Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 5

по курсу «Теория искусственных нейронных сетей»

«Сверточные нейронные сети (CNN)»

Студент группы ИУ9-72Б Терентьева А. С.

Преподаватель Каганов Ю. Т.

Москва 2023

1 Цель

1. Изучение сверточных нейронных сетей.
2. Программная реализация архитектур сверточных нейронных сетей.
3. Обучение нейронных сетей на распознавание изображений.

2 Задание

1. Реализовать три модели CNN:
   (a) LeNet (dataset: MNIST)
   (b) VGG16 (dataset: CIFAR10)
   (c) ResNet (dataset: CIFAR100)
2. Провести сравнительный анализ современных методов оптимизации (SGD, NAG, AdaDelta, ADAM) для каждой модели
3. Провести поиск оптимальных гиперпараметров для каждой оптимизации.

Для реализации использовать фреймворк PyTorch.

3 Реализация

Исходный код программы представлен в листингах 1– 3.

Листинг 1: LeNet.py

```
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
from itertools import islice
import time
import matplotlib.pyplot as plt

batch_size = 64
```

```python
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True)
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=len(testset), shuffle=True)

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool2 = nn.AvgPool2d(2, 2)
        self.fc1 = nn.Linear(256, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool2(torch.relu(self.conv2(x)))
        x = x.view(-1, 256)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net0 = LeNet().to(device)
net = LeNet().to(device)
criterion = nn.CrossEntropyLoss()

def resetNet():
    net.load_state_dict(net0.state_dict())

def countAccuracy():
    correct = 0
    size = len(testloader)
    for image, label in islice(testloader, size):
        image, label = image.to(device), label.to(device)
        output = net(image)
        _, predicted = torch.max(output.data, 1)
        correct += (predicted == label).sum().item()
```

```
    return 100 * correct / len(testset)

def train(optimizer, optim_name):
    print(f"{optim_name}:")
    start_time = time.time()
    xs, ys = [], []
    batch_num = len(trainloader)
    for epoch in range(5):
        running_loss = 0.0
        for inputs, labels in islice(trainloader, batch_num):
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        xs.append(epoch + 1)
        ys.append(running_loss / len(trainset))
        print(f'          : {xs[-1]},              : {ys[-1]}')
    plt.plot(xs, ys, label = optim_name)
    print(f"                    {optim_name}: {countAccuracy():.2f}%")
    print(f"                                : {time.time() - start_time}    \n
    ")

resetNet()
print(f"                                                  : {countAccuracy():.2f}%
    ")
SGD = optim.SGD(net.parameters(), lr=0.1)
train(SGD, "SGD")
resetNet()
AdaDelta = optim.Adadelta(net.parameters(), lr=1.0)
train(AdaDelta, "AdaDelta")
resetNet()
NAG = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, nesterov=True)
train(NAG, "NAG")
resetNet()
Adam = optim.Adam(net.parameters(), lr=0.005)
train(Adam, "Adam")

plt.legend()
plt.show()
```

Листинг 2: VGG16.py

```
import torch
import torch.nn as nn
```

```python
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import time
import matplotlib.pyplot as plt

batch_size = 64

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((32, 32)),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data/CIFAR10', train=
    True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
    shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data/CIFAR10', train=
    False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle
    =True, num_workers=2)

class myVGG16(nn.Module):
    def __init__(self):
        super(myVGG16, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
```

```
                nn.MaxPool2d(kernel_size=2, stride=2)
            )
            #
            self.classifier = nn.Sequential(
                nn.Linear(512 * 4 * 4, 1024),
                nn.ReLU(inplace=True),
                nn.Dropout(),
                nn.Linear(1024, 1024),
                nn.ReLU(inplace=True),
                nn.Dropout(),
                nn.Linear(1024, 10)
            )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net0 = myVGG16().to(device)
net = myVGG16().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.1)

def resetNet():
    net.load_state_dict(net0.state_dict())

def countAccuracy():
    correct = 0
    for image, label in trainloader:
        image, label = image.to(device), label.to(device)
        output = net(image)
        _, predicted = torch.max(output.data, 1)
        correct += (predicted == label).sum().item()
    return 100 * correct / len(trainset)

def train(name):
    xs, ys = [], []
    for epoch in range(10):
        running_loss = 0.0
        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()

            outputs = net(inputs)
```

```
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
        xs.append(epoch + 1)
        ys.append(running_loss / len(trainset))
        print(f'        : {xs[-1]},          : {ys[-1]}')

    print(f"        {name}               : {countAccuracy():.2f}%")
    plt.plot(xs, ys, label = name)
    return xs, ys

resetNet()
print(f"                    : {countAccuracy():.2f}%")

optimizer = optim.SGD(net.parameters(), lr=0.1)
xs1, ys1 = train("SGD")

resetNet()
optimizer = optim.Adadelta(net.parameters(), lr=0.1)
xs2, ys2 = train("AdaDelta")

resetNet()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, nesterov=True)
xs3, ys3 = train("NAG")

resetNet()
optimizer = optim.Adam(net.parameters(), lr=0.001)
xs4, ys4 = train("Adam")

plt.legend()
plt.show()
```

Листинг 3: ResNet.py

```
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
from itertools import islice
import time
```

```
import matplotlib.pyplot as plt

batch_size = 64

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((32, 32)),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR100(root='./data/CIFAR100', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2)
# testset = torchvision.datasets.CIFAR100(root='./data/CIFAR100', train=False, download=True, transform=transform)
# testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=True, num_workers=2)

class Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsampling=False):
        super(Block, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsampling = downsampling
        self.relu = nn.ReLU(inplace=True)
        self.downsample = nn.Sequential()
        if downsampling:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
```

```
        if self.downsampling:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out


class ResNet(nn.Module):
    def __init__(self, num_classes=100):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(64, blocks=3, stride=1)
        self.layer2 = self.make_layer(128, blocks=4, stride=2)
        self.layer3 = self.make_layer(256, blocks=6, stride=2)
        self.layer4 = self.make_layer(512, blocks=3, stride=2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, out_channels, blocks, stride):
        layers = []
        layers.append(Block(self.in_channels, out_channels, stride, downsampling=True))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(Block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net0 = ResNet().to(device)
```

```
net = ResNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.1)

def resetNet():
    net.load_state_dict(net0.state_dict())

def countAccuracy():
    correct = 0
    for image, label in trainloader:
        image, label = image.to(device), label.to(device)
        output = net(image)
        _, predicted = torch.max(output.data, 1)
        correct += (predicted == label).sum().item()
    return 100 * correct / len(trainset)

def train(name):
    xs, ys = [], []
    for epoch in range(10):
        running_loss = 0.0
        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()

            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
        xs.append(epoch + 1)
        ys.append(running_loss / len(trainset))
        print(f'          : {xs[-1]},               : {ys[-1]}')

    print(f"                    {name}                              : {countAccuracy():.2f}%")
    plt.plot(xs, ys, label = name)
    return xs, ys

resetNet()
print(f"                                                  : {countAccuracy():.2f}%")

optimizer = optim.SGD(net.parameters(), lr=0.1)
xs1, ys1 = train("SGD")

```

```
resetNet()
optimizer = optim.Adadelta(net.parameters(), lr=0.1)
xs2, ys2 = train("AdaDelta")

resetNet()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, nesterov=
    True)
xs3, ys3 = train("NAG")

resetNet()
optimizer = optim.Adam(net.parameters(), lr=0.001)
xs4, ys4 = train("Adam")

plt.legend()
plt.show()
```

4 Результат работы

1. LeNet

Количество эпох: 5

Точность до обучения: 9,58%

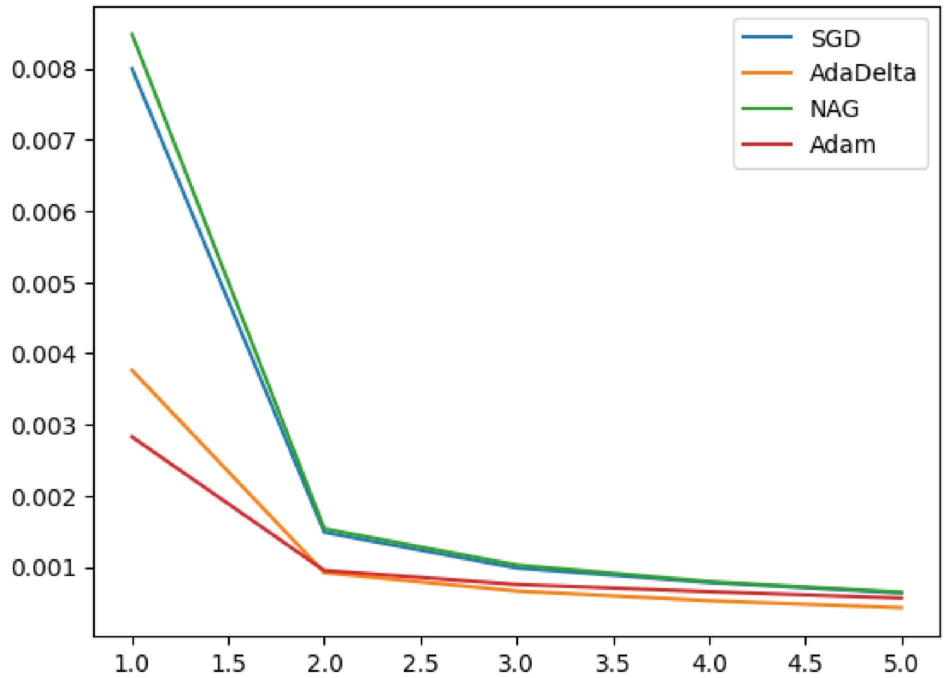

Рис. 1 — Сравнительный график сходимости методов оптимизации: зависимость значения ошибки от количества эпох

Таблица 1: Вариация гиперпараметров

Оптимизатор	Скорость обучения	Верность
SGD	0.1	98.53%
AdaDelta	1.0	98.79%
NAG	0.01	98.39%
Adam	0.005	98.69%

2. VGG16

Количество эпох: 10

Точность до обучения: 10%

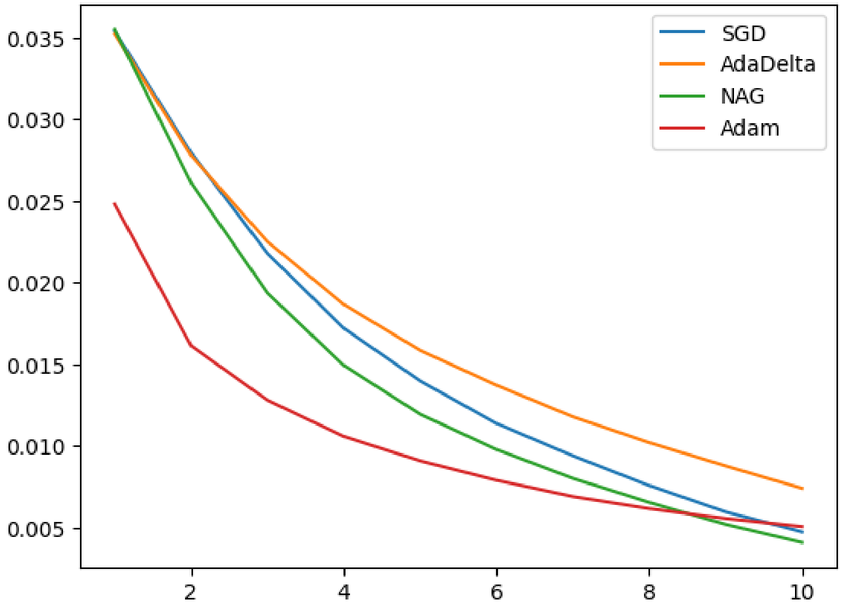

Рис. 2 — Сравнительный график сходимости методов оптимизации: зависимость значения ошибки от количества эпох

Таблица 2: Вариация гиперпараметров

Оптимизатор	Скорость обучения	Верность
SGD	0.1	91.81%
AdaDelta	0.1	84.60%
NAG	0.01	92.56%
Adam	0.001	91.19%

3. ResNet

Количество эпох: 10

Точность до обучения: 1.15%

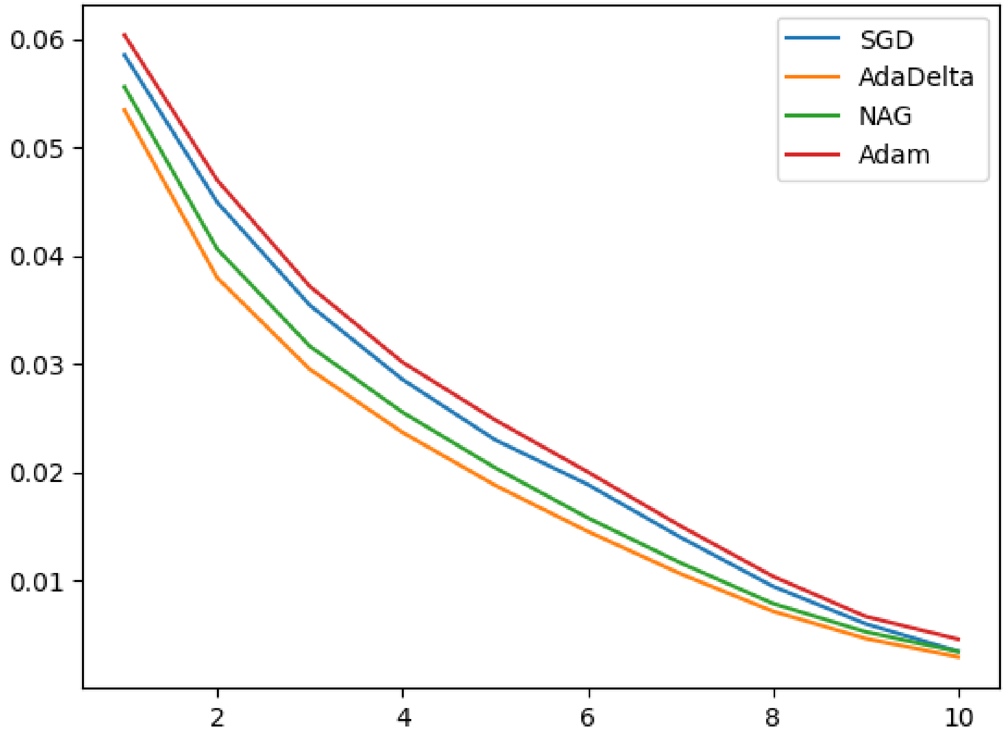

Рис. 3 — Сравнительный график сходимости методов оптимизации: зависимость значения ошибки от количества эпох

Таблица 3: Вариация гиперпараметров

Оптимизатор	Скорость обучения	Верность
SGD	0.1	93.58%
AdaDelta	0.1	95.61%
NAG	0.01	95.82%
Adam	0.001	94.79%

5 Выводы

В ходе выполнения лабораторной работы были изучены три архитектуры сверточных нейронных сетей: LeNet, VGG16 и ResNet, была написана их реализация на языке программирования Python с использованием фреймворка PyTorch. Были обучены все модели в среде выполнения Google Colab с применением GPU.

Был проведен сравнительный анализ современных методов оптимизации на каждой из модели, были подобраны оптимальные значения скорости обучения для каждой оптимизации.

В ходе эксперимента по исследованию работы программы на основе различных методов оптимизации (SGD, NAG, Adagrad, ADAM), для каждой модели были определены методы, показавшие лучший результат:

1. LeNet - AdaDelta / Adam
2. VGG16 - NAG / SGD
3. ResNet - NAG / AdaDelta