Домашнее задание № 5: Реализация свёрточной нейронной сети

26 ноября 2024 г.

Сергей Виленский, ИУ9-72Б

Цель работы

Ознакомиться со сверточными нейронными сетями и исслеовать их эффективность при использовании разных гиперпараметов.

Постановка задачи

Требуется изучить особенности архитектур сверточных нейроных сетей (`LeNet`, `VGG16`, `ReNet34`). Сравить эффективность работы сетей для разных оптимизаторов (`SGD`, `AdaDelta`, `NAG`, `Adam`) и гиперпараметов (скорость обучения, кол-во эпох).

Реализация

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset

import numpy as np
import matplotlib.pyplot as plt


class CNN_default(nn.Module):
    def __init__(self, optimizer_type, learning_rate):
        super(CNN_default, self).__init__()
        self.optimizer_type = optimizer_type
        self.learning_rate = learning_rate
        self.loss_function = nn.CrossEntropyLoss()
```

```
    def forward(self, x):
        return x

    def fit(self, train_loader, epochs):
        optimizer = self.optimizer_type(self.parameters(), lr=self.learning_rate)
        self.train()

        for images, labels in train_loader:
            # Прямой проход
            outputs = self(images)
            loss = self.loss_function(outputs, labels)

            # Обратный проход и оптимизация
            loss.backward()
            optimizer.step()

    def evaluate(self, test_loader):
        correct = 0
        total = 0

        total_loss = 0

        self.eval()

        with torch.no_grad():
            for images, labels in test_loader:
                outputs = self(images)

                _, predicted = torch.max(outputs, 1)
                correct += (predicted == labels).sum().item()
                total += labels.size(0)

                loss = self.loss_function(outputs, labels)
                total_loss += loss.item()

        return correct / total, total_loss / len(test_loader)


class LeNet(CNN_default):
    def __init__(self, optimizer, learning_rate):
        super(LeNet, self).__init__(optimizer, learning_rate)

        self.conv1 = nn.Conv2d(3, 6, kernel_size=5, stride=1, padding=2)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
```

```
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(16 * 6 * 6, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16 * 6 * 6)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

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

transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.Grayscale(num_output_channels=3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

MNIST_train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=tra
MNIST_test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=tra

MNIST_train_loader = DataLoader(MNIST_train_dataset, batch_size=32, shuffle=True)
MNIST_test_loader = DataLoader(MNIST_test_dataset, batch_size=32, shuffle=False)

print(f"Размер обучающего датасета: {len(MNIST_train_dataset)}")
print(f"Размер тестового датасета: {len(MNIST_test_dataset)}")

LeNet_SGD_accuracies = []
LeNet_SGD_losses = []

LeNet_SGD_model = LeNet(optim.SGD, learning_rate=1e-6)
for _ in range(10):
    LeNet_SGD_model.fit(MNIST_train_loader, epochs=1)
    accurace, losses = LeNet_SGD_model.evaluate(MNIST_test_loader)
    LeNet_SGD_accuracies.append(accurace)
    LeNet_SGD_losses.append(losses)
    print(accurace, losses)
```

```
LeNet_AdaDelta_accuracies = []
LeNet_AdaDelta_losses = []

LeNet_AdaDelta_model = LeNet(optim.Adadelta, learning_rate=1e-4)
for _ in range(10):
    LeNet_AdaDelta_model.fit(MNIST_train_loader, epochs=1)
    accurace, losses = LeNet_AdaDelta_model.evaluate(MNIST_test_loader)
    LeNet_AdaDelta_accuracies.append(accurace)
    LeNet_AdaDelta_losses.append(losses)
    print(accurace, losses)

LeNet_NAG_accuracies = []
LeNet_NAG_losses = []

LeNet_NAG_model = LeNet(lambda params, lr: optim.SGD(params, lr=lr, momentum=.9, nesterov=Tr
for _ in range(10):
    LeNet_NAG_model.fit(MNIST_train_loader, epochs=1)
    accurace, losses = LeNet_NAG_model.evaluate(MNIST_test_loader)
    LeNet_NAG_accuracies.append(accurace)
    LeNet_NAG_losses.append(losses)
    print(accurace, losses)

LeNet_Adam_accuracies = []
LeNet_Adam_losses = []

LeNet_Adam_model = LeNet(optim.Adam, learning_rate=1e-5)
for _ in range(10):
    LeNet_Adam_model.fit(MNIST_train_loader, epochs=1)
    accurace, losses = LeNet_Adam_model.evaluate(MNIST_test_loader)
    LeNet_Adam_accuracies.append(accurace)
    LeNet_Adam_losses.append(losses)
    print(accurace, losses)

epoch_list = list(range(10))

plt.plot(epoch_list, LeNet_SGD_accuracies, label="SGD")
plt.plot(epoch_list, LeNet_AdaDelta_accuracies, label="AdaDelta")
plt.plot(epoch_list, LeNet_NAG_accuracies, label="NAG")
plt.plot(epoch_list, LeNet_Adam_accuracies, label="Adam")

plt.title("Графики зависимости точности моделей от использованного метода оптимизации для Le
plt.xlabel("Номер эпохи")
plt.ylabel("Точность модели")
plt.grid(True)
plt.legend()
plt.show()
```

```
plt.plot(epoch_list, LeNet_SGD_losses, label="SGD")
plt.plot(epoch_list, LeNet_AdaDelta_losses, label="AdaDelta")
plt.plot(epoch_list, LeNet_NAG_losses, label="NAG")
plt.plot(epoch_list, LeNet_Adam_losses, label="Adam")

plt.title("Графики зависимости функции потерь моделей от использованного метода оптимизации ")
plt.xlabel("Номер эпохи")
plt.ylabel("Значение функции потерь")
plt.grid(True)
plt.legend()
plt.show()

class VGG16(CNN_default):
    def __init__(self, optimizer, learning_rate):
        super(VGG16, self).__init__(optimizer, learning_rate)

        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=3, 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 * 3 * 3, 1024)
        self.dropout1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(1024, 1024)
        self.dropout2 = nn.Dropout(0.5)

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

    def forward(self, x):
        x = F.relu(self.conv1_1(x))
        x = F.relu(self.conv1_2(x))
        x = self.pool1(x)

        x = F.relu(self.conv2_1(x))
        x = F.relu(self.conv2_2(x))
        x = self.pool2(x)

        x = F.relu(self.conv3_1(x))
```

```
        x = F.relu(self.conv3_2(x))
        x = self.pool3(x)

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x))
        x = self.dropout1(x)
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)

        x = self.fc3(x)

        return x


transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

CIFAR10_train_dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform
CIFAR10_test_dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=

CIFAR10_train_loader = DataLoader(CIFAR10_train_dataset, batch_size=32, num_workers=6, shuff
CIFAR10_test_loader = DataLoader(CIFAR10_test_dataset, batch_size=32, num_workers=6, shuffle

print(f"Размер обучающего набора: {len(CIFAR10_train_dataset)}")
print(f"Размер тестового набора: {len(CIFAR10_test_dataset)}")

VGG16_SGD_accuracies = []
VGG16_SGD_losses = []

VGG16_SGD_model = VGG16(optim.SGD, learning_rate=1e-4)
for _ in range(10):
    VGG16_SGD_model.fit(CIFAR10_train_loader, epochs=1)
    accurace, losses = VGG16_SGD_model.evaluate(CIFAR10_test_loader)
    VGG16_SGD_accuracies.append(accurace)
    VGG16_SGD_losses.append(losses)
    print(accurace, losses)
```

```
VGG16_AdaDelta_accuracies = []
VGG16_AdaDelta_losses = []

VGG16_AdaDelta_model = VGG16(optim.Adadelta, learning_rate=1e-4)
for _ in range(10):
    VGG16_AdaDelta_model.fit(CIFAR10_train_loader, epochs=1)
    accurace, losses = VGG16_AdaDelta_model.evaluate(CIFAR10_test_loader)
    VGG16_AdaDelta_accuracies.append(accurace)
    VGG16_AdaDelta_losses.append(losses)
    print(accurace, losses)

VGG16_NAG_accuracies = []
VGG16_NAG_losses = []

VGG16_NAG_model = VGG16(lambda params, lr: optim.SGD(params, lr=lr, momentum=.9, nesterov=Tr
for _ in range(10):
    VGG16_NAG_model.fit(CIFAR10_train_loader, epochs=1)
    accurace, losses = VGG16_NAG_model.evaluate(CIFAR10_test_loader)
    VGG16_NAG_accuracies.append(accurace)
    VGG16_NAG_losses.append(losses)
    print(accurace, losses)

VGG16_Adam_accuracies = []
VGG16_Adam_losses = []

VGG16_Adam_model = VGG16(optim.Adam, learning_rate=1e-7)
for _ in range(10):
    VGG16_Adam_model.fit(CIFAR10_train_loader, epochs=1)
    accurace, losses = VGG16_Adam_model.evaluate(CIFAR10_test_loader)
    VGG16_Adam_accuracies.append(accurace)
    VGG16_Adam_losses.append(losses)
    print(accurace, losses)

plt.plot(epoch_list, VGG16_SGD_accuracies, label="SGD")
plt.plot(epoch_list, VGG16_AdaDelta_accuracies, label="AdaDelta")
plt.plot(epoch_list, VGG16_NAG_accuracies, label="NAG")
plt.plot(epoch_list, VGG16_Adam_accuracies, label="Adam")

plt.title("Графики зависимости точности моделей от использованного метода оптимизации для VG
plt.xlabel("Номер эпохи")
plt.ylabel("Точность модели")
plt.grid(True)
plt.legend()
plt.show()
```

```
plt.plot(epoch_list, VGG16_SGD_losses, label="SGD")
plt.plot(epoch_list, VGG16_AdaDelta_losses, label="AdaDelta")
plt.plot(epoch_list, VGG16_NAG_losses, label="NAG")
plt.plot(epoch_list, VGG16_Adam_losses, label="Adam")

plt.title("Графики зависимости функции потерь моделей от использованного метода оптимизации
plt.xlabel("Номер эпохи")
plt.ylabel("Значение функции потерь")
plt.grid(True)
plt.legend()
plt.show()

from sklearn.model_selection import train_test_split

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((.4772, .4505, .4168), (.2307, .2271, .2288))
])

ImageNet_dataset = datasets.ImageFolder(root='./data/imagenet', transform=transform)

train_indices, test_indices = train_test_split(list(range(len(ImageNet_dataset))), test_size

train_dataset = Subset(ImageNet_dataset, train_indices)
test_dataset = Subset(ImageNet_dataset, test_indices)

ImageNet_train_loader = DataLoader(train_dataset, num_workers=6, shuffle=True)
ImageNet_test_loader = DataLoader(test_dataset, num_workers=6, shuffle=False)

print(f"Размер обучающего набора: {len(ImageNet_train_loader)}")
print(f"Размер тестового набора: {len(ImageNet_test_loader)}")

ResNet34_SGD_accuracies = []
ResNet34_SGD_losses = []

ResNet34_SGD_model = ResNet34(optim.SGD, learning_rate=1e-7)
for _ in range(10):
    ResNet34_SGD_model.fit(ImageNet_train_loader, epochs=1)
    accurace, losses = ResNet34_SGD_model.evaluate(ImageNet_test_loader)
    ResNet34_SGD_accuracies.append(accurace)
    ResNet34_SGD_losses.append(losses)
    print(accurace, losses)

ResNet34_AdaDelta_accuracies = []
ResNet34_AdaDelta_losses = []
```

```
ResNet34_AdaDelta_model = ResNet34(optim.Adadelta, learning_rate=1e-4)
for _ in range(10):
    ResNet34_AdaDelta_model.fit(ImageNet_train_loader, epochs=1)
    accurace, losses = ResNet34_AdaDelta_model.evaluate(ImageNet_test_loader)
    ResNet34_AdaDelta_accuracies.append(accurace)
    ResNet34_AdaDelta_losses.append(losses)
    print(accurace, losses)

ResNet34_NAG_accuracies = []
ResNet34_NAG_losses = []

ResNet34_NAG_model = ResNet34(lambda params, lr: optim.SGD(params, lr=lr, momentum=.9, neste
for _ in range(10):
    ResNet34_NAG_model.fit(ImageNet_train_loader, epochs=1)
    accurace, losses = ResNet34_NAG_model.evaluate(ImageNet_test_loader)
    ResNet34_NAG_accuracies.append(accurace)
    ResNet34_NAG_losses.append(losses)
    print(accurace, losses)

ResNet34_Adam_accuracies = []
ResNet34_Adam_losses = []

ResNet34_Adam_model = ResNet34(optim.Adam, learning_rate=1e-5)
for _ in range(10):
    ResNet34_Adam_model.fit(ImageNet_train_loader, epochs=1)
    accurace, losses = ResNet34_Adam_model.evaluate(ImageNet_test_loader)
    ResNet34_Adam_accuracies.append(accurace)
    ResNet34_Adam_losses.append(losses)
    print(accurace, losses)

plt.plot(epoch_list, ResNet34_SGD_accuracies, label="SGD")
plt.plot(epoch_list, ResNet34_AdaDelta_accuracies, label="AdaDelta")
plt.plot(epoch_list, ResNet34_NAG_accuracies, label="NAG")
plt.plot(epoch_list, ResNet34_Adam_accuracies, label="Adam")

plt.title("Графики зависимости точности моделей от использованного метода оптимизации для Re
plt.xlabel("Номер эпохи")
plt.ylabel("Точность модели")
plt.grid(True)
plt.legend()
plt.show()

plt.plot(epoch_list, ResNet34_SGD_losses, label="SGD")
plt.plot(epoch_list, ResNet34_AdaDelta_losses, label="AdaDelta")
plt.plot(epoch_list, ResNet34_NAG_losses, label="NAG")
```

```
plt.plot(epoch_list, ResNet34_Adam_losses, label="Adam")

plt.title("Графики зависимости функции потерь моделей от использованного метода оптимизации
plt.xlabel("Номер эпохи")
plt.ylabel("Значение функции потерь")
plt.grid(True)
plt.legend()
plt.show()
```

Результаты

LeNet

	Оптимизатор	Кол-во эпох	Скорость обучения	Верность
1	SGD	10	1e-6	0.9346
2	AdaDelta	10	1e-4	0.9586
3	NAG	10	1e-7	0.8949
4	Adam	10	1e-5	0.9806

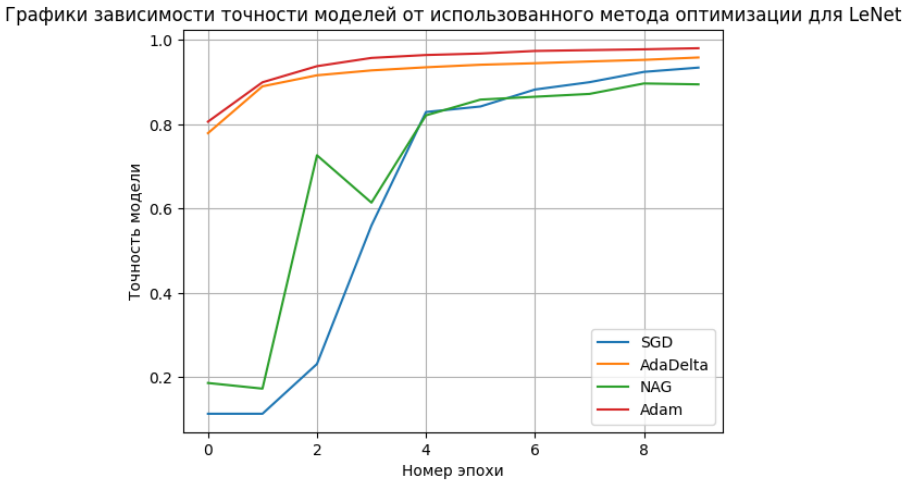


Figure 1: LeNet Сравнение точности

VGG16

Графики зависимости функции потерь моделей от использованного метода оптимизации для LeNet

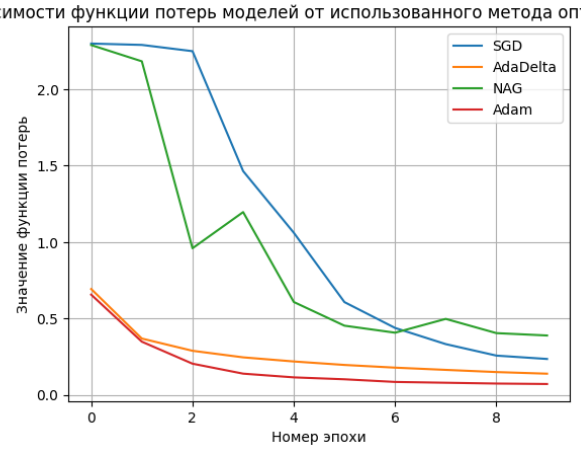


Figure 2: LeNet Сравнение лосс-функции

	Оптимизатор	Кол-во эпох	Скорость обучения	Dropout	Верность
1	SGD	4	1e-4	0.5	0.42054
2	AdaDelta	10	1e-4	0.5	0.62604
3	NAG	10	1e-6	0.5	0.26624
4	Adam	10	1e-7	0.5	0.2828

Графики зависимости точности моделей от использованного метода оптимизации для VGG16

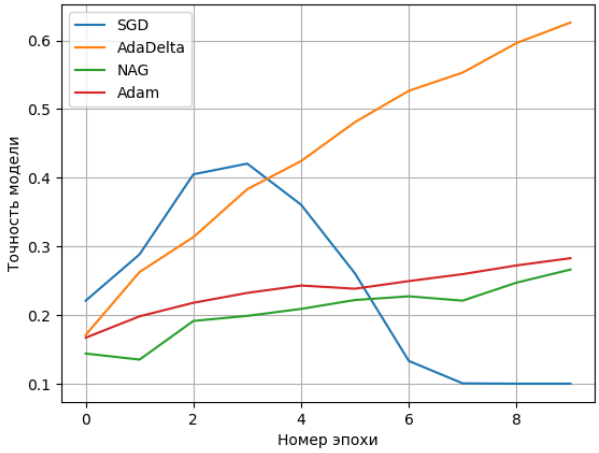
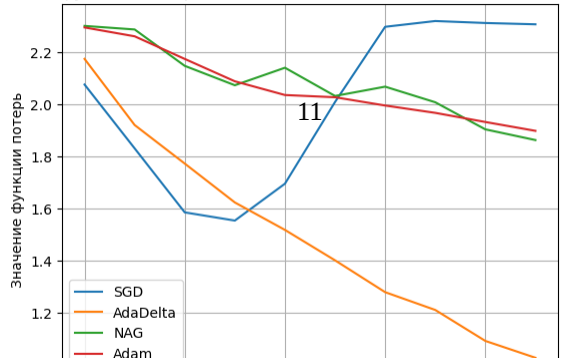


Figure 3: VGG16 Сравнение точности

Графики зависимости функции потерь моделей от использованного метода оптимизации для VGG16



	Оптимизатор	Кол-во эпох	Скорость обучения	Верность
1	SGD	8	1e-7	0.3142
2	AdaDelta	10	1e-4	0.3212
3	NAG	7	1e-8	0.2925
4	Adam	4	1e-5	0.3041

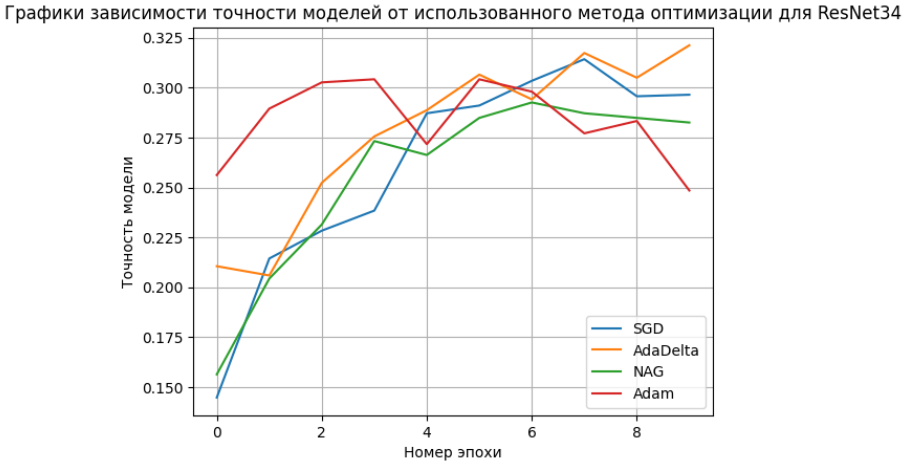


Figure 5: ResNet34 Сравнение точности

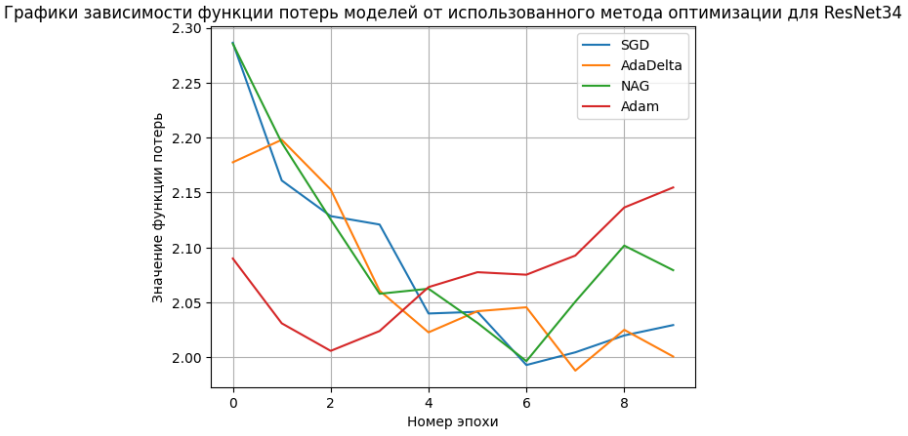


Figure 6: ResNet34 Сравнение лосс-функции

Выводы

В результате исследования эффективности работы разных архитектур сверточных нейронных сетей с использованием различных оптимизаторов было выявлено, что для сверточной нейронной сети с архитектурой `LeNet` больше всего подходят такие оптимизаторы, как `Adam` и `AdaDelta`, для `VGG16` - `AdaDelta` и для `ResNet34` - `AdaDelta` и `SGD`.