, 120,
↪  160], 0.2)
global_step = 0
best_eval_acc = 0

# 训练模型
for epoch in range(num_epochs):
  model.train()
  for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)

    # 前向传播
    outputs = model(data)
    loss = criterion(outputs, target)
    _, predicted = torch.max(outputs, 1)
    acc = (target == predicted).sum().item() / batch_size

    # 反向传播
    optimizer.zero_grad()
```

```
    loss.backward()
    optimizer.step()
    scheduler.step()
    global_step += 1

    if (batch_idx + 1) % 100 == 0:
      writer.add_scalar("chart/loss", loss.item(), global_step)
      writer.add_scalar("chart/train_acc", acc, global_step)
      writer.add_scalar("chart/learning_rate", scheduler.get_last_lr()[0],
      ↪  global_step)
      print(f"Epoch [{epoch + 1}/{num_epochs}], Step [{batch_idx +
      ↪  1}/{len(train_loader)}], Loss: {loss.item():.4f}, Acc:
      ↪  {acc:.4f}")

  # scheduler.step()

  # 测试模型
  model.eval()
  correct = 0
  total = 0
  with torch.no_grad():
    for data, target in test_loader:
      data, target = data.to(device), target.to(device)
      outputs = model(data)
      _, predicted = torch.max(outputs.data, 1)
      total += target.size(0)
      correct += (predicted == target).sum().item()

  eval_acc = correct / total
  print(f"Test Accuracy: {100 * eval_acc:.2f}%")
  writer.add_scalar("chart/eval_acc", eval_acc, global_step)

  if eval_acc > best_eval_acc:
    best_eval_acc = eval_acc
    # 保存最优 eval 模型
    path_save_model = f"cifar10_wide_resnet_model_best_eval.pth"
    torch.save(model.state_dict(), path_log / path_save_model)
    print(f"Best eval model ({100*eval_acc:.2f}%) saved as {path_log /
    ↪  path_save_model}")

# 保存模型
path_save_model = f"cifar10_wide_resnet_model_{global_step}.pth"
torch.save(model.state_dict(), path_log / path_save_model)
print(f"Last model saved as {path_log / path_save_model}")
print(f"Best eval accuracy {100 * best_eval_acc:.2f}%")
```

TensorBoard 日志图片如下



(a) 损失函数图像 (b) 训练中准确率图像 (c) 训练中准确率图像

图 5: 训练 120 个 epochs 的 TensorBoard 日志图像，在验证集上的最优准确率为第 116 个 epoch 时的 96.44%

4 总结

表 1: PyTorch 在 CIFAR10 数据集上模型性能对比表

网络模型	学习率调整	L2 正则	数据增强	总 epoch	验证集最优 epoch	验证集最优准确率	训练集最优准确率
两层全连接	无	无	无	20	11	52.54%	85.94%
3 层卷积网络	无	无	裁剪、水平翻转、色彩抖动	20	19	84.48%	93.75%
WidResNet	两段变换	10^{-4}	无	120	71	94.91%	100%
WidResNet	Cosine	10^{-4}	无	120	115	94.86%	100%
WidResNet	Cosine	10^{-4}	Auto-Augment	120	116	**96.44%**	99.22%

代码使用说明，上述三个报告分别对应代码`1_relu.py`,`2_mish.py`,`3_wide_resnet.py`，直接执行代码即可开始训练，TensorBaord 查看方式：

```
# 安装 tensorboard
pip install tensorboard
# 进入终端执行
tensorboard --logdir ./logs
```

软件说明 使用的 Python 版本为 3.11（任何 3.8 以上版本均可），PyTorch 版本为 2.5.1-cuda（任何 PyTorch 版本均可）.

硬件说明 使用的 CPU 为 AMD 5700X，显卡为 RTX-4080，训练上述三个模型分别用时 1m27s，5m23s，2h11m58s.