

Московский государственный технический университет им. Н.Э. Баумана  
Кафедра «Системы обработки информации и управления»



Домашнее задание  
по дисциплине  
«Методы машинного обучения»

Выполнил:  
студент группы ИУ5и-24М  
Аунг Пьюи Нанда

Москва — 2024 г.

## Задание

Домашнее задание по дисциплине направлено на анализ современных методов машинного обучения и их применение для решения практических задач. Домашнее задание включает три основных этапа:

1. выбор задачи;
2. теоретический этап;
3. практический этап.

Этап выбора задачи предполагает анализ ресурса [paperswithcode](https://paperswithcode.com/). Данный ресурс включает описание нескольких тысяч современных задач в области машинного обучения. Каждое описание задачи содержит ссылки на наиболее современные и актуальные научные статьи, предназначенные для решения задачи (список статей регулярно обновляется авторами ресурса). Каждое описание статьи содержит ссылку на репозиторий с открытым исходным кодом, реализующим представленные в статье эксперименты. На этапе выбора задачи обучающийся выбирает одну из задач машинного обучения, описание которой содержит ссылки на статьи и репозитории с исходным кодом.

Теоретический этап включает проработку как минимум двух статей, относящихся к выбранной задаче. Результаты проработки обучающийся излагает в теоретической части отчета по домашнему заданию, которая может включать:

- описание общих подходов к решению задачи;
- конкретные топологии нейронных сетей, нейросетевых ансамблей или других моделей машинного обучения, предназначенных для решения задачи;
- математическое описание, алгоритмы функционирования, особенности обучения используемых для решения задачи нейронных сетей, нейросетевых ансамблей или других моделей машинного обучения;
- описание наборов данных, используемых для обучения моделей;
- оценка качества решения задачи, описание метрик качества и их значений;
- предложения обучающегося по улучшению качества решения задачи.

Практический этап включает повторение экспериментов авторов статей на основе представленных авторами репозитория с исходным кодом и возможное улучшение обучающимися полученных результатов. Результаты

проработки обучающийся излагает в практической части отчета по домашнему заданию, которая может включать:

- исходные коды программ, представленные авторами статей, результаты документирования программ обучающимися с использованием диаграмм UML, путем визуализации топологий нейронных сетей и другими способами;
- результаты выполнения программ, вычисление значений для описанных в статьях метрик качества, выводы обучающегося о воспроизводимости экспериментов авторов статей и соответствии практических экспериментов теоретическим материалам статей;
- предложения обучающегося по возможным улучшениям решения задачи, результаты практических экспериментов (исходные коды, документация) по возможному улучшению решения задачи.

## **1. Постановку выбранной задачи машинного обучения, соответствующую этапу выбора задачи.**

Для выполнения домашнего задания выбраны две задачи машинного обучения в области классификации изображений на наборах данных CIFAR-100 и ImageNet. Обе задачи являются стандартными и широко изучаемыми в сообществе машинного обучения.

### **1) Постановка задачи машинного обучения: CIFAR-100 Image Classification**

Разработать и обучить модель машинного обучения для классификации изображений на наборе данных CIFAR-100 с высокой точностью и обобщающей способностью.

Набор данных CIFAR-100 содержит 60 000 цветных изображений размером 32x32 пикселя, разделенных на 100 классов, по 600 изображений на класс. Эти классы включают 20 широких категорий (например, рыбы, птицы, инсектоеды) и по 5 подкатегорий в каждой.

Методы:

Использование сверточных нейронных сетей (CNN) является стандартным подходом для классификации изображений на CIFAR-100. Другие возможные методы включают в себя передаточное обучение, аугментацию данных и оптимизацию гиперпараметров.

Ожидаемые результаты:

Высокая точность классификации на тестовом наборе данных. Эффективное обобщение на новые изображения из реального мира. Воспроизводимость результатов, достигнутых в существующих исследованиях по CIFAR-100.

Метрики качества:

Основные метрики, используемые для оценки качества модели, включают в себя Accuracy, Precision, Recall и F1-score. Также может быть полезно изучить кривые обучения и валидации, чтобы оценить процесс обучения.

Этапы решения задачи:

- 1) Подготовка данных: Загрузка, предварительная обработка и разделение данных на обучающий, валидационный и тестовый наборы.
- 2) Разработка модели: Выбор архитектуры нейронной сети, определение слоев и гиперпараметров модели.
- 3) Обучение модели: Обучение модели на обучающем наборе данных с использованием выбранного алгоритма оптимизации.
- 4) Оценка модели: Оценка производительности модели на валидационном и тестовом наборах данных.
- 5) Улучшение модели: Тюнинг гиперпараметров, аугментация данных и другие методы для улучшения качества модели.

## **2) Постановка задачи машинного обучения для классификации изображений на наборе данных CIFAR-10**

Разработать модель машинного обучения, способную автоматически классифицировать изображения на десять различных категорий.

CIFAR-10 - это набор данных, состоящий из 60 000 цветных изображений размером 32x32 пикселя, разделенных на 10 классов: самолеты, автомобили, птицы, кошки, олени, собаки, лягушки, лошади, корабли и грузовики.

Анализ существующих методов:

Методы глубокого обучения, такие как сверточные нейронные сети (CNN), являются стандартом для решения задачи классификации изображений на CIFAR-10. Мы сосредоточимся на изучении таких архитектур как ResNet и Wide Residual Networks (WRN), известных своей эффективностью на этом наборе данных.

Метрики качества:

Для оценки качества модели будем использовать точность классификации (accuracy), которая измеряет процент изображений, верно классифицированных моделью.

Предложения по улучшению:

После анализа существующих методов и экспериментов с ними мы можем предложить улучшения, такие как изменение архитектуры сети, оптимизацию гиперпараметров, аугментацию данных или использование передовых методов, таких как ансамблирование или обучение с подкреплением.

Таким образом, основной задачей нашего исследования является разработка и сравнение различных архитектур нейронных сетей для классификации изображений на наборе данных CIFAR-10 с целью достижения наивысшей возможной точности классификации.

## **2. Теоретическую часть**

Общие подходы к классификации изображений:

### **1. Сверточные нейронные сети (CNN)**

CNN являются наиболее распространенным и эффективным методом для классификации изображений. Они способны извлекать признаки изображений на разных уровнях абстракции благодаря своей архитектуре, которая включает сверточные слои, слои пулинга и полносвязные слои. Примеры популярных архитектур CNN включают в себя AlexNet, VGG, ResNet, Inception и EfficientNet.

### **2. Передаточное обучение (Transfer Learning)**

Передаточное обучение позволяет использовать заранее обученные модели на больших наборах данных (например, ImageNet) для решения новых задач. Это особенно полезно, когда у нас есть ограниченное количество размеченных данных для новой задачи. Стандартным подходом является

замораживание весов предварительно обученных слоев и дообучение только последних слоев на новых данных.

### **3. Аугментация данных (Data Augmentation)**

Аугментация данных позволяет увеличить разнообразие обучающего набора данных путем применения различных преобразований к изображениям, таких как повороты, отражения, изменения размера и изменения яркости. Это помогает улучшить обобщающую способность модели и справиться с проблемой переобучения.

### **Конкретные топологии нейронных сетей**

ResNet (Residual Networks):

Введение блоков-остатков позволяет эффективно обучать глубокие нейронные сети, минимизируя проблему затухающих градиентов.

Inception:

Использование нескольких параллельных операций свертки с разными размерами ядер позволяет эффективно извлекать признаки изображений на разных уровнях.

EfficientNet:

Эффективная масштабируемость архитектуры путем коэффициентного масштабирования параметров сети, ширины и глубины, что позволяет добиться лучшей производительности при заданных ограничениях ресурсов.

### **Математическое описание и алгоритмы функционирования**

Сверточные слои:

Применение операции свертки к входным изображениям с использованием ядра свертки для выделения локальных признаков.

Пулинг слои:

Уменьшение размерности признаков карт путем выбора значений с определенными интервалами.

Полносвязные слои:

Соединение признаков, извлеченных из предыдущих слоев, с последующими классификационными слоями.

## Описание набора данных Cifar100 и Cifar 10

Несколько базовых формул, которые могут быть полезны при описании теоретической части отчета по задаче классификации изображений на наборе данных Cifar100 и Cifar 10:

### 1. Функция активации (Activation Function)

Сигмоидная функция:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Гиперболический тангенс (tanh):

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

ReLU (Rectified Linear Unit):

$$\text{ReLU}(z) = \max(0, z)$$

### 2. Функция потерь (Loss Function)

Кросс-энтропия (Cross-Entropy):

$$H(y, \hat{y}) = - \sum_i y_i \log(\hat{y}_i)$$

### 3. Оптимизатор (Optimizer)

Градиентный спуск (Gradient Descent):

$$\theta = \theta - \alpha \nabla J(\theta)$$

где  $\theta$  - параметры модели,  $\alpha$  - скорость обучения,  $J(\theta)$  - функция потерь.

### 4. Метрики оценки (Evaluation Metrics)



Accuracy (Точность):

$$\text{Accuracy} = \frac{\text{Количество правильных предсказаний}}{\text{Общее количество предсказаний}}$$

Precision (Точность):

$$\text{Precision} = \frac{TP}{TP+FP}$$

Recall (Полнота):

$$\text{Recall} = \frac{TP}{TP+FN}$$

F1-score (F1-мера):

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$



### 3. Практическую часть

В данной практической части отчета проводится экспериментальное исследование задач классификации изображений на наборах данных CIFAR-100 и ImageNet, используя описанные в теоретической части архитектуры нейронных сетей: Wide Residual Networks (WRN) и DenseNet для CIFAR-100 и Cifar-10. Включены результаты запуска исходных кодов, документирование программ, анализ полученных результатов, а также предложения по улучшению моделей и их реализация.

#### 1. Классификация изображений на наборе данных CIFAR-100

Wide Residual Networks (WRN)

```
[1] import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
import math
```

```
class BasicBlock(nn.Module):
    def __init__(self, in_planes, out_planes, stride, dropRate=0.0):
        super(BasicBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                                padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_planes)
        self.conv2 = nn.Conv2d(out_planes, out_planes, kernel_size=3, stride=1,
                                padding=1, bias=False)
        self.droprate = dropRate
        self.equalInOut = (in_planes == out_planes)
        self.convShortcut = (not self.equalInOut) and nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride,
                                                                    padding=0, bias=False) or None

    def forward(self, x):
        out = self.conv1(self.relu(self.bn1(x)))
        out = self.conv2(self.relu(self.bn2(out)))
        if self.equalInOut:
            return torch.add(x, out)
        else:
            return torch.add(self.convShortcut(x), out)
```

```
class NetworkBlock(nn.Module):
    def __init__(self, nb_layers, in_planes, out_planes, block, stride, dropRate=0.0):
        super(NetworkBlock, self).__init__()
        self.layer = self._make_layer(block, in_planes, out_planes, nb_layers, stride, dropRate)

    def _make_layer(self, block, in_planes, out_planes, nb_layers, stride, dropRate):
        layers = []
        for i in range(nb_layers):
            layers.append(block(i == 0 and in_planes or out_planes, out_planes, i == 0 and stride or 1, dropRate))
        return nn.Sequential(*layers)

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

```
[4] class NetworkBlock(nn.Module):
    def __init__(self, nb_layers, in_planes, out_planes, block, stride, dropRate=0.0):
        super(NetworkBlock, self).__init__()
        self.layer = self._make_layer(block, in_planes, out_planes, nb_layers, stride, dropRate)
```

```
[5] def _make_layer(self, block, in_planes, out_planes, nb_layers, stride, dropRate):
    layers = []
    for i in range(nb_layers):
        layers.append(block(i == 0 and in_planes or out_planes, out_planes, i == 0 and stride or 1, dropRate))
    return nn.Sequential(*layers)
```

```
[6] def forward(self, x):
    return self.layer(x)
```

```

class WideResNet(nn.Module):
    def __init__(self, depth, num_classes, widen_factor=1, dropRate=0.0):
        super(WideResNet, self).__init__()
        nChannels = [16, 16*widen_factor, 32*widen_factor, 64*widen_factor]
        assert ((depth - 4) % 6 == 0)
        n = (depth - 4) // 6
        block = BasicBlock
        self.conv1 = nn.Conv2d(3, nChannels[0], kernel_size=3, stride=1, padding=1, bias=False)
        self.block1 = NetworkBlock(n, nChannels[0], nChannels[1], block, 1, dropRate)
        self.block2 = NetworkBlock(n, nChannels[1], nChannels[2], block, 2, dropRate)
        self.block3 = NetworkBlock(n, nChannels[2], nChannels[3], block, 2, dropRate)
        self.bn1 = nn.BatchNorm2d(nChannels[3])
        self.relu = nn.ReLU(inplace=True)
        self.fc = nn.Linear(nChannels[3], num_classes)
        self.nChannels = nChannels[3]

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        out = self.conv1(x)
        out = self.block1(out)
        out = self.block2(out)
        out = self.block3(out)
        out = self.relu(self.bn1(out))
        out = nn.functional.avg_pool2d(out, 8)
        out = out.view(-1, self.nChannels)
        return self.fc(out)

```

```

# Преобразования для набора данных
transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761)),
])

```

```

[9] # Загрузка данных
trainset = torchvision.datasets.CIFAR100(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)

```

Files already downloaded and verified

```

[11] # Устройство для вычислений
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

```

```

class NetworkBlock(nn.Module):
    def __init__(self, nb_layers, in_planes, out_planes, block, stride, dropRate=0.0):
        super(NetworkBlock, self).__init__()
        self.layer = self._make_layer(block, in_planes, out_planes, nb_layers, stride, dropRate)

    def _make_layer(self, block, in_planes, out_planes, nb_layers, stride, dropRate):
        layers = []
        for i in range(nb_layers):
            layers.append(block(i == 0 and in_planes or out_planes, out_planes, i == 0 and stride or 1, dropRate))
        return nn.Sequential(*layers)

    def forward(self, x):
        return self.layer(x)

```

```

[13] # Создание модели
net = WideResNet(depth=28, num_classes=100, widen_factor=10, dropRate=0.3).to(device)

```

```

[14] # Определение функции потерь и оптимизатора
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=60, gamma=0.2)

```

```

✓ 42m # Обучение модели
for epoch in range(10):
    net.train()
    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()
    scheduler.step()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}")

```

```

⇒ Epoch 1, Loss: 3.9565042229869483
Epoch 2, Loss: 3.40515643495428
Epoch 3, Loss: 3.0147732389552515
Epoch 4, Loss: 2.695617649561304
Epoch 5, Loss: 2.433935480959275
Epoch 6, Loss: 2.2085251091691234
Epoch 7, Loss: 2.0194114940550625
Epoch 8, Loss: 1.860038174387744
Epoch 9, Loss: 1.7203145255822965
Epoch 10, Loss: 1.5871508475154867

```

```

✓ 2s # Evaluation of the model
net.eval()
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
print(f"Test Accuracy: {100 * correct / total}%")

```

⇒ Test Accuracy: 45.34%

## DenseNet на CIFAR-100

```

✓ 0s import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
from torchvision import models

```

```

✓ 0s [22] # Определение модели DenseNet
class Bottleneck(nn.Module):
    def __init__(self, nChannels, growthRate):
        super(Bottleneck, self).__init__()
        interChannels = 4 * growthRate
        self.bn1 = nn.BatchNorm2d(nChannels)
        self.conv1 = nn.Conv2d(nChannels, interChannels, kernel_size=1, bias=False)
        self.bn2 = nn.BatchNorm2d(interChannels)
        self.conv2 = nn.Conv2d(interChannels, growthRate, kernel_size=3, padding=1, bias=False)

```

```

✓ 0s [23] def forward(self, x):
    out = self.conv1(self.bn1(x))
    out = self.conv2(self.bn2(out))
    out = torch.cat([x, out], 1)
    return out

```

```

[24] class Transition(nn.Module):
    def __init__(self, nChannels, nOutChannels):
        super(Transition, self).__init__()
        self.bn1 = nn.BatchNorm2d(nChannels)
        self.conv1 = nn.Conv2d(nChannels, nOutChannels, kernel_size=1, bias=False)
        self.pool = nn.AvgPool2d(2)

    def forward(self, x):
        out = self.conv1(self.bn1(x))
        out = self.pool(out)
        return out

```

```

class DenseNet(nn.Module):
    def __init__(self, growthRate, depth, reduction, nClasses):
        super(DenseNet, self).__init__()
        nDenseBlocks = (depth - 4) // 3
        nChannels = 2 * growthRate
        self.conv1 = nn.Conv2d(3, nChannels, kernel_size=3, padding=1, bias=False)
        self.dense1 = self._make_dense(nChannels, growthRate, nDenseBlocks)
        nChannels += nDenseBlocks * growthRate
        nOutChannels = int(math.floor(nChannels * reduction))
        self.trans1 = Transition(nChannels, nOutChannels)
        nChannels = nOutChannels
        self.dense2 = self._make_dense(nChannels, growthRate, nDenseBlocks)
        nChannels += nDenseBlocks * growthRate
        nOutChannels = int(math.floor(nChannels * reduction))
        self.trans2 = Transition(nChannels, nOutChannels)
        nChannels = nOutChannels
        self.dense3 = self._make_dense(nChannels, growthRate, nDenseBlocks)
        nChannels += nDenseBlocks * growthRate
        self.bn1 = nn.BatchNorm2d(nChannels)
        self.fc = nn.Linear(nChannels, nClasses)

    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.Linear):
            nn.init.constant_(m.bias, 0)

```

```

def _make_dense(self, nChannels, growthRate, nDenseBlocks):
    layers = []
    for i in range(nDenseBlocks):
        layers.append(Bottleneck(nChannels, growthRate))
        nChannels += growthRate
    return nn.Sequential(*layers)

```

```

[27] def forward(self, x):
    out = self.conv1(x)
    out = self.trans1(self.dense1(out))
    out = self.trans2(self.dense2(out))
    out = self.bn1(self.dense3(out))
    out = nn.functional.avg_pool2d(out, 8)
    out = out.view(out.size(0), -1)
    return self.fc(out)

```

```

[28] transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761)),
])

```

```
✓ 0s [29] trainset = torchvision.datasets.CIFAR100(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)

Files already downloaded and verified

✓ 0s testset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=2)

Files already downloaded and verified

✓ 0s [32] # Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the DenseNet model
net = models.densenet121(pretrained=False, num_classes=100).to(device)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=60, gamma=0.2)

for epoch in range(10):
    net.train()
    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()
    scheduler.step()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}")

Epoch 1, Loss: 3.8885617207383256
Epoch 2, Loss: 3.298895679776321
Epoch 3, Loss: 2.9824776862893265
Epoch 4, Loss: 2.7500028482178593
Epoch 5, Loss: 2.569366424589816
Epoch 6, Loss: 2.42239154238835
Epoch 7, Loss: 2.2773823860051383
Epoch 8, Loss: 2.166843742055966
Epoch 9, Loss: 2.05681072689993
Epoch 10, Loss: 1.9590622744596828

net.eval()
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
    print(f"Test Accuracy: {100 * correct / total}%")

Test Accuracy: 42.04%
```

Результаты выполнения программ

**WRN на CIFAR-100**

Точность на тестовой выборке: 45.34%

Ошибка классификации: 1.59%

## DenseNet на CIFAR-100

Точность на тестовой выборке: 42.04%

Ошибка классификации: 1.96%

### Анализ результатов

Модели WRN и DenseNet показали результаты на наборе данных CIFAR-100, что подтверждает их эффективность. DenseNet немного превосходит WRN по точности, что может быть связано с более плотными соединениями, улучшающими распространение информации и градиентов.

## 2. Классификация изображений на наборе данных Cifar-10

### Wide Residual Networks (WRN)

```
[6] import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
import math
```

```
class BasicBlock(nn.Module):
    def __init__(self, in_planes, out_planes, stride, dropRate=0.0):
        super(BasicBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                                padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_planes)
        self.conv2 = nn.Conv2d(out_planes, out_planes, kernel_size=3, stride=1,
                                padding=1, bias=False)
        self.dropRate = dropRate
        self.equalInOut = (in_planes == out_planes)
        self.convShortcut = (not self.equalInOut) and nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride,
                                                                    padding=0, bias=False) or None

    def forward(self, x):
        out = self.conv1(self.relu(self.bn1(x)))
        out = self.conv2(self.relu(self.bn2(out)))
        if self.equalInOut:
            return torch.add(x, out)
        else:
            return torch.add(self.convShortcut(x), out)
```

```

[8] class NetworkBlock(nn.Module):
    def __init__(self, nb_layers, in_planes, out_planes, block, stride, dropRate=0.0):
        super(NetworkBlock, self).__init__()
        self.layer = self._make_layer(block, in_planes, out_planes, nb_layers, stride, dropRate)

    def _make_layer(self, block, in_planes, out_planes, nb_layers, stride, dropRate):
        layers = []
        for i in range(nb_layers):
            layers.append(block(i == 0 and in_planes or out_planes, out_planes, i == 0 and stride or 1, dropRate))
        return nn.Sequential(*layers)

    def forward(self, x):
        return self.layer(x)

```

```

class NetworkBlock(nn.Module):
    def __init__(self, nb_layers, in_planes, out_planes, block, stride, dropRate=0.0):
        super(NetworkBlock, self).__init__()
        self.layer = self._make_layer(block, in_planes, out_planes, nb_layers, stride, dropRate)

```

```

[10] def _make_layer(self, block, in_planes, out_planes, nb_layers, stride, dropRate):
    layers = []
    for i in range(nb_layers):
        layers.append(block(i == 0 and in_planes or out_planes, out_planes, i == 0 and stride or 1, dropRate))
    return nn.Sequential(*layers)

```

```

[11] def forward(self, x):
    return self.layer(x)

```

```

class WideResNet(nn.Module):
    def __init__(self, depth, num_classes, widen_factor=1, dropRate=0.0):
        super(WideResNet, self).__init__()
        nChannels = [16, 16*widen_factor, 32*widen_factor, 64*widen_factor]
        assert ((depth - 4) % 6 == 0)
        n = (depth - 4) // 6
        block = BasicBlock
        self.conv1 = nn.Conv2d(3, nChannels[0], kernel_size=3, stride=1, padding=1, bias=False)
        self.block1 = NetworkBlock(n, nChannels[0], nChannels[1], block, 1, dropRate)
        self.block2 = NetworkBlock(n, nChannels[1], nChannels[2], block, 2, dropRate)
        self.block3 = NetworkBlock(n, nChannels[2], nChannels[3], block, 2, dropRate)
        self.bn1 = nn.BatchNorm2d(nChannels[3])
        self.relu = nn.ReLU(inplace=True)
        self.fc = nn.Linear(nChannels[3], num_classes)
        self.nChannels = nChannels[3]

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        out = self.conv1(x)
        out = self.block1(out)
        out = self.block2(out)
        out = self.block3(out)
        out = self.relu(self.bn1(out))
        out = nn.functional.avg_pool2d(out, 8)
        out = out.view(-1, self.nChannels)
        return self.fc(out)

```

```


[13] # Предобработка данных
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

```

```

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


```

 Downloading <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz> to ./data/cifar-10-python.tar.gz  
 100% | 170498071/170498071 [00:02<00:00, 79621774.83it/s]  
 Extracting ./data/cifar-10-python.tar.gz to ./data

```

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

```

 Files already downloaded and verified

```

# Устройство для вычислений
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

```



```

[16] # Устройство для вычислений
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

[17] class NetworkBlock(nn.Module):
    def __init__(self, nb_layers, in_planes, out_planes, block, stride, dropRate=0.0):
        super(NetworkBlock, self).__init__()
        self.layer = self._make_layer(block, in_planes, out_planes, nb_layers, stride, dropRate)

    def _make_layer(self, block, in_planes, out_planes, nb_layers, stride, dropRate):
        layers = []
        for i in range(nb_layers):
            layers.append(block(i == 0 and in_planes or out_planes, out_planes, i == 0 and stride or 1, dropRate))
        return nn.Sequential(*layers)

    def forward(self, x):
        return self.layer(x)

[18] # Создание модели
net = WideResNet(depth=28, num_classes=100, widen_factor=10, dropRate=0.3).to(device)

[19] # Определение функции потерь и оптимизатора
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=60, gamma=0.2)

```

```

# Обучение модели
for epoch in range(10):
    net.train()
    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()
    scheduler.step()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}")

```

Epoch 1, Loss: 1.3994203514760108  
 Epoch 2, Loss: 0.9079497391000733  
 Epoch 3, Loss: 0.6933529624701156  
 Epoch 4, Loss: 0.5416872582929518  
 Epoch 5, Loss: 0.4311249174768358  
 Epoch 6, Loss: 0.3437807446207537  
 Epoch 7, Loss: 0.2725122553746566  
 Epoch 8, Loss: 0.21877645567783613  
 Epoch 9, Loss: 0.16838963350990926  
 Epoch 10, Loss: 0.13453845871979242

```

[15] # Evaluation of the model
net.eval()
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
    print(f"Test Accuracy: {100 * correct / total}%")

```

Test Accuracy: 80.45%

## DenseNet на CIFAR-10

```
✓ [59] import torch
0s      import torchvision
      import torchvision.transforms as transforms
      import torch.nn as nn
      import torch.optim as optim
      from torchvision import models
```

```
✓ [60] # Определение модели DenseNet
0s      class Bottleneck(nn.Module):
      def __init__(self, nChannels, growthRate):
          super(Bottleneck, self).__init__()
          interChannels = 4 * growthRate
          self.bn1 = nn.BatchNorm2d(nChannels)
          self.conv1 = nn.Conv2d(nChannels, interChannels, kernel_size=1, bias=False)
          self.bn2 = nn.BatchNorm2d(interChannels)
          self.conv2 = nn.Conv2d(interChannels, growthRate, kernel_size=3, padding=1, bias=False)
```

```
✓ [61] def forward(self, x):
0s      out = self.conv1(self.bn1(x))
      out = self.conv2(self.bn2(out))
      out = torch.cat([x, out], 1)
      return out
```

```
✓ [62] class Transition(nn.Module):
0s      def __init__(self, nChannels, nOutChannels):
      super(Transition, self).__init__()
      self.bn1 = nn.BatchNorm2d(nChannels)
      self.conv1 = nn.Conv2d(nChannels, nOutChannels, kernel_size=1, bias=False)
      self.pool = nn.AvgPool2d(2)

      def forward(self, x):
          out = self.conv1(self.bn1(x))
          out = self.pool(out)
          return out
```

```

0s [63] class DenseNet(nn.Module):
    def __init__(self, growthRate, depth, reduction, nClasses):
        super(DenseNet, self).__init__()
        nDenseBlocks = (depth - 4) // 3
        nChannels = 2 * growthRate
        self.conv1 = nn.Conv2d(3, nChannels, kernel_size=3, padding=1, bias=False)
        self.dense1 = self._make_dense(nChannels, growthRate, nDenseBlocks)
        nChannels += nDenseBlocks * growthRate
        nOutChannels = int(math.floor(nChannels * reduction))
        self.trans1 = Transition(nChannels, nOutChannels)
        nChannels = nOutChannels
        self.dense2 = self._make_dense(nChannels, growthRate, nDenseBlocks)
        nChannels += nDenseBlocks * growthRate
        nOutChannels = int(math.floor(nChannels * reduction))
        self.trans2 = Transition(nChannels, nOutChannels)
        nChannels = nOutChannels
        self.dense3 = self._make_dense(nChannels, growthRate, nDenseBlocks)
        nChannels += nDenseBlocks * growthRate
        self.bn1 = nn.BatchNorm2d(nChannels)
        self.fc = nn.Linear(nChannels, nClasses)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)

```

```

0s [64] def _make_dense(self, nChannels, growthRate, nDenseBlocks):
    layers = []
    for i in range(nDenseBlocks):
        layers.append(Bottleneck(nChannels, growthRate))
        nChannels += growthRate
    return nn.Sequential(*layers)

```

```

0s [65] def forward(self, x):
    out = self.conv1(x)
    out = self.trans1(self.dense1(out))
    out = self.trans2(self.dense2(out))
    out = self.bn1(self.dense3(out))
    out = nn.functional.avg_pool2d(out, 8)
    out = out.view(out.size(0), -1)
    return self.fc(out)

```

```


0s [66] # Предобработка данных
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

```

```

7s [67] trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

```

 Downloading <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz> to ./data/cifar-10-python.tar.gz  
 100%|██████████| 170498071/170498071 [00:03<00:00, 49339002.88it/s]  
 Extracting ./data/cifar-10-python.tar.gz to ./data

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

Files already downloaded and verified

```
[71] # Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the DenseNet model
net = models.densenet121(pretrained=False, num_classes=100).to(device)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=60, gamma=0.2)
```

```
[72] for epoch in range(10):
    net.train()
    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()
    scheduler.step()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}")
```

Epoch 1, Loss: 1.5637220420953257  
Epoch 2, Loss: 1.1302182935083005  
Epoch 3, Loss: 0.9065015601837422  
Epoch 4, Loss: 0.7574080247098528  
Epoch 5, Loss: 0.6389941974445377  
Epoch 6, Loss: 0.5422828198241456  
Epoch 7, Loss: 0.4687292457503431  
Epoch 8, Loss: 0.4141872313321399  
Epoch 9, Loss: 0.3595283222587212  
Epoch 10, Loss: 0.30874331353608603

```
✓ [73] net.eval()
7s correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
    print(f"Test Accuracy: {100 * correct / total}%")
```

Test Accuracy: 74.62%

Результаты выполнения программ

**WRN на CIFAR-10**

Точность на тестовой выборке: 80.45%

Ошибка классификации: 0.13%

## **DenseNet на CIFAR-10**

Точность на тестовой выборке: 74.62%

Ошибка классификации: 0.3%

### **Анализ результатов**

Модели WRN и DenseNet показали результаты на наборе данных CIFAR-100, что подтверждает их эффективность. DenseNet немного превосходит WRN по точности, что может быть связано с более плотными соединениями, улучшающими распространение информации и градиентов.

## **Выводы**

Wide Residual Networks (WRN) продемонстрировали лучшую производительность по сравнению с DenseNet на наборе данных CIFAR-100. Точность WRN составила 45.34%, что на 3.3% выше, чем у DenseNet. Ошибка классификации у WRN также ниже (1.59 против 1.96 у DenseNet), что свидетельствует о более высокой способности WRN различать классы в этом более сложном наборе данных.

На более простом наборе данных CIFAR-10 WRN снова показали лучшие результаты с точностью 80.45%, что на 5.83% выше, чем у DenseNet. Ошибка классификации у WRN (0.13) заметно ниже, чем у DenseNet (0.3), что подчеркивает лучшую производительность WRN для задач классификации на CIFAR-10.

Wide Residual Networks (WRN) демонстрируют более высокую точность и меньшую ошибку классификации как на CIFAR-100, так и на CIFAR-10 по сравнению с DenseNet. Это может быть связано с архитектурными особенностями WRN, такими как более широкие слои и остаточные соединения, которые позволяют модели лучше обучаться и избегать затухания градиентов. DenseNet также показывает хорошую производительность, но немного уступает WRN.

Можно попробовать различные настройки гиперпараметров, такие как скорость обучения, размер батча и число эпох, чтобы улучшить производительность моделей. Применение более сложных техник аугментации данных может помочь улучшить способность моделей обобщать и, соответственно, повысить точность. Применение методов регуляризации, таких как Dropout или L2-регуляризация, может помочь избежать переобучения и улучшить общую производительность моделей.

## **Список литературы**

- [1] Гапанюк Ю. Е. Домашнее задание по дисциплине «Методы машинного обучения»[Электронный ресурс] // GitHub. — 2019. — Режим доступа: [https://github.com/ugapanyuk/courses\\_current/wiki/DZ\\_MMO](https://github.com/ugapanyuk/courses_current/wiki/DZ_MMO).
- [2] You are my Sunshine [Electronic resource] // Space Apps Challenge. — 2017. Access mode: <https://2017.spaceappschallenge.org/challenges/earth-and-us/you-are-my-sunshine/details> (online; accessed: 22.02.2019).
- [3] dronio. Solar Radiation Prediction [Electronic resource] // Kaggle. — 2017. — Access mode: <https://www.kaggle.com/dronio/SolarEnergy> (online; accessed: 18.02.2019).
- [4] Team The IPython Development. IPython 7.3.0 Documentation [Electronic resource] //Read the Docs. — 2019. — Access mode: <https://ipython.readthedocs.io/en/stable/> (online; accessed: 20.02.2019).
- [5] Waskom M. seaborn 0.9.0 documentation [Electronic resource] // PyData. — 2018. Access mode: <https://seaborn.pydata.org/> (online; accessed: 20.02.2019).
- [6] pandas 0.24.1 documentation [Electronic resource] // PyData. — 2019. — Access mode:<http://pandas.pydata.org/pandas-docs/stable/> (online; accessed: 20.02.2019).
- [7] Chrétien M. Convert datetime.time to seconds [Electronic resource] // Stack Overflow. 2017. — Access mode: <https://stackoverflow.com/a/44823381> (online; accessed:20.02.2019).