Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №1

По дисциплине: «ОИвИС»

Тема: «Обучение классификаторов средствами библиотеки PyTorch»

Выполнил:

Студент 4 курса

Группы ИИ-23

Романюк А. П.

Проверила:

Андренко К.В.

Брест 2025

Цель: научиться конструировать нейросетевые классификаторы и выполнять их обучение на известных выборках компьютерного зрения

Общее задание

1. Выполнить конструирование своей модели СНС, обучить ее на выборке по заданию (использовать torchvision.datasets). Предпочтение отдавать как можно более простым архитектурам, базирующимся на базовых типах слоев (сверточный, полносвязный, подвыборочный, слой нелинейного преобразования). Оценить эффективность обучения на тестовой выборке, построить график изменения ошибки (matplotlib);

2. Ознакомьтесь с state-of-the-art результатами для предлагаемых выборок (из материалов в сети Интернет). Сделать выводы о результатах обучения СНС из п. 1;

3. Реализовать визуализацию работы СНС из пункта 1 (выбор и подачу на архитектуру произвольного изображения с выводом результата);

4. Оформить отчет по выполненной работе, загрузить исходный код и отчет в соответствующий репозиторий на github.

Вариант:

9	CIFAR-100	32X32	Adam

Код программы:

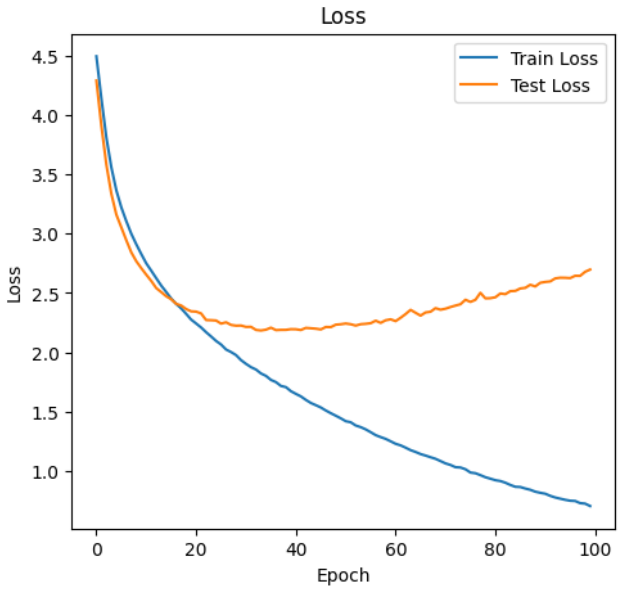
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm

train_transform = transforms.Compose([
  transforms.RandomHorizontalFlip(),
  transforms.RandomRotation(10),
  transforms.ToTensor(),
  transforms.Normalize((0.5071, 0.4865, 0.4409), (0.2673, 0.2564, 0.2761))
])

test_transform = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.5071, 0.4865, 0.4409), (0.2673, 0.2564, 0.2761))
])

train_loader = torch.utils.data.DataLoader(
  torchvision.datasets.CIFAR100('/files/', train=True, download=True,
                    transform=train_transform),
  batch_size=128, shuffle=True)
```

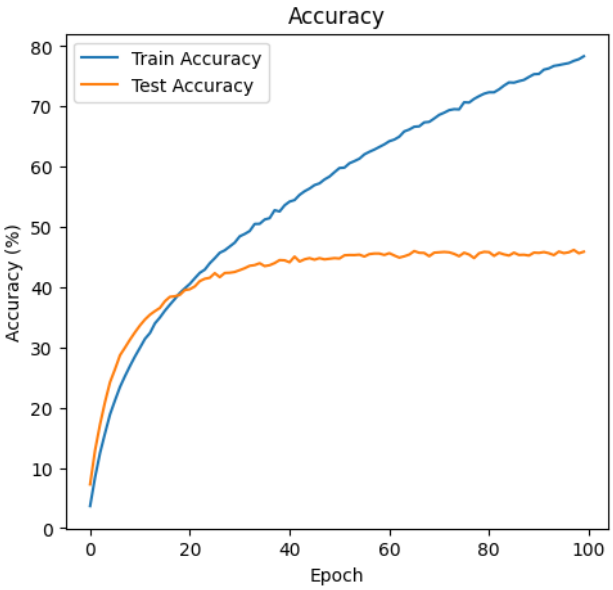
```
test_loader = torch.utils.data.DataLoader(
   torchvision.datasets.CIFAR100('/files/', train=False, download=True,
                           transform=test_transform),
   batch_size=256, shuffle=False)

class NN(nn.Module):
   def __init__(self):
      super(NN, self).__init__()
      self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
      self.bn1 = nn.BatchNorm2d(32)
      self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
      self.bn2 = nn.BatchNorm2d(64)
      self.pool = nn.MaxPool2d(2, 2)

      self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
      self.bn3 = nn.BatchNorm2d(128)
      self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
      self.bn4 = nn.BatchNorm2d(256)

      self.conv5 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
      self.bn5 = nn.BatchNorm2d(512)
      self.conv6 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
      self.bn6 = nn.BatchNorm2d(512)

      self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))

      self.fc1 = nn.Linear(512, 256)
      self.fc2 = nn.Linear(256, 128)
      self.fc3 = nn.Linear(128, 100)
      self.relu = nn.LeakyReLU()
      self.dropout4 = nn.Dropout(0.5)

   def forward(self, x):
      x = self.pool(self.relu(self.bn1(self.conv1(x))))
      x = self.pool(self.relu(self.bn2(self.conv2(x))))

      x = self.pool(self.relu(self.bn3(self.conv3(x))))
      x = self.pool(self.relu(self.bn4(self.conv4(x))))

      x = self.pool(self.relu(self.bn5(self.conv5(x))))
      x = self.relu(self.bn6(self.conv6(x)))

      x = self.global_avg_pool(x)
      x = x.view(-1, 512)

      x = self.relu(self.fc1(x))
      x = self.dropout4(x)
      x = self.relu(self.fc2(x))
      x = self.fc3(x)
      return x
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = NN().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.00003)


def train(model, loader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct, total = 0, 0

    progress = tqdm(loader, desc="Training", leave=False)
    for images, labels in progress:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

        progress.set_postfix(loss=loss.item())

    accuracy = 100 * correct / total
    return running_loss / len(loader), accuracy


def test(model, loader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct, total = 0, 0

    progress = tqdm(loader, desc="Testing", leave=False)
    with torch.no_grad():
        for images, labels in progress:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            running_loss += loss.item()

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

            progress.set_postfix(loss=loss.item())
```

```
    accuracy = 100 * correct / total
    return running_loss / len(loader), accuracy

train_losses, test_losses = [], []
train_accuracies, test_accuracies = [], []

num_epochs = 100
for epoch in range(num_epochs):
    print(f"\nEpoch {epoch+1}/{num_epochs}")

    train_loss, train_accuracy = train(model, train_loader, criterion, optimizer, device)
    test_loss, test_accuracy = test(model, test_loader, criterion, device)

    train_losses.append(train_loss)
    test_losses.append(test_loss)
    train_accuracies.append(train_accuracy)
    test_accuracies.append(test_accuracy)

    print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_accuracy:.2f}% | "
          f"Test Loss: {test_loss:.4f}, Test Acc: {test_accuracy:.2f}%")

plt.figure(figsize=(12,5))

plt.subplot(1,2,1)
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss")
plt.legend()

plt.subplot(1,2,2)
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy")
plt.legend()

plt.show()

classes = train_loader.dataset.classes

def imshow(img):
    img = img.cpu().numpy().transpose((1, 2, 0))
    mean = np.array((0.5071, 0.4865, 0.4409))
    std = np.array((0.2673, 0.2564, 0.2761))
    img = std * img + mean
    img = np.clip(img, 0, 1)
    plt.imshow(img)
```

```
    plt.axis("off")

dataiter = iter(test_loader)
images, labels = next(dataiter)
images, labels = images.to(device), labels.to(device)

model.eval()
with torch.no_grad():
    outputs = model(images)
    _, predicted = torch.max(outputs, 1)

plt.figure(figsize=(16,8))
for i in range(8):
    plt.subplot(2,4,i+1)
    imshow(images[i])
    plt.title(f"Pred: {classes[predicted[i]]}\nTrue: {classes[labels[i]]}",
              fontsize=9, color=("green" if predicted[i]==labels[i] else "red"))
plt.show()
```





SOTA-результаты для выборки:

Для сравнения я взял свою компактную 6-слойную сверточную сеть с BatchNorm, LeakyReLU и Global Average Pooling, которая обучалась на CIFAR-100 100 эпох, batch size 128 и learning rate 3e-5. Моя модель показала точность :

Train Loss: 0.7023, Train Acc: 78.33% | Test Loss: 2.6973, Test Acc: 45.89%

Современная гибридная архитектура ConvMLP-S сочетает сверточные слои и MLP-блоки, использует stage-wise иерархический дизайн и оптимизирована для эффективного извлечения локальных и глобальных признаков. ConvMLP-S достигает **76.8% top-1 точности на ImageNet-1k**, при этом имеет 9 млн параметров и 2.4G MACs, что значительно меньше, чем у MLP-Mixer-B/16. Код и предобученные модели доступны по ссылке https://hyper.ai/en/sota/tasks/image-classification/benchmark/image-classification-on-cifar-100.

Разница в точности объясняется тем, что ConvMLP-S:

1. Более глубокая и сложная модель с большим числом параметров.
2. Использует MLP-блоки и stage-wise архитектуру для моделирования глобальных зависимостей.
3. Оптимизирована для эффективного вычисления и извлечения признаков.
4. Применяет продвинутые подходы к обучению и регуляризации.

Моя модель компактная, ориентирована на базовые локальные свёртки, и показывает адекватную производительность на CIFAR-100, однако уступает современным гибридным архитектурам по точности и способности извлекать более сложные признаки.

Вывод: научился конструировать нейросетевые классификаторы и выполнять их обучение на известных выборках компьютерного зрения