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

Ежевский Е. Р.

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:

№ варианта	Выборка	Размер исходного изображения	Оптимизатор
1	MNIST	28X28	SGD

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((0.1307,), (0.3081,))

])
```

```
test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_loader = DataLoader(
    torchvision.datasets.MNIST('/files/', train=True, download=True,
                               transform=train_transform),
    batch_size=128, shuffle=True)

test_loader = DataLoader(
    torchvision.datasets.MNIST('/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(1, 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.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc1 = nn.Linear(128, 64)
        self.fc2 = nn.Linear(64, 10)
        self.relu = nn.LeakyReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
```

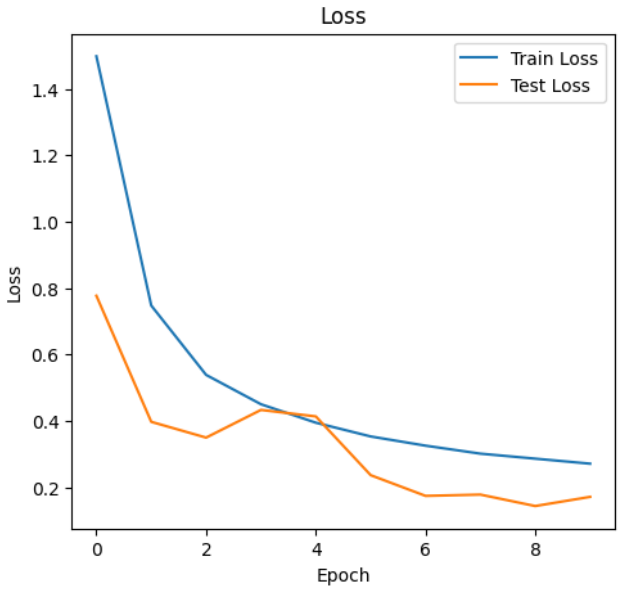
```
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.pool(self.relu(self.bn2(self.conv2(x))))
        x = self.relu(self.bn3(self.conv3(x)))
        x = self.pool(x)
        x = self.global_avg_pool(x)
        x = x.view(-1, 128)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = NN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=1e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
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()
```

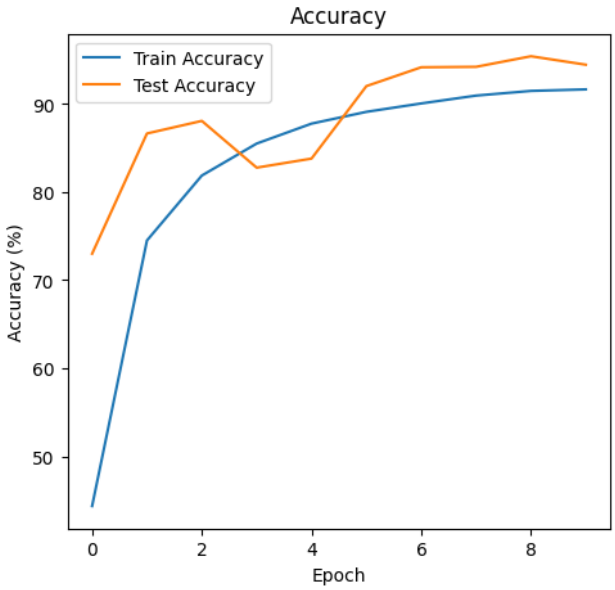
```
        total += labels.size(0)

        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 = 10
```

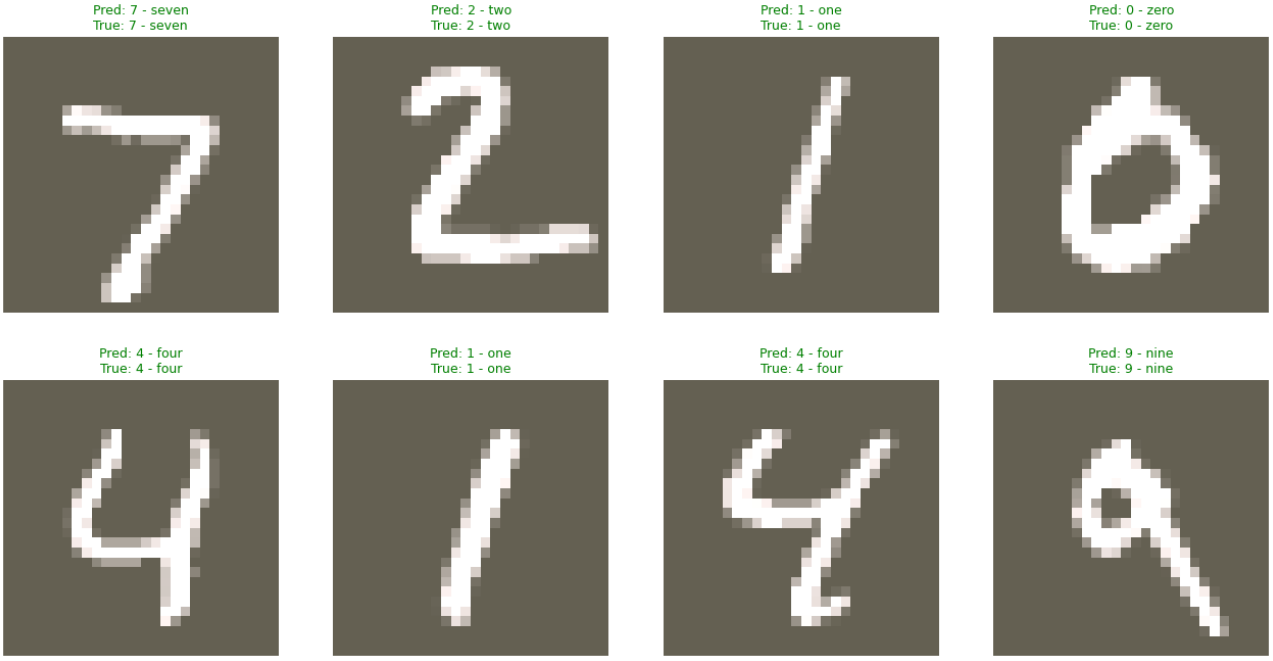
```
for epoch in range(num_epochs):
    print(f"\nEpoch {epoch+1}/{num_epochs}")
    train_loss, train_accuracy = train(model, train_loader, criterion, optimizer, device)
    test_loss, test_accuracy = test(model, test_loader, criterion, device)
    scheduler.step()
    train_losses.append(train_loss)
    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}%")
    print(f"Learning rate: {scheduler.get_last_lr()[0]:.6f}")
plt.figure(figsize=(12, 4))
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.legend()
plt.title('Loss over epochs')
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.legend()
plt.title('Accuracy over epochs')
plt.tight_layout()
plt.show()

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 сверточную модель для MNIST — сеть из трёх сверточных блоков с BatchNorm, LeakyReLU, MaxPool и Global Average Pooling, дополненную двухслойным классификатором и Dropout-регуляризацией. Модель обучалась 10 эпох с batch size 128 и оптимизатором SGD (lr=0.01). Полученные результаты составили:

Train Loss: 0.2717, Train Acc: 91.67% | Test Loss: 0.1718, Test Acc: 94.47%

Несмотря на простоту архитектуры, модель показывает стабильное обучение и высокую точность на MNIST, что достигается за счёт комбинации нормализации, LeakyReLU-активации и глобального усреднения признаков.

Однако в литературе существуют значительно более сильные решения. Например, ансамбль простых CNN разной конфигурации (3×3, 5×5 и 7×7 ядра), представленный в работе *An Ensemble of Simple Convolutional Neural Network Models for MNIST Digit Recognition*, достигает **99.87%** точности. Более продвинутый двухуровневый ансамбль — уже **99.91%**, что фактически приближается к пределам ошибаемости на MNIST.

Также современные гибридные архитектуры, такие как Capsule-CNN или сверточно-MLP-комбинации, демонстрируют точность порядка **99.8%**, используя более глубокие сети и глобальное моделирование зависимостей.

Разница в точности объясняется тем, что SOTA-модели для MNIST:

1. **Используют ансамбли из нескольких сетей**, что существенно снижает ошибку за счёт усреднения предсказаний.
2. **Обладают большей глубиной или разнообразием архитектур** (CNN разных масштабов, капсульные слои, гибриды CNN + MLP).
3. **Применяют расширенные техники регуляризации и аугментации**, уменьшающие переобучение.
4. **Максимально используют слабую вариативность датасета**, подгоняя архитектуру под его структуру для достижения результатов >99.8%.

Моя модель, напротив, компактная и ориентирована на локальные свёрточные операции, без ансамблей или сложных блоков глобального взаимодействия. Она демонстрирует адекватную и высокую точность для своей архитектуры, но закономерно уступает современным SOTA-решениям, специально оптимизированным для выжимания последних десятых процента на MNIST.

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