Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №2

По дисциплине: «ОИвИС»

Тема: «Конструирование моделей на базе предобученных нейронных сетей»

Выполнила:

Студентка 4 курса

Группы ИИ-23

Тутина Е.Д.

Проверила:

Андренко К. В.

Брест 2025

Цель работы: осуществлять обучение НС, сконструированных на базе предобученных архитектур НС .

Вариант 11

Выборка: MNIST
Оптимизатор: Adadelta
Предобученная архитектура: ResNet34

Общее задание:

1. Для заданной выборки и архитектуры предобученной нейронной организовать процесс обучения НС, предварительно изменив структуру слоев, в соответствии с предложенной выборкой. Использовать тот же оптимизатор, что и в ЛР №1. Построить график изменения ошибки и оценить эффективность обучения на тестовой выборке;

2. Сравнить полученные результаты с результатами, полученными на кастомных архитектурах из ЛР №1;

3. Ознакомиться с state-of-the-art результатами для предлагаемых выборок (по материалам в сети Интернет). Сделать выводы о результатах обучения НС из п. 1 и 2;

4. Реализовать визуализацию работы предобученной СНС и кастомной (из ЛР 1). Визуализация осуществляется посредством выбора и подачи на сеть произвольного изображения (например, из сети Интернет) с отображением результата классификации;

5. Оформить отчет по выполненной работе, залить исходный код и отчет в соответствующий репозиторий на github.

Код работы:

```
import os

import time

import copy

from pathlib import Path

import requests

from io import BytesIO

from PIL import Image

import argparse



import torch

import torch.nn as nn
```

```python
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, models

import matplotlib.pyplot as plt

IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD = [0.229, 0.224, 0.225]

def repeat_gray(x):
    if x.shape[0] == 1:
        return x.repeat(3,1,1)
    return x

def get_model(num_classes=10, pretrained=True, repeat_gray=True):
    model = models.resnet34(weights=models.ResNet34_Weights.IMAGENET1K_V1 if pretrained else None)
    if not repeat_gray:
        old_conv = model.conv1
        new_conv = nn.Conv2d(1, old_conv.out_channels, kernel_size=old_conv.kernel_size,
                             stride=old_conv.stride, padding=old_conv.padding, bias=old_conv.bias is not None)
        if pretrained:
            with torch.no_grad():
                new_conv.weight[:,0:1,:,:] = old_conv.weight.mean(dim=1, keepdim=True)
                if old_conv.bias is not None:
                    new_conv.bias.copy_(old_conv.bias)
        model.conv1 = new_conv
    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, num_classes)
    return model


def train_one_epoch(model, loader, criterion, optimizer, device, verbose=True):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for i, (images, labels) in enumerate(loader, 1):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        running_loss += loss.item() * images.size(0)
        _, preds = outputs.max(1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

        if verbose and i % 50 == 0:  # каждые 50 батчей
            print(f'Batch {i}/{len(loader)} | Loss: {running_loss/total:.4f} | Acc: {correct/total:.4f}', flush=True)

    epoch_loss = running_loss / total
    epoch_acc = correct / total
    return epoch_loss, epoch_acc

def evaluate(model, loader, criterion, device, verbose=True):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for i, (images, labels) in enumerate(loader, 1):
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            running_loss += loss.item() * images.size(0)
            _, preds = outputs.max(1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)

            if verbose and i % 50 == 0:
                print(f'Validation Batch {i}/{len(loader)} | Loss: {running_loss/total:.4f} | Acc: {correct/total:.4f}',
flush=True)

    return running_loss / total, correct / total



def load_image_from_path_or_url(path_or_url, resize=224, repeat_gray=True):
    if path_or_url.startswith('http://') or path_or_url.startswith('https://'):
        resp = requests.get(path_or_url)
        img = Image.open(BytesIO(resp.content)).convert('L')
    else:
        img = Image.open(path_or_url).convert('L')
    img_for_model = img.resize((resize, resize))
    img_tensor = transforms.ToTensor()(img_for_model)
    if repeat_gray:
```

```python
        img_tensor = img_tensor.repeat(3,1,1)
    img_tensor = transforms.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD)(img_tensor)
    return img, img_tensor.unsqueeze(0)

def predict_and_show(model, path_or_url):
    model.eval()
    img, tensor = load_image_from_path_or_url(path_or_url)
    tensor = tensor.to('cuda' if torch.cuda.is_available() else 'cpu')
    with torch.no_grad():
        outputs = model(tensor)
        probs = F.softmax(outputs, dim=1).cpu().numpy()[0]
        pred = probs.argmax()
    print(f'Predicted class: {pred}  (prob={probs[pred]:.4f})')
    display_img = img.resize((200,200)).convert('RGB')
    display_img.show()

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--predict", type=str, help="Path to image for prediction, default 'digit.png'",
                        default="digit.png")
    args = parser.parse_args()

    DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
    DATA_DIR = './data'
    BATCH_SIZE = 128
    NUM_EPOCHS = 8
    IMAGE_SIZE = 64
    NUM_CLASSES = 10
    MODEL_SAVE = 'resnet34_mnist_adadelta_best.pth'
    PLOT_SAVE = 'training_curves.png'
    PRED_SAVE = 'prediction.png'
    USE_PRETRAINED = True
    REPEAT_GRAY_TO_3 = True

    train_transform = transforms.Compose([
        transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        transforms.Lambda(repeat_gray),
        transforms.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD),
    ])

    val_transform = transforms.Compose([
        transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
```

```python
    transforms.ToTensor(),
    transforms.Lambda(repeat_gray),
    transforms.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD),
])

train_dataset = datasets.MNIST(root=DATA_DIR, train=True, download=True, transform=train_transform)
val_dataset = datasets.MNIST(root=DATA_DIR, train=False, download=True, transform=val_transform)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4, pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4, pin_memory=True)

model = get_model(NUM_CLASSES, pretrained=USE_PRETRAINED, repeat_gray=REPEAT_GRAY_TO_3).to(DEVICE)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adadelta(model.parameters())
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

train_losses, train_accs = [], []
val_losses, val_accs = [], []
best_model_wts = copy.deepcopy(model.state_dict())
best_acc = 0.0

for epoch in range(NUM_EPOCHS):
    print(f'\n=== Epoch {epoch + 1}/{NUM_EPOCHS} ===', flush=True)
    t0 = time.time()
    train_loss, train_acc = train_one_epoch(model, train_loader, criterion, optimizer, DEVICE)
    val_loss, val_acc = evaluate(model, val_loader, criterion, DEVICE)
    train_losses.append(train_loss)
    train_accs.append(train_acc)
    val_losses.append(val_loss)
    val_accs.append(val_acc)

    if val_acc > best_acc:
        best_acc = val_acc
        best_model_wts = copy.deepcopy(model.state_dict())
        torch.save(model.state_dict(), MODEL_SAVE)

    scheduler.step()
    t1 = time.time()
    print(f'Epoch {epoch + 1} finished | train_loss={train_loss:.4f} acc={train_acc:.4f} | '
          f'val_loss={val_loss:.4f} acc={val_acc:.4f} | time={(t1 - t0):.1f}s', flush=True)

model.load_state_dict(best_model_wts)
torch.save(model.state_dict(), MODEL_SAVE.replace('.pth', '_final.pth'))
print(f"Saved best model as {MODEL_SAVE} and final model as {MODEL_SAVE.replace('.pth', '_final.pth')}")
```

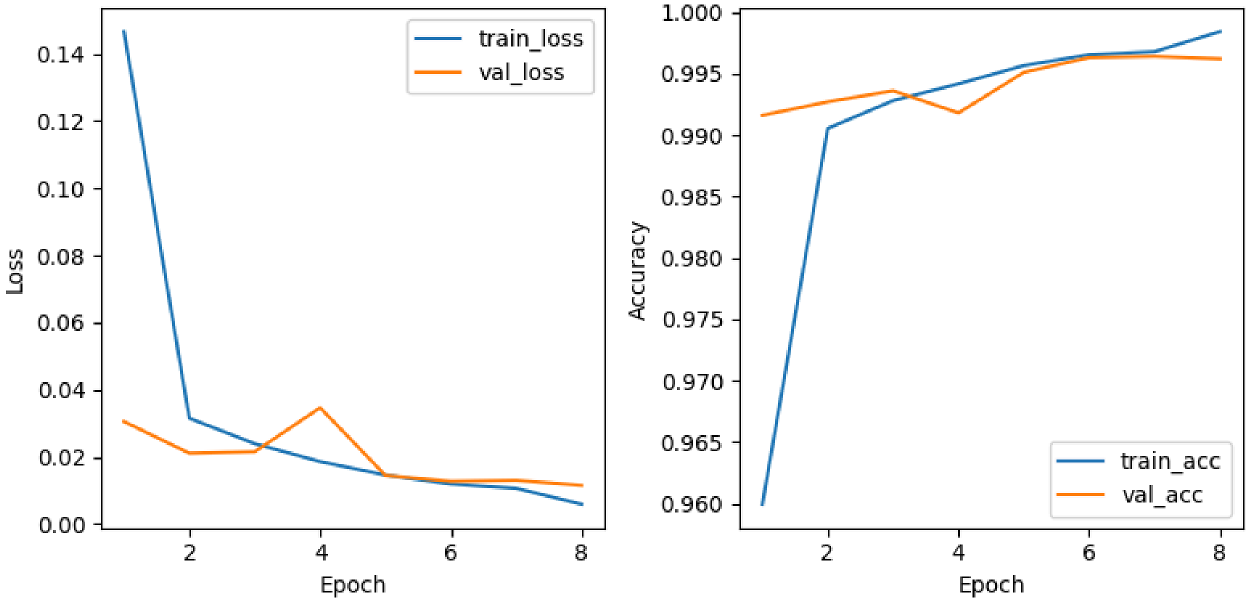
```python
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.plot(range(1, NUM_EPOCHS + 1), train_losses, label='train_loss')
    plt.plot(range(1, NUM_EPOCHS + 1), val_losses, label='val_loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(range(1, NUM_EPOCHS + 1), train_accs, label='train_acc')
    plt.plot(range(1, NUM_EPOCHS + 1), val_accs, label='val_acc')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.tight_layout()
    plt.savefig(PLOT_SAVE)
    print(f'Saved training curves to {PLOT_SAVE}')


    model.eval()
    img_path = args.predict
    img, tensor = load_image_from_path_or_url(img_path)
    tensor = tensor.to(DEVICE)
    with torch.no_grad():
        outputs = model(tensor)
        probs = F.softmax(outputs, dim=1).cpu().numpy()[0]
        pred = probs.argmax()
    print(f'Predicted class: {pred}  (prob={probs[pred]:.4f})')

    plt.imshow(img, cmap='gray')
    plt.title(f"Prediction: {pred}")
    plt.axis('off')
    plt.savefig(PRED_SAVE)
    print(f"Saved prediction image as {PRED_SAVE}")
```

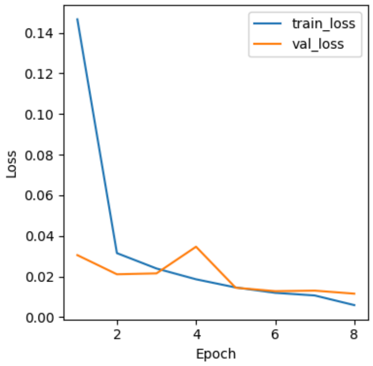
```
=== Epoch 1/8 ===
Batch 50/469 | Loss: 0.7975 | Acc: 0.7908
Batch 100/469 | Loss: 0.4653 | Acc: 0.8760
Batch 150/469 | Loss: 0.3403 | Acc: 0.9089
Batch 200/469 | Loss: 0.2728 | Acc: 0.9268
Batch 250/469 | Loss: 0.2299 | Acc: 0.9376
Batch 300/469 | Loss: 0.2031 | Acc: 0.9450
```

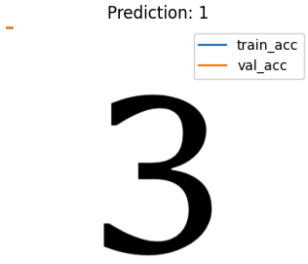
```
Batch 350/469 | Loss: 0.1816 | Acc: 0.9507
Batch 400/469 | Loss: 0.1648 | Acc: 0.9552
Batch 450/469 | Loss: 0.1511 | Acc: 0.9588
Validation Batch 50/79 | Loss: 0.0299 | Acc: 0.9911
Epoch 1 finished | train_loss=0.1466 acc=0.9599 | val_loss=0.0305 acc=0.9916 | time=625.8s

=== Epoch 2/8 ===
Batch 50/469 | Loss: 0.0346 | Acc: 0.9903
Batch 100/469 | Loss: 0.0340 | Acc: 0.9903
Batch 150/469 | Loss: 0.0341 | Acc: 0.9904
Batch 200/469 | Loss: 0.0329 | Acc: 0.9903
Batch 250/469 | Loss: 0.0341 | Acc: 0.9899
Batch 300/469 | Loss: 0.0339 | Acc: 0.9898
Batch 350/469 | Loss: 0.0331 | Acc: 0.9900
Batch 400/469 | Loss: 0.0321 | Acc: 0.9903
Batch 450/469 | Loss: 0.0312 | Acc: 0.9906
Validation Batch 50/79 | Loss: 0.0265 | Acc: 0.9912
Epoch 2 finished | train_loss=0.0315 acc=0.9905 | val_loss=0.0211 acc=0.9927 | time=637.7s

=== Epoch 3/8 ===
Batch 50/469 | Loss: 0.0237 | Acc: 0.9911
Batch 100/469 | Loss: 0.0264 | Acc: 0.9913
Batch 150/469 | Loss: 0.0234 | Acc: 0.9925
Batch 200/469 | Loss: 0.0222 | Acc: 0.9930
Batch 250/469 | Loss: 0.0214 | Acc: 0.9934
Batch 300/469 | Loss: 0.0223 | Acc: 0.9931
Batch 350/469 | Loss: 0.0229 | Acc: 0.9929
Batch 400/469 | Loss: 0.0237 | Acc: 0.9928
Batch 450/469 | Loss: 0.0243 | Acc: 0.9927
Validation Batch 50/79 | Loss: 0.0267 | Acc: 0.9914
Epoch 3 finished | train_loss=0.0240 acc=0.9928 | val_loss=0.0216 acc=0.9936 | time=694.2s

=== Epoch 4/8 ===
Batch 50/469 | Loss: 0.0137 | Acc: 0.9953
Batch 100/469 | Loss: 0.0173 | Acc: 0.9942
Batch 150/469 | Loss: 0.0184 | Acc: 0.9941
Batch 200/469 | Loss: 0.0183 | Acc: 0.9943
Batch 250/469 | Loss: 0.0182 | Acc: 0.9942
Batch 300/469 | Loss: 0.0173 | Acc: 0.9946
```

```
Batch 350/469 | Loss: 0.0173 | Acc: 0.9946
Batch 400/469 | Loss: 0.0181 | Acc: 0.9943
Batch 450/469 | Loss: 0.0179 | Acc: 0.9943
Validation Batch 50/79 | Loss: 0.0317 | Acc: 0.9911
Epoch 4 finished | train_loss=0.0186 acc=0.9942 | val_loss=0.0347 acc=0.9918 | time=694.0s

=== Epoch 5/8 ===
Batch 50/469 | Loss: 0.0158 | Acc: 0.9967
Batch 100/469 | Loss: 0.0129 | Acc: 0.9966
Batch 150/469 | Loss: 0.0123 | Acc: 0.9965
Batch 200/469 | Loss: 0.0131 | Acc: 0.9962
Batch 250/469 | Loss: 0.0132 | Acc: 0.9961
Batch 300/469 | Loss: 0.0135 | Acc: 0.9959
Batch 350/469 | Loss: 0.0145 | Acc: 0.9956
Batch 400/469 | Loss: 0.0145 | Acc: 0.9956
Batch 450/469 | Loss: 0.0148 | Acc: 0.9956
Validation Batch 50/79 | Loss: 0.0170 | Acc: 0.9941
Epoch 5 finished | train_loss=0.0146 acc=0.9957 | val_loss=0.0145 acc=0.9951 | time=691.6s

=== Epoch 6/8 ===
Batch 50/469 | Loss: 0.0096 | Acc: 0.9969
Batch 100/469 | Loss: 0.0108 | Acc: 0.9969
Batch 150/469 | Loss: 0.0103 | Acc: 0.9969
Batch 200/469 | Loss: 0.0122 | Acc: 0.9964
Batch 250/469 | Loss: 0.0118 | Acc: 0.9966
Batch 300/469 | Loss: 0.0117 | Acc: 0.9965
Batch 350/469 | Loss: 0.0118 | Acc: 0.9966
Batch 400/469 | Loss: 0.0120 | Acc: 0.9965
Batch 450/469 | Loss: 0.0119 | Acc: 0.9965
Validation Batch 50/79 | Loss: 0.0149 | Acc: 0.9952
Epoch 6 finished | train_loss=0.0120 acc=0.9965 | val_loss=0.0128 acc=0.9963 | time=690.6s

=== Epoch 7/8 ===
Batch 50/469 | Loss: 0.0061 | Acc: 0.9984
Batch 100/469 | Loss: 0.0075 | Acc: 0.9973
Batch 150/469 | Loss: 0.0084 | Acc: 0.9971
Batch 200/469 | Loss: 0.0100 | Acc: 0.9969
Batch 250/469 | Loss: 0.0108 | Acc: 0.9967
Batch 300/469 | Loss: 0.0110 | Acc: 0.9967
```

```
Batch 350/469 | Loss: 0.0105 | Acc: 0.9969
Batch 400/469 | Loss: 0.0111 | Acc: 0.9967
Batch 450/469 | Loss: 0.0110 | Acc: 0.9967
Validation Batch 50/79 | Loss: 0.0132 | Acc: 0.9959
Epoch 7 finished | train_loss=0.0107 acc=0.9968 | val_loss=0.0131 acc=0.9964 | time=627.8s

=== Epoch 8/8 ===
Batch 50/469 | Loss: 0.0056 | Acc: 0.9986
Batch 100/469 | Loss: 0.0079 | Acc: 0.9980
Batch 150/469 | Loss: 0.0072 | Acc: 0.9980
Batch 200/469 | Loss: 0.0067 | Acc: 0.9981
Batch 250/469 | Loss: 0.0073 | Acc: 0.9981
Batch 300/469 | Loss: 0.0068 | Acc: 0.9982
Batch 350/469 | Loss: 0.0064 | Acc: 0.9983
Batch 400/469 | Loss: 0.0063 | Acc: 0.9983
Batch 450/469 | Loss: 0.0060 | Acc: 0.9984
Validation Batch 50/79 | Loss: 0.0121 | Acc: 0.9956
Epoch 8 finished | train_loss=0.0059 acc=0.9984 | val_loss=0.0116 acc=0.9962 | time=594.4s
Saved best model as resnet34_mnist_adadelta_best.pth and final model as resnet34_mnist_adadelta_best_final.pth
Saved training curves to training_curves.png
Predicted class: 1  (prob=1.0000)
Saved prediction image as prediction.png
```







Полученные результаты в данной работе: `Acc: 0.9984,` В ЛР1 `: 0.9929`

Предобученная ResNet34 показывает чуть более высокую точность на тестовой выборке MNIST по сравнению с кастомной сетью из ЛР1. Разница ≈ 0.55%, что на MNIST, где задача относительно простая, уже заметно.

Сравнение с результатами SOTA :

State-of-the-Art (SOTA) по MNIST:
Классические CNN (LeNet-5): ~99.2–99.3%
ResNet, VGG, более сложные CNN: ~99.5–99.7%
Capsule Networks (Hinton, 2017): ~99.75%
Ансамбль моделей: ~99.8–99.9%
Результаты, полученные при обучении модели в ходе лабораторной работы: `TestAcc=99.84%` Отставание от лучших моделей (99.8–99.9%) отставания практически нет, значение в пределах 99.8–99.9%.