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»

«Свёрточные нейронные сети (CNN)»

Студент группы ИУ9-72Б Шевченко К.

Преподаватель Каганов Ю. Т.

Москва, 2023

Цель работы

Исследовать работу свёрточных нейронных сетей на примере архитектур LeNet5, VGG16, ResNet при использовании различных оптимизаторов (SGD, AdaDelta, NAG, Adam).

Постановка задачи

1. Загрузить и подготовить исходные данные для обучения свёрточных нейронных сетей: MNIST для LeNet, CIFAR-10 для VGG16, CIFAR-100 для ResNet;
2. Реализовать программно архитектуру моделей при помощи фреймворка PyTorch;
3. Провести обучение моделей на обучающей выборке с использованием оптимизаторов SGD, AdaDelta, NAG, Adam;
4. Подготовить сравнительные таблицы с графиками результатов исследования и проанализировать полученные результаты.

Реализация

Исходный код программы приведён в листингах 1–11.

Листинг 1: Импорт необходимых зависимостей

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchinfo import summary
import torchvision
import torchvision.transforms as transforms
import gc
import matplotlib.pyplot as plt
import numpy as np
```

Листинг 2: Скачивание и подготовка данных базы MNIST

```
transform = transforms.Compose([transforms.Resize((32,32)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.1307,), (0.3081,))])

```

```
mnist_training_data = torchvision.datasets.MNIST(root='./datasets/mnist/', train=True, download=True, transform=transform)
mnist_test_data = torchvision.datasets.MNIST(root='./datasets/mnist/', train=False, download=True, transform=transform)
mnist_training_data, mnist_test_data

```

Листинг 3: Скачивание и подготовка данных базы CIFAR-10

```
cifar10_transform_train = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.RandomHorizontalFlip(p=0.7),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
cifar10_transform_test = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

cifar10_training_data = torchvision.datasets.CIFAR10(root='./datasets/CIFAR10/', train=True, download=True, transform=cifar10_transform_train)
cifar10_test_data = torchvision.datasets.CIFAR10(root='./datasets/CIFAR10/', train=False, download=True, transform=cifar10_transform_test)
cifar10_training_data, cifar10_test_data
```

Листинг 4: Скачивание и подготовка данных базы CIFAR-100

```
cifar100_transform_train = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.RandomHorizontalFlip(p=0.7),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
cifar100_transform_test = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

cifar100_training_data = torchvision.datasets.CIFAR100(root='./datasets/CIFAR100/', train=True, download=True, transform=cifar100_transform_train)
cifar100_test_data = torchvision.datasets.CIFAR100(root='./datasets/CIFAR100/', train=False, download=True, transform=cifar100_transform_test)
cifar100_training_data, cifar100_test_data
```

Листинг 5: Отрисовка цифры MNIST

```
def draw_mnist_digit(data, example):
    plt.figure(figsize=(4, 4))
    img, label = data[example]
    plt.title('Example: %d, label: %d' % (example, label))
    plt.imshow(img.permute(1, 2, 0), cmap='gray')

draw_mnist_digit(mnist_training_data, example=7)
```

Листинг 6: Отрисовка изображений CIFAR-10

```
classes = cifar10_training_data.classes

def draw_cifar10_classes(data):
    class_images = {cls: [] for cls in classes}
    for img, label in data:
        class_images[classes[label]].append(img)
        if all(len(class_images[cls]) > 0 for cls in classes):
            break
    fig, axs = plt.subplots(2, 5, figsize=(12, 4))
    fig.subplots_adjust(hspace=0.5, wspace=0.01)
    axs = axs.ravel()
    for i, cls in enumerate(classes):
        axs[i].set_title(cls)
        img = class_images[cls][0] / 2 + 0.5
        npimg = img.numpy()
        npimg = np.clip(npimg, 0, 1)
        axs[i].imshow(np.transpose(npimg, (1, 2, 0)))

draw_cifar10_classes(cifar10_training_data)
```

Листинг 7: Отрисовка изображений CIFAR-100

```
classes = cifar100_training_data.classes

def draw_cifar100_classes(data):
    class_images = {cls: [] for cls in classes}
    for img, label in data:
        class_images[classes[label]].append(img)
        if all(len(class_images[cls]) > 0 for cls in classes):
            break
    fig, axs = plt.subplots(20, 5, figsize=(12, 48))
    fig.subplots_adjust(hspace=0.5, wspace=0.01)
    axs = axs.ravel()
    for i, cls in enumerate(classes):
        axs[i].set_title(cls)
        img = class_images[cls][0] / 2 + 0.5
        npimg = img.numpy()
        npimg = np.clip(npimg, 0, 1)
        axs[i].imshow(np.transpose(npimg, (1, 2, 0)))

draw_cifar100_classes(cifar100_training_data)
```

Листинг 8: Загрузка подготовленных данных в пакеты

```
mnist_training_data_loader = torch.utils.data.DataLoader(mnist_training_data, batch_size=64, shuffle=True)
mnist_test_data_loader = torch.utils.data.DataLoader(mnist_test_data, batch_size=64, shuffle=True)

cifar10_training_data_loader = torch.utils.data.DataLoader(cifar10_training_data, batch_size=64, shuffle=True)
cifar10_test_data_loader = torch.utils.data.DataLoader(cifar10_test_data, batch_size=64, shuffle=True)

cifar100_training_data_loader = torch.utils.data.DataLoader(cifar100_training_data, batch_size=64, shuffle=True)
cifar100_test_data_loader = torch.utils.data.DataLoader(cifar100_test_data, batch_size=64, shuffle=True)
```

Листинг 9: Архитектура LeNet5

```
class LeNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(in_features=400, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        self.fc3 = nn.Linear(in_features=84, out_features=10)

    def forward(self, x):
        output = self.pool1(F.relu(self.conv1(x)))
        output = self.pool2(F.relu(self.conv2(output)))
        output = output.view(-1, 16 * 5 * 5)
        output = F.relu(self.fc1(output))
        output = F.relu(self.fc2(output))
        output = self.fc3(output)
        return output

    def __str__(self):
        return 'LeNet'
```

Листинг 10: Архитектура VGG16

```
class VGG16(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.conv1_bn = nn.BatchNorm2d(num_features=64)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
        self.conv2_bn = nn.BatchNorm2d(num_features=64)
        self.max_pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,
        ↪  padding=1)
        self.conv3_bn = nn.BatchNorm2d(num_features=128)
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=128,
        ↪  kernel_size=3, padding=1)
        self.conv4_bn = nn.BatchNorm2d(num_features=128)
        self.max_pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256,
        ↪  kernel_size=3, padding=1)
        self.conv5_bn = nn.BatchNorm2d(num_features=256)
        self.conv6 = nn.Conv2d(in_channels=256, out_channels=256,
        ↪  kernel_size=3, padding=1)
        self.conv6_bn = nn.BatchNorm2d(num_features=256)
        self.conv7 = nn.Conv2d(in_channels=256, out_channels=256,
        ↪  kernel_size=3, padding=1)
        self.conv7_bn = nn.BatchNorm2d(num_features=256)
        self.max_pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv8 = nn.Conv2d(in_channels=256, out_channels=512,
        ↪  kernel_size=3, padding=1)
        self.conv8_bn = nn.BatchNorm2d(num_features=512)
        self.conv9 = nn.Conv2d(in_channels=512, out_channels=512,
        ↪  kernel_size=3, padding=1)
        self.conv9_bn = nn.BatchNorm2d(num_features=512)
        self.conv10 = nn.Conv2d(in_channels=512, out_channels=512,
        ↪  kernel_size=3, padding=1)
        self.conv10_bn = nn.BatchNorm2d(num_features=512)
        self.max_pool4 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv11 = nn.Conv2d(in_channels=512, out_channels=512,
        ↪  kernel_size=3, padding=1)
        self.conv11_bn = nn.BatchNorm2d(num_features=512)
        self.conv12 = nn.Conv2d(in_channels=512, out_channels=512,
        ↪  kernel_size=3, padding=1)
        self.conv12_bn = nn.BatchNorm2d(num_features=512)
        self.conv13 = nn.Conv2d(in_channels=512, out_channels=512,
        ↪  kernel_size=3, padding=1)
        self.conv13_bn = nn.BatchNorm2d(num_features=512)
        self.max_pool5 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(7 * 7 * 512, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 10)

    def forward(self, x):
        output = F.relu(self.conv1_bn(self.conv1(x)))
        output = F.relu(self.conv2_bn(self.conv2(output)))
        output = self.max_pool1(output)
        output = F.relu(self.conv3_bn(self.conv3(output)))
        output = F.relu(self.conv4_bn(self.conv4(output)))
        output = self.max_pool2(output)
        output = F.relu(self.conv5_bn(self.conv5(output)))
        output = F.relu(self.conv6_bn(self.conv6(output)))
        output = F.relu(self.conv7_bn(self.conv7(output)))
        output = self.max_pool3(output)
        output = F.relu(self.conv8_bn(self.conv8(output)))
        output = F.relu(self.conv9_bn(self.conv9(output)))
        output = F.relu(self.conv10_bn(self.conv10(output)))
        output = self.max_pool4(output)
        output = F.relu(self.conv11_bn(self.conv11(output)))
```

```
        output = F.relu(self.conv12_bn(self.conv12(output)))
        output = F.relu(self.conv13_bn(self.conv13(output)))
        output = self.max_pool5(output)
        output = output.view(-1, 7 * 7 * 512)
        output = F.relu(self.fc1(output))
        output = F.dropout(output, 0.2)
        output = F.relu(self.fc2(output))
        output = F.dropout(output, 0.2)
        output = self.fc3(output)
        return output

    def __str__(self):
        return 'VGG16'
```

Листинг 11: Архитектура ResNet

```
class BasicBlock(nn.Module):
    def __init__(self, in_planes, planes, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=in_planes, out_channels=planes,
        ↪  kernel_size=3, stride=stride, padding=1)
        self.conv1_bn = nn.BatchNorm2d(num_features=planes)
        self.conv2 = nn.Conv2d(in_channels=planes, out_channels=planes,
        ↪  kernel_size=3, stride=1, padding=1)
        self.conv2_bn = nn.BatchNorm2d(num_features=planes)

        self.shortcut = None
        if stride != 1 or in_planes != planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels=in_planes, out_channels=planes,
                ↪  kernel_size=1, stride=stride),
                nn.BatchNorm2d(num_features=planes)
            )

    def forward(self, x):
        residual = x
        output = F.relu(self.conv1_bn(self.conv1(x)))
        output = self.conv2_bn(self.conv2(output))
        output += self.shortcut(x) if self.shortcut else residual
        output = F.relu(output)
        return output

class ResNet(nn.Module):
    def __init__(self, block, num_blocks):
        super().__init__()
        self.in_planes = 64

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7,
        ↪  stride=2, padding=3)
        self.conv1_bn = nn.BatchNorm2d(num_features=64)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self.__make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self.__make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self.__make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self.__make_layer(block, 512, num_blocks[3], stride=2)
        self.avgpool = nn.AvgPool2d(kernel_size=7, stride=1)
```

```
        self.fc = nn.Linear(in_features=512, out_features=100)

    def __make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = [block(self.in_planes, planes, stride)]
        self.in_planes = planes
        for stridee in strides[1:]:
            layers.append(block(self.in_planes, planes))
        return nn.Sequential(*layers)

    def forward(self, x):
        output = F.relu(self.conv1_bn(self.conv1(x)))
        output = self.maxpool(output)
        output = self.layer1(output)
        output = self.layer2(output)
        output = self.layer3(output)
        output = self.layer4(output)
        output = self.avgpool(output)
        output = output.view(output.size(0), -1)
        output = self.fc(output)
        return output

    def __str__(self):
        return 'ResNet'
```

Тестирование

Исходный код для тестирования разработанной программы представлен в листинге 12.

Листинг 12: Тестирование

```
lr = 0.01
epochs = 30

models = (
    LeNet,
    VGG16,
    ResNet,
)

training_data_loader = {
    'LeNet': mnist_training_data_loader,
    'VGG16': cifar10_training_data_loader,
    'ResNet': cifar100_training_data_loader
}

test_data_loader = {
    'LeNet': mnist_test_data_loader,
    'VGG16': cifar10_test_data_loader,
    'ResNet': cifar100_test_data_loader
}

```

```
loss_func = nn.CrossEntropyLoss()

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

def train_and_evaluate(model, optimizer, optimizer_name):
    model = model.to(device)
    print(f'optimizer: {optimizer_name}')
    loss_values = []
    for epoch in range(1, epochs + 1):
        running_loss = 0
        for i, (inputs, labels) in enumerate(training_data_loader[str(model)]):
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            loss = loss_func(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            del inputs, labels, outputs
            torch.cuda.empty_cache()
            gc.collect()
        loss_values.append(running_loss / len(training_data_loader[str(model)]))
        print(f'Epoch {epoch}, loss: {running_loss / len(training_data_loader[str(model)])}')

    with torch.no_grad():
        correct = 0
        total = 0
        for inputs, labels in test_data_loader[str(model)]:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            del inputs, labels, outputs

        print('Accuracy: {} %'.format(100 * correct / total))
    return loss_values

def get_models_and_optimizers(model):
    model_parameters = []
    if model.__name__ == 'ResNet':
        model_parameters = [BasicBlock] + [[3, 4, 6, 3]]
    models = {
        'SGD': model(*model_parameters),
        'Adadelta': model(*model_parameters),
        'NAG': model(*model_parameters),
        'Adam': model(*model_parameters),
    }
    optimizers = {
        'SGD': optim.SGD(models['SGD'].parameters(), weight_decay=0.0001, lr=lr),
        'Adadelta': optim.Adadelta(models['Adadelta'].parameters(), lr=lr),
        'NAG': optim.SGD(models['NAG'].parameters(), lr=lr, momentum=0.9, nesterov=True),
```

```
76          'Adam': optim.Adam(models['Adam'].parameters(), lr=lr, betas=(0.9,
            ↪  0.999)),
77      }
78      return models, optimizers
79
80  def train_model(model, optimizer_data):
81      optimizer_name, optimizer = optimizer_data
82      return train_and_evaluate(model, optimizer, optimizer_name)
83
84  def plot_data(model, optimizers, y_s):
85      plt.title(model.__name__)
86      for i, optimizer_data in enumerate(optimizers.items()):
87          optimizer_name, optimizer = optimizer_data
88          x, y = range(1, epochs + 1), y_s[i]
89          plt.plot(x, y, label=optimizer_name)
90      plt.legend(bbox_to_anchor=(1, 1), loc='upper left')
91      plt.show()
92
```

В качестве функции активации нейронов использовалась функция ReLu, а на последнем слое - функция Softmax. В качестве функции потерь применялась категориальная перекрёстная энтропия (categorial cross entropy), являющаяся отличным решением в задачах классификации изображений. Данные из датасетов MNIST, CIFAR-10 и CIFAR-100 были предобработаны для улучшения качества обучения моделей и загружены в пакеты размером 64.

Вариация гиперпараметров для обучения модели LeNet5 представлена в таблице 1.

Таблица 1 — Вариация гиперпараметров LeNet5

	Оптимизатор	**Кол-во эпох**	**Скорость обучения**	**Верность**
1	SGD	10	0.01	97.3%
2	AdaDelta	10	0.01	94.65%
3	NAG	10	0.01	99.11%
4	Adam	10	0.01	98.36%

Из таблицы 1 видно, что для обучения модели с архитектурой LeNet5 оказалось достаточно всего 10 эпох при значении скорости обучения равном 0.01. Оптимизатор NAG (Nesterov accelerared gradient) показал наилучшую производительность с точностью 99.11%, что делает его предпочтительным выбором для решения данной задачи. Не меньшую эффективность продемонстрировали Adam и SGD, в то время как AdaDelta показал наименьшую точность, что может указывать на необходимость дополнительной настройки параметров или использования дополнительных подходов для улучшения производительности с данным оптимизатором.

Рисунок 1 демонстрирует зависимость loss-функции от числа эпох. Видно, что оптимизаторы NAG и Adam делают резкий скачок к минимуму уже на первой эпохе. Если сравнить полученные результаты с результатами обучения многослойного персептрона в лабораторной работе №2 для распознавания цифр MNIST, то можно сделать вывод, что использование свёрточных и pooling-слоёв действительно даёт ощутимый прирост в точности распознавания. LeNet5 позволила избежать недостатков полносвязной нейронной сети, не учитывающей пространственные связи на изображениях.

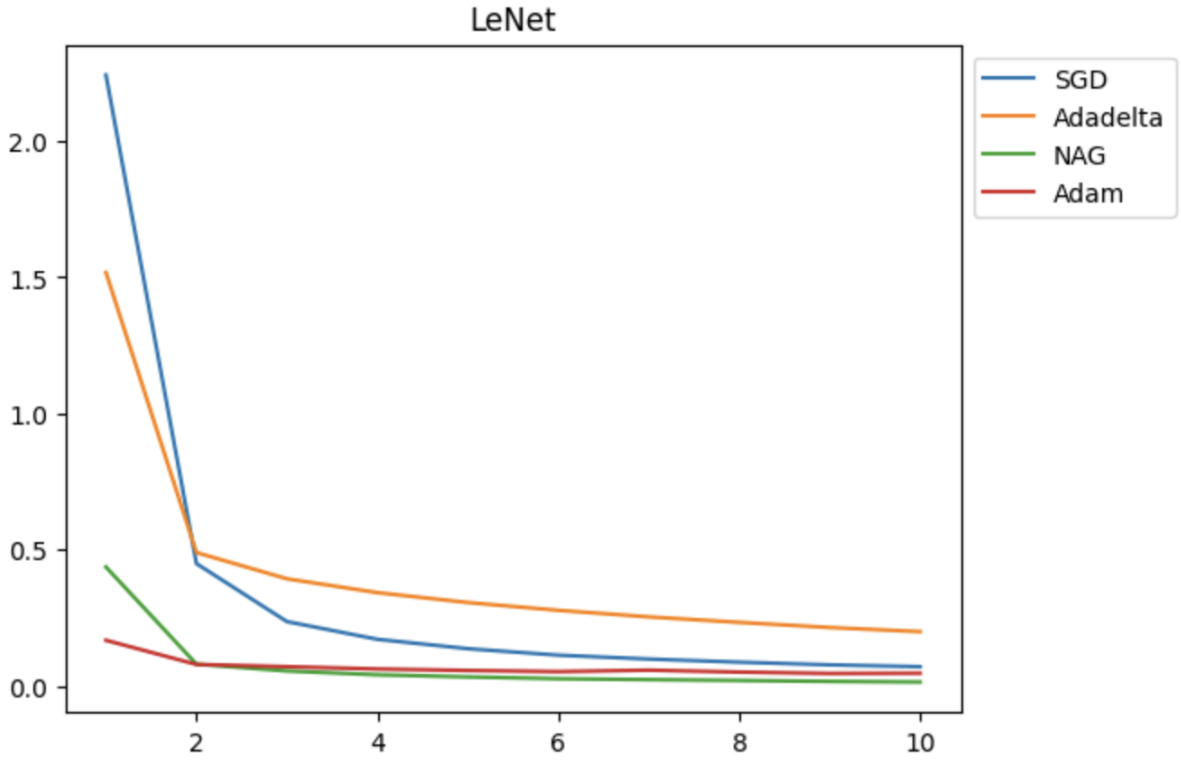

Рисунок 1 — Графики зависимости функции потерь от числа эпох при использовании различных методов оптимизации (LeNet5)

Вариация гиперпараметров для обучения модели VGG16 представлена в таблице 2.

Таблица 2 — Вариация гиперпараметров VGG16

	Оптимизатор	**Кол-во эпох**	**Скорость обучения**	**Dropout**	**Верность**
1	SGD	10	0.01	0.2	84.38%
2	AdaDelta	10	0.01	0.2	84.2%
3	NAG	10	0.01	0.2	85.44%
4	Adam	10	0.01	0.2	10.00%

В отличие от архитектуры LeNet5, архитектура VGG16 оказалась крайне глубокой и тяжёлой. Основными её недостаткоми стали большой объём занимаемой памяти из-за глубины и количества полносвязных узлов, а также медленная скорость обуче-

ния. Обучение осуществялось с использованием облачного сервера с поддержкой GPU Nvidia P100 и заняло в общей сложности около 6-7 часов.

Анализ данных из таблицы 2 говорит о приблизительно одинаковой эффективности обучения при использовании различных оптимизаторов. Исключение составил лишь Adam, использование которого на данных параметрах не привело к обучению модели. Это можно объяснить сложной архитектурой модели, чувствительной к выбору гиперпараметров и возможным застреваниям в локальных минимумах ввиду непростого ландшафта.

На рисунке 2 показаны графики зависимости функции потерь от числа эпох при использовании различных оптимизаторов за исключением Adam, продемонстрировавшим плохую эффективность.

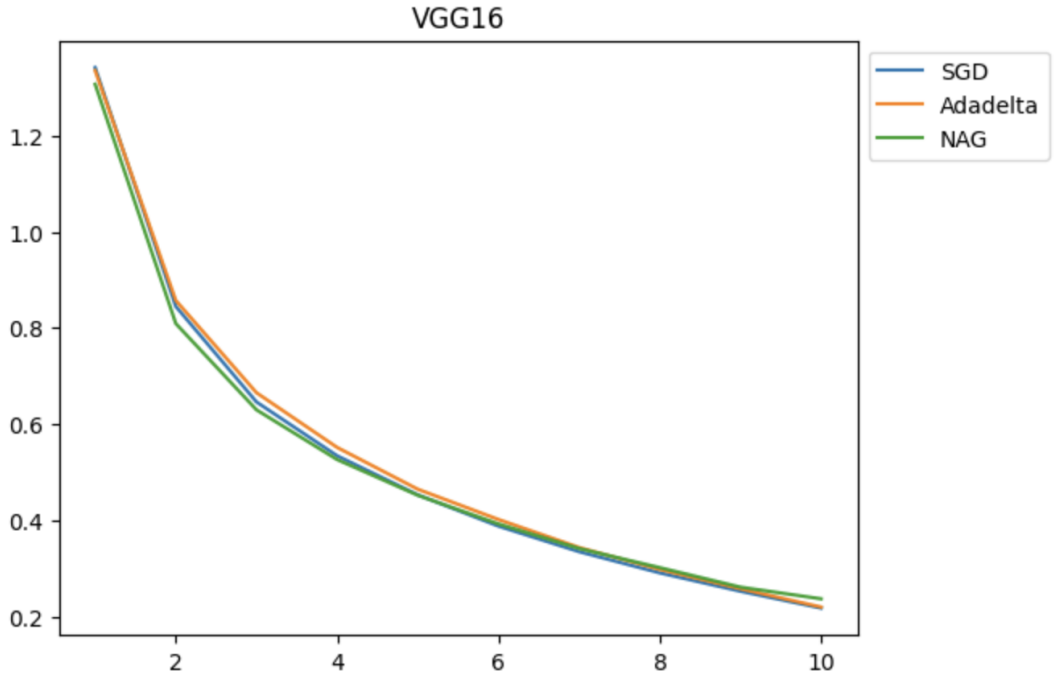

Рисунок 2 — Графики зависимости функции потерь от числа эпох при использовании различных методов оптимизации (VGG16)

Таблица 3 иллюстрирует вариацию гиперпараметров для обучения модели с архитектурой ResNet.

Таблица 3 — Вариация гиперпараметров ResNet

	Оптимизатор	**Кол-во эпох**	**Скорость обучения**	**Верность**
1	SGD	30	0.01	63.88%
2	AdaDelta	30	0.01	58.03%
3	NAG	30	0.01	65.65%
4	Adam	30	0.01	59.36%

Из таблицы видно, что из всех оптимизаторов наилучшую точность продемонстрировал ускоренный градиент Нестерова (NAG). Стохастический градиентный спуск также показал хорошие результаты обучения, опередив AdaDelta и Adam благодаря своей стабильности в контексте глубоких сетей.

Архитектура ResNet состоит из 34 слоёв, но основным её отличием от LeNet5 и VGG16 является наличие остаточных связей, позволяющих градиенту распространяться на большую глубину сети без значительного затухания. Это делает возможным обучение сетей из сотен и тысяч слоёв. Дополнительно стоит отметить, что в данной задаче классификации из 100 классов полученная максимальная верность равная 65.65% является хорошим результатом, поскольку набор данных CIFAR-100 содержит всего 600 изображений на каждый класс (для сравнения: MNIST и CIFAR-10 содержат 6000 изображений на класс). Для получения более высокой точности предсказания следует увеличивать число эпох обучения, однако обучение модели с архитектурой ResNet – задача, требующая больших вычислительных ресурсов и времени, поэтому в учебных целях было принято решение ограничиться 30 эпохами.

Рисунок 3 иллюстрирует графики зависимости функции потерь от числа эпох при использовании различных оптимизаторов.

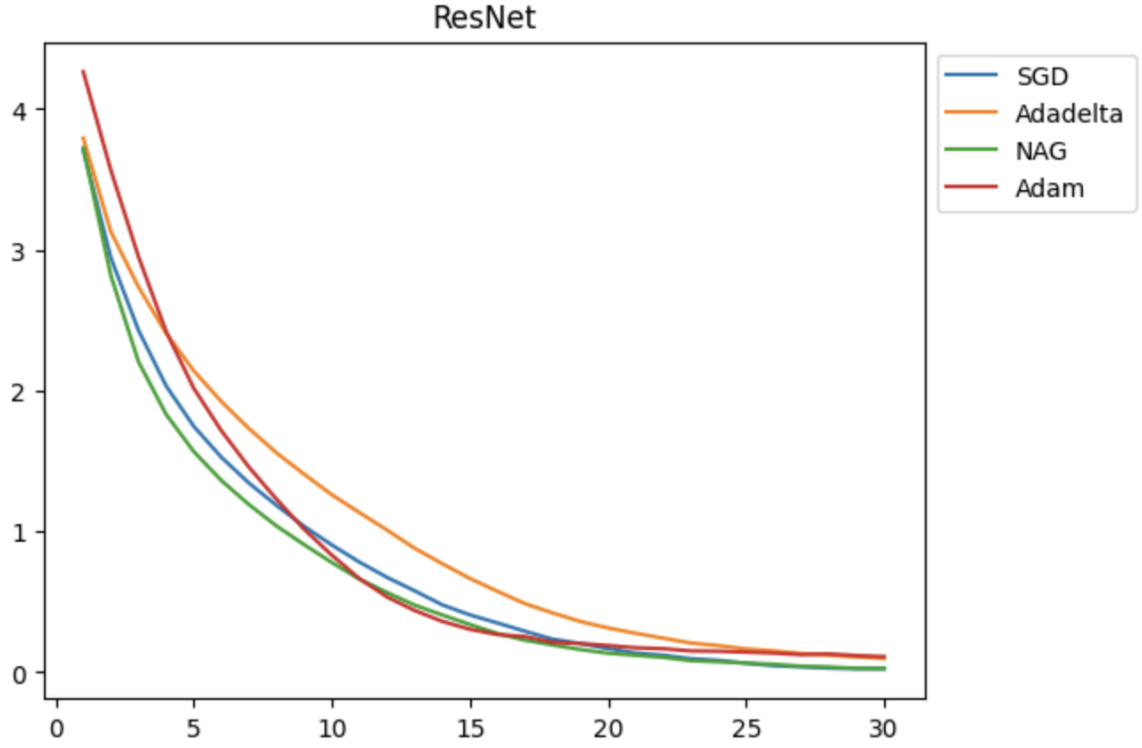

Рисунок 3 — Графики зависимости функции потерь от числа эпох при использовании различных методов оптимизации (ResNet)

Выводы

В ходе выполнения лабораторной работы были изучены основные принципы работы свёрточных нейронных сетей на примере архитектур LeNet5, VGG16 и ResNet с использованием различных оптимизаторов (SGD, AdaDelta, NAG, Adam).

Полученные результаты позволяют сделать следующие выводы:

1. Архитектура LeNet5 показала высокую эффективность в задаче классификации цифр из датасета MNIST, превзойдя результаты обучения многослойного персептрона в рамках лабораторной работы №2;
2. Несмотря на глубину архитектуры, VGG16 оказалась менее эффективной, но всё ещё с хорошими результатами обучения. Основные недостатки сети – крайне большой объём занимаемой памяти и медленая скорость обучения, что говорит о необходимости наличия мощных вычислительных ресурсов для организации эффективного обучения данной модели;
3. Архитектура ResNet выделилась благодаря наличию остаточных связей, позволяющих обучать гораздо более глубокие сети. Модель с данной архитектурой также показала приемлемую производительность, особенно с учётом сложности набора данных CIFAR-100;
4. Гиперпараметры играют ключевую роль в производительности моделей нейронных сетей. Особенно это стало заметно при анализе результатов обучения VGG16;
5. Разные оптимизаторы показывают разную эффективность. В данной работе наивысшую точность показал ускоренный градиент Нестерова (NAG), продемонстрировав важность правильного выбора оптимизатора для конкретной задачи.

В заключение, результаты лабораторной работы демонстрируют, что эффективность свёрточных нейросетей сильно зависит от выбранной архитектуры и настройки гиперпараметров. Эти факторы должны быть тщательно подобраны для достижения оптимальных результатов в задачах классификации изображений.