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»

Домашняя работа № 5

по курсу «Теория искусственных нейронных сетей»

«Сверточные нейронные сети»

Студент группы ИУ9-71Б Булкин В.А.

Преподаватель Каганов Ю.Т.

Москва 2024

1 Цель

Целью работы является изучение сверточных нейронных сетей.

2 Задачи

1. Изучить устройство LeNet, VGG16, ResNet.

2. Сравнить точность и потери в зависимости от оптимизаторов.

3 Реализация

Исходный код программы представлен в листинге 1.

Листинг 1 – CNN

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
from torchvision.datasets import MNIST, cifar


# Определение архитектуры LeNet
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1, padding=2)  # 
3x32x32 -> 6x32x32
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)  # 6x32x32 -> 6x16x16
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1)  # 6x16x16 -> 
16x12x12
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)  # 16x12x12 -> 16x6x6
        self.fc1 = nn.Linear(in_features=16 * 5 * 5, out_features=120)  # 1x576 -> 1x120
        self.fc2 = nn.Linear(in_features=120, out_features=84)  # 1x120 -> 1x84
        self.fc3 = nn.Linear(in_features=84, out_features=10)  # 1x84 -> 1x10
        # Функция активации
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
```

```
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
# Загрузка данных CIFAR-10
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
train_dataset = MNIST('.', train=True, download=True,transform=transforms.ToTensor())
test_dataset = MNIST('.', train=False,transform=transforms.ToTensor())

# Ограничение данных
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
# Функция обучения модели
def train_model(model, optimizer, criterion, train_loader, device):
    model.train()
    total_loss = 0
    correct = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
    accuracy = correct / len(train_loader.dataset)
    return total_loss / len(train_loader), accuracy

# Основной процесс
if __name__ == "__main__":
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Список оптимизаторов
    optimizers = {
        "SGD": lambda params: optim.SGD(params, lr=0.01),
        "Adagrad": lambda params: optim.Adagrad(params, lr=0.001),
        "NAG": lambda params: optim.SGD(params, lr=0.001, momentum=0.9, nesterov=True),
        "Adam": lambda params: optim.Adam(params, lr=0.001),
    }

    num_epochs = 10
    results = {}

    for opt_name, opt_func in optimizers.items():
        print(f"\nTraining with {opt_name} optimizer")

        model = LeNet().to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = opt_func(model.parameters())
```

```
    for epoch in range(num_epochs):
      train_loss, train_accuracy = train_model(model, optimizer, criterion, train_loader, device)

      print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Accuracy:
{train_accuracy:.4f}")

    results[opt_name] = train_accuracy

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

class VGG16(nn.Module):
  def __init__(self, dropout=True):
    super(VGG16, self).__init__()
    self.conv1_1 = nn.Conv2d(3, 128, kernel_size=3, padding=1)
    self.conv1_2 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
    self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.conv2_1 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
    self.conv2_2 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
    self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.conv3_1 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
    self.conv3_2 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
    self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.fc1 = nn.Linear(512 * 4 * 4, 1024)
    self.bn1 = nn.BatchNorm1d(1024)
    self.dropout = nn.Dropout() if dropout else nn.Identity()

    self.fc2 = nn.Linear(1024, 1024)
    self.bn2 = nn.BatchNorm1d(1024)

    self.fc3 = nn.Linear(1024, 10)
    self.bn3 = nn.BatchNorm1d(10)

  def forward(self, x):
    # Conv Block 1
    x = self.conv1_1(x)
    x = nn.ReLU()(x)
    x = self.conv1_2(x)
    x = nn.ReLU()(x)
    x = self.pool1(x)

    # Conv Block 2
    x = self.conv2_1(x)
    x = nn.ReLU()(x)
    x = self.conv2_2(x)
    x = nn.ReLU()(x)
    x = self.pool2(x)

    # Conv Block 3
    x = self.conv3_1(x)
```

```
        x = nn.ReLU()(x)
        x = self.conv3_2(x)
        x = nn.ReLU()(x)
        x = self.pool3(x)

        # Flatten
        x = x.view(x.size(0), -1)  # 512 * 4 * 4

        # Fully Connected Block 1
        x = self.fc1(x)
        x = self.bn1(x)
        x = nn.ReLU()(x)
        x = self.dropout(x)

        # Fully Connected Block 2
        x = self.fc2(x)
        x = self.bn2(x)
        x = nn.ReLU()(x)
        x = self.dropout(x)

        # Output Layer
        x = self.fc3(x)
        x = self.bn3(x)
        x = nn.ReLU()(x)

        return x

# Загрузка данных CIFAR-10
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
print("imhere")

full_train_dataset = datasets.CIFAR10(root="./data", train=True, download=True, transform=transform)
full_test_dataset = datasets.CIFAR10(root="./data", train=False, download=True, transform=transform)

# Ограничение данных
train_dataset = Subset(full_train_dataset, range(800))
test_dataset = Subset(full_test_dataset, range(200))

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
print("imhere")
# Функция обучения модели
def train_model(model, optimizer, criterion, train_loader, device):
    total_loss = 0
    correct = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
```

```
        correct += (predicted == labels).sum().item()
    accuracy = correct / len(train_loader.dataset)
    return total_loss / len(train_loader), accuracy

# Основной процесс
if __name__ == "__main__":
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Список оптимизаторов
    optimizers = {
        "SGD": lambda params: optim.SGD(params, lr=0.1),
        "Adagrad": lambda params: optim.Adagrad(params, lr=0.001),
        "NAG": lambda params: optim.SGD(params, lr=0.001, momentum=0.9, nesterov=True),
        "Adam": lambda params: optim.Adam(params, lr=0.001),
    }

    num_epochs = 10
    results = {}

    for opt_name, opt_func in optimizers.items():
        print(f"\nTraining with {opt_name} optimizer")

        model = VGG16().to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = opt_func(model.parameters())

        for epoch in range(num_epochs):
            train_loss, train_accuracy = train_model(model, optimizer, criterion, train_loader, device)
            print(f"Epoch {epoch+1}/{num_epochs/10}, Train Loss: {train_loss:.4f}, Accuracy:
{train_accuracy:.4f}")


        results[opt_name] = train_accuracy

    print("\nFinal Results:")
    for opt_name, accuracy in results.items():
        print(f"{opt_name}: {accuracy:.4f}")

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torch.utils.data import DataLoader

# Остаточный блок
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(BasicBlock, 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.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
```

```
        self.downsample = downsample

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out


# Модель ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.layer1 = self._make_layer(block, 16, layers[0])
        self.layer2 = self._make_layer(block, 32, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 64, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 128, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(128 * block.expansion, num_classes)

    def _make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if stride != 1 or self.in_channels != out_channels * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels * block.expansion, kernel_size=1, stride=stride,
bias=False),
                nn.BatchNorm2d(out_channels * block.expansion),
            )

        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
```

```
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.pool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

3 Результаты

Результаты для LeNet представлены на рисунках 1-2. Для VGG16 на рисунках 3-4, для ResNet на рисунках 5-6.

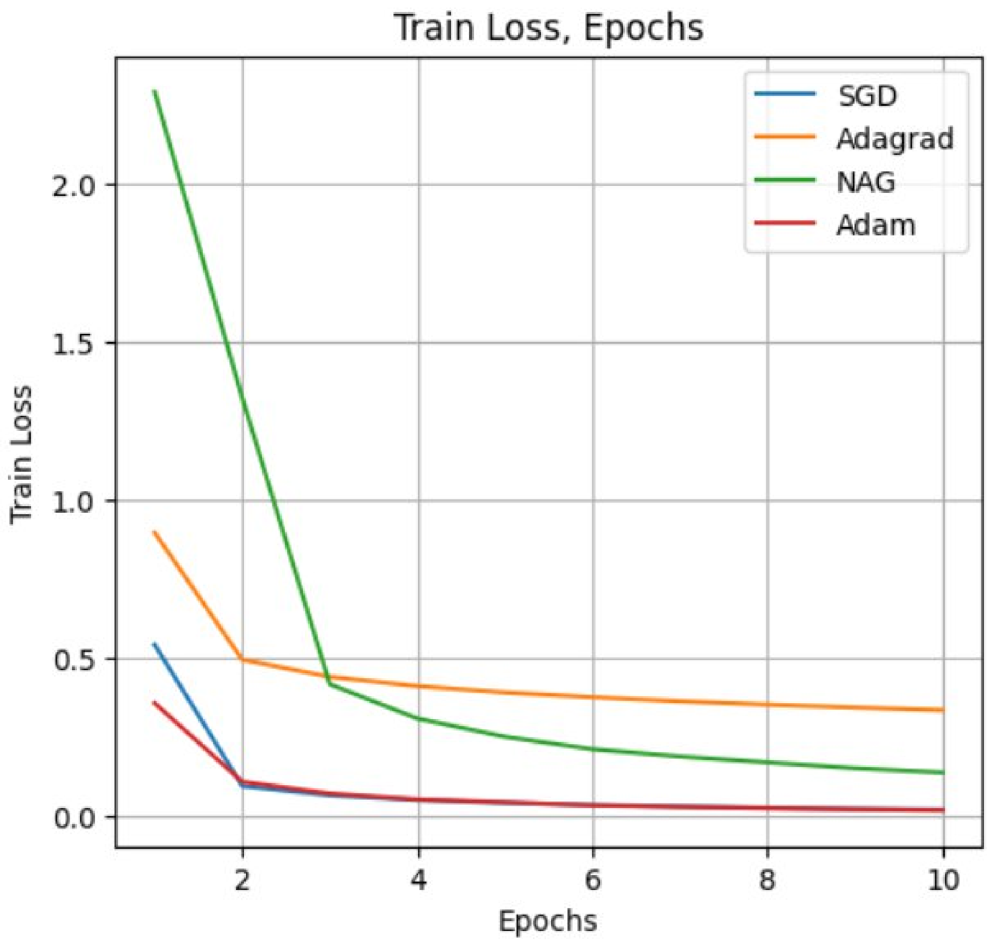

Рисунок 1 – Результаты LeNet Loss

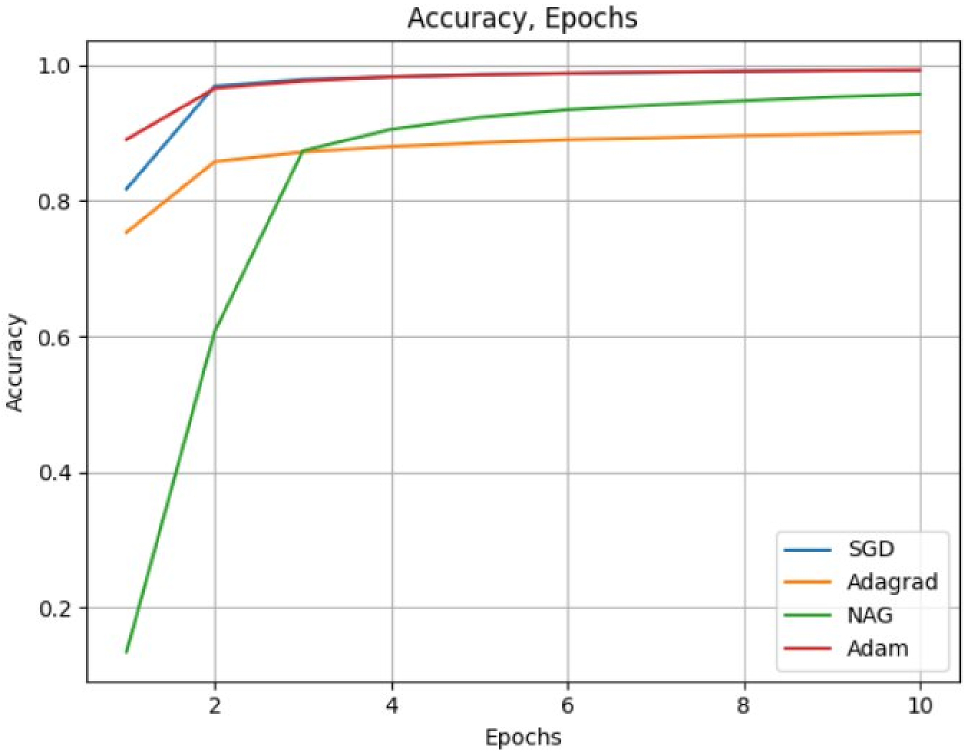

Рисунок 2 – Результаты LeNet Accuracy

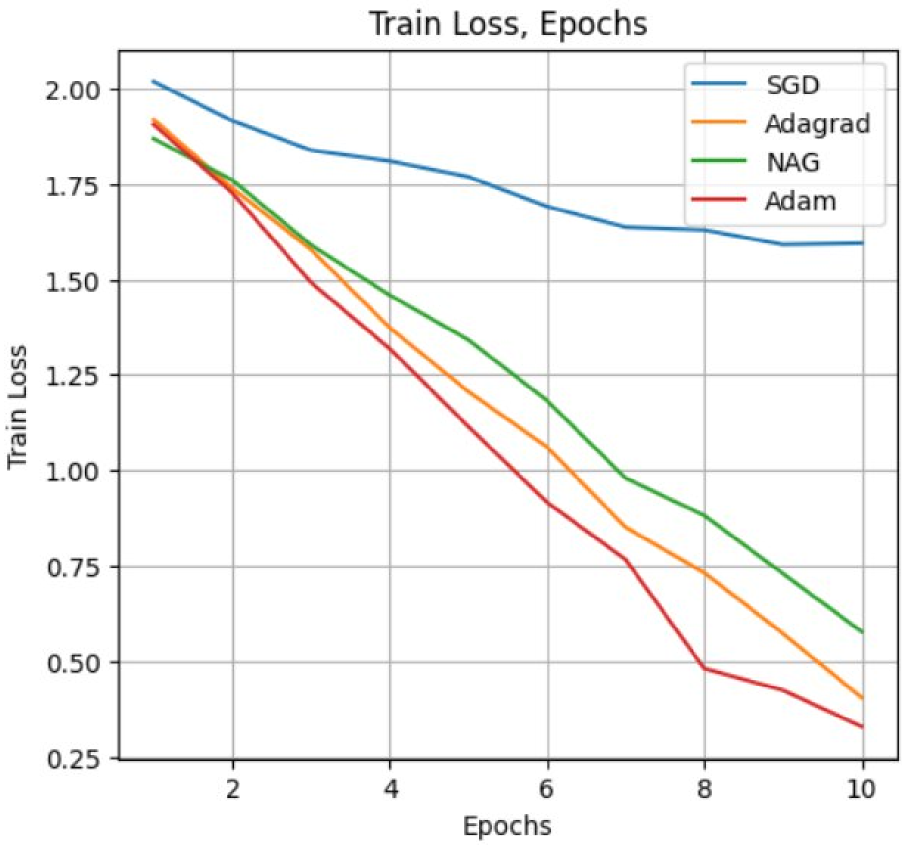

Рисунок 3 – Результаты VGG16 Loss

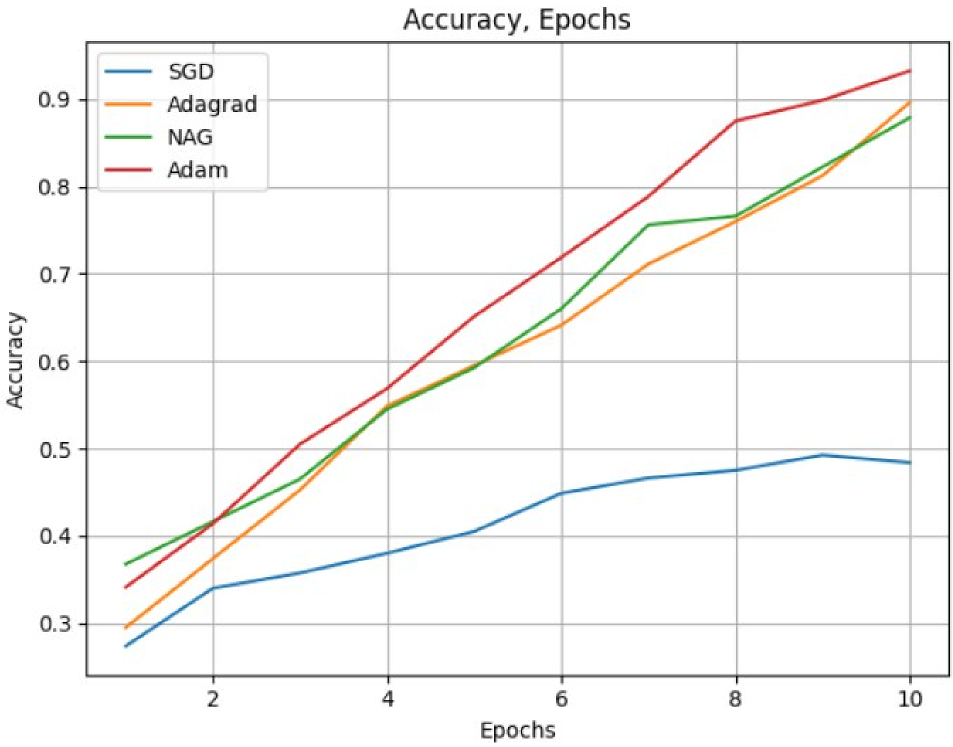

Рисунок 4 – Результаты VGG16 Accuracy

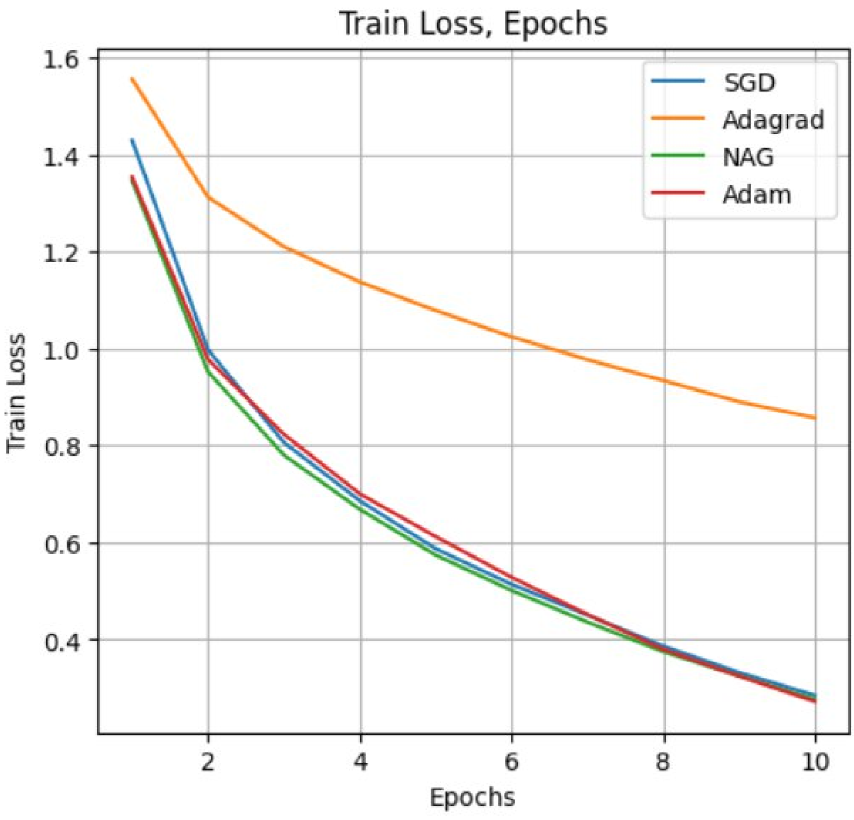

Рисунок 5 – Результаты ResNet Loss

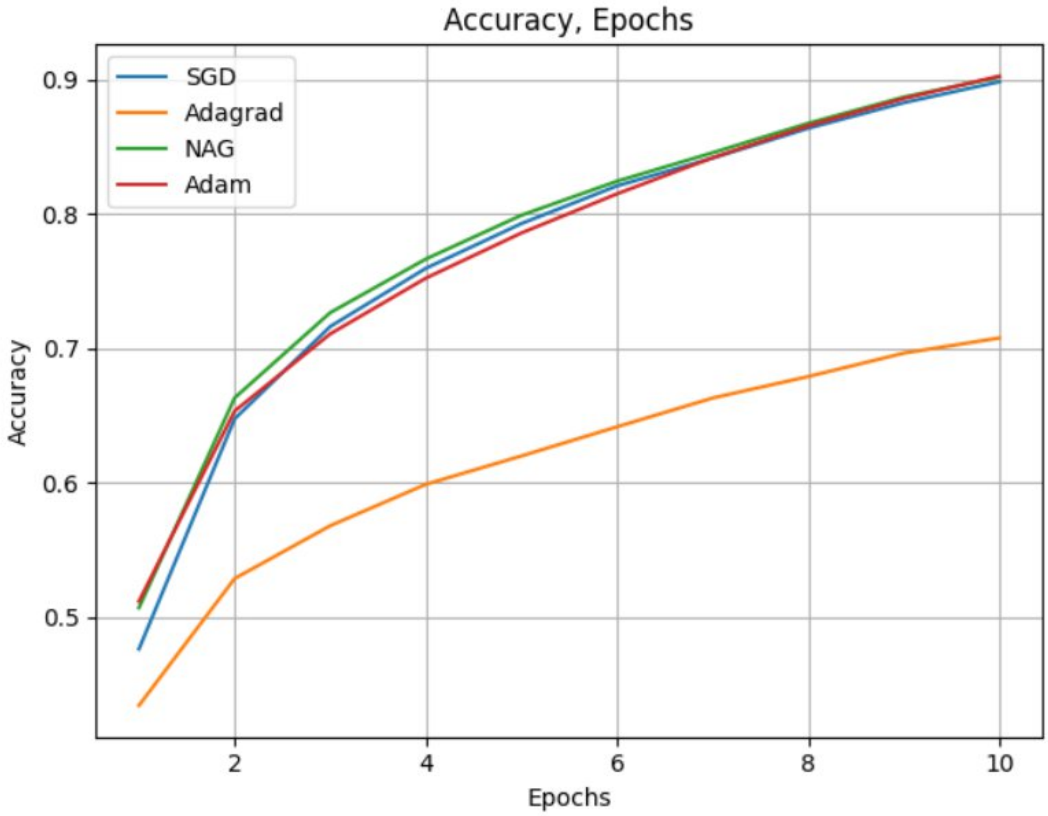

Рисунок 6 – Результаты ResNet Accuracy

5 Вывод

Были разработаны три модели сверточных нейронных сетей, и во всех случаях наилучшие результаты продемонстрировал оптимизатор Adam, что соответствует теоретическим ожиданиям.