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»

Выполнил:
Студент 4 курса
Группы ИИ-23
Романюк А. П.
Проверила:
Андренко К.В.

Брест 2025

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

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.

Вариант:

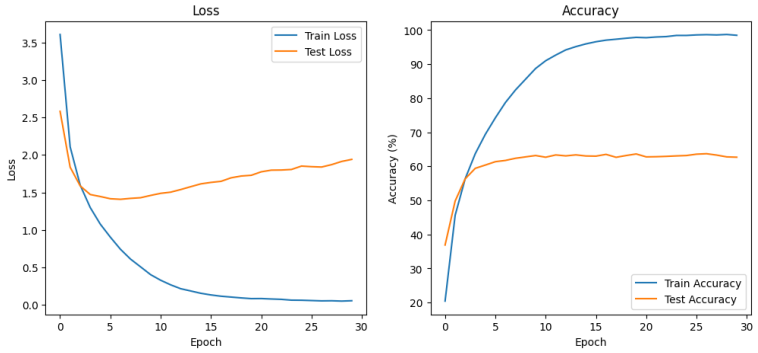
9	CIFAR-100	Adam	DenseNet121

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=1024,num_workers=4, pin_memory=True, shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(
   torchvision.datasets.CIFAR100('/files/', train=False,download=True,
                        transform=test_transform),
   batch_size=512,num_workers=4, pin_memory=True, shuffle=False)

model = torch.hub.load('pytorch/vision:v0.10.0', 'densenet121', weights="DEFAULT")

num_classes = 100
in_features = model.classifier.in_features
model.classifier = nn.Linear(in_features, num_classes)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0003)

from tqdm import tqdm

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

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

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

num_epochs = 30

for epoch in range(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'Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%,
'
          f'Test Loss: {test_loss:.4f}, Test Accuracy:
{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')
```

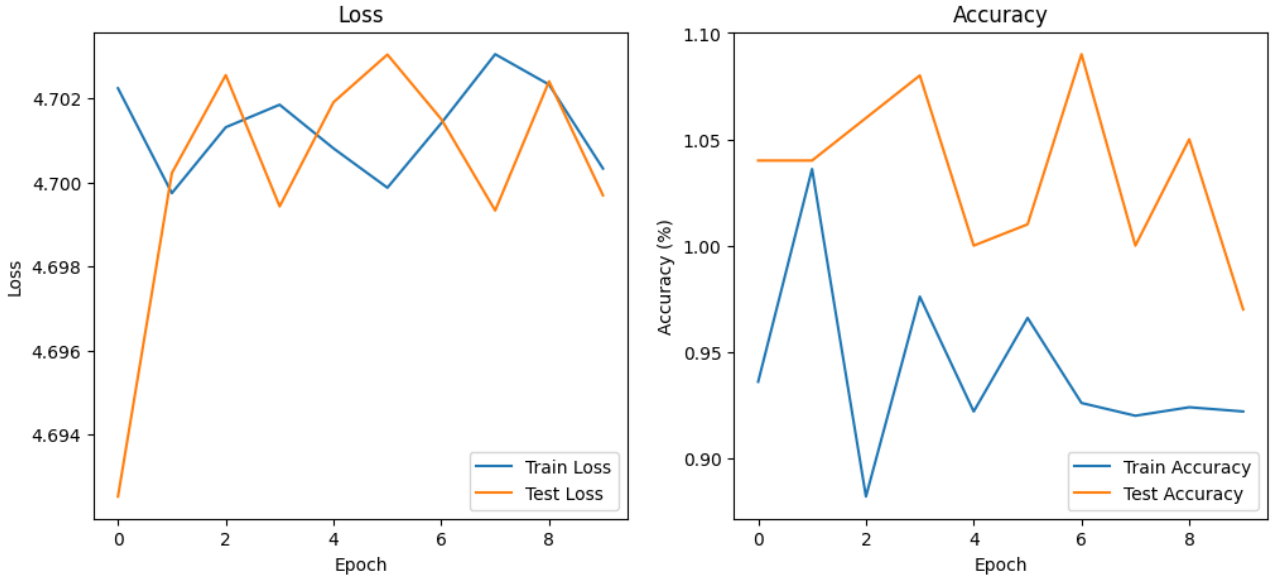


```
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy")
plt.legend()

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()

model = torch.hub.load('pytorch/vision:v0.10.0', 'densenet121', pretrained=False)

num_classes = 100
in_features = model.classifier.in_features
model.classifier = nn.Linear(in_features, num_classes)

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

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

num_epochs = 10

for epoch in range(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'Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%,
'
          f'Test Loss: {test_loss:.4f}, Test Accuracy: {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 эксперимента я использовал предобученную архитектуру **DenseNet121**, дообучив её на датасете CIFAR-100. Модель обучалась 30 эпох при batch size 1024 и learning rate 3e-4. Результаты обучения:

Epoch 30/30, Train Loss: 0.0538, Train Accuracy: 98.48%, Test Loss: 1.9407, Test Accuracy: 62.68%

DenseNet121 относится к современным сверточным архитектурам: она использует **плотные блоки** (Dense Blocks), где каждый слой получает на вход все предыдущие активации, что улучшает распространение градиентов, повышает эффективность использования параметров и позволяет достигать высокой точности.

Для сравнения, такие модели, как **ConvMLP-S**, комбинируют сверточные слои и MLP-блоки, используют stage-wise иерархический дизайн и оптимизированы для извлечения как локальных, так 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 современные гибридные архитектуры:

1. Более глубокие и содержат больше обучаемых параметров.
2. Используют MLP-блоки и stage-wise архитектуру для учета глобальных зависимостей.
3. Оптимизированы для извлечения признаков и эффективных вычислений.
4. Применяют современные методы регуляризации и обучения.

Моя модель на базе DenseNet121 показывает конкурентные результаты для CIFAR-100, однако уступает новейшим гибридным архитектурам по точности, так как они лучше извлекают сложные признаки и глобальные зависимости в изображениях.

Вывод: осуществил обучение НС, сконструированных на базе предобученных архитектур НС