МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра №806 «Вычислительная математика и программирование»

Курсовой работа
по курсу «Методы, средства и технологии мультимедиа»

Выполнил: А.А. Почечура
Группа: М8О-406Б-20
Преподаватели: Б.В. Вишняков

Москва, 2024

Условие

Цель работы. Научиться решать задачу классификации изображений с помощью применения технологии Transfer Learning, а также с помощью другой нейросетевой архитектуры и узнать, какой подход продемонстрирует лучшие результаты и почему.

Задание. Требуется изучить туториал, посвящённый обучению свёрточной сети (ConvNet/CNN) для классификации изображений с помощью трансферного обучения. Затем необходимо реализовать данный алгоритм, используя в качестве тренировочной и валидационной выборки самостоятельно выбранный датасет. Далее требуется оценить безлайн и на основе полученных выводов улучшить его. Затем нужно написать собственную нейросетевую архитектуру, которая также смогла бы позволить решать задачу классификации изображений.

Набор данных. Мною был выбран датасет с дорожными знаками. Каждый класс в нём обозначает определённый знак ограничения скорости. Датасет довольно актуальный в наше время, так как создан для решения реальных практических задач: обучение искусственного интеллекта соблюдать правила движения при управлении автомобилями.

Программное и аппаратное обеспечение

Графический процессор:

Compute capability: 7.5
Name: Tesla T4
Total Global Memory: 15835398144
Shared memory per block: 49152
Registers per block: 65536
Warp size: 32
Max threads per block: (1024, 1024, 64)
Max block: (2147483647, 65535, 65535)
Total constant memory: 65536
Multiprocessors count: 40

Процессор:

vendor_id : GenuineIntel
cpu family: 6
model: 85
model name: Intel(R) Xeon(R) CPU @ 2.00GHz
stepping: 3
microcode: 0xffffffff
cpu MHz: 2000.184

cache size : 39424 KB
physical id: 0
siblings: 2
core id: 0
cpu cores : 1
apicid: 0
initial apicid: 0
fpu: yes
fpu_exception: yes
cpuid level: 13
wp: yes
bogomips : 4000.36
clflush size: 64
cache_alignment : 64
address sizes: 46 bits physical, 48 bits virtual

Оперативная память и жёсткий диск:

Memory:
description: System memory
physical id: 0
size: 13GiB

Mem**:** 12982.6 MiB
Storage**:** 55 GiB

Программное обеспечение:

Google Colab
Операционная система: Ubuntu 22.04.2 LTS
Оболочка: python3
inxi: 3.3.13

Метод решения

Для работы с безлайном большинство кода будет задействовано с сайта. Если говорить о содержании туториала, то в нём очень хорошо и понятно объясняется, как обучить свою свёрточную сеть (ConvNet/CNN) для классификации изображений с помощью трансферного обучения.

Глубокие нейронные сети требовательны к большим объемам данных для сходимости обучения. И зачастую в нашей частной задаче недостаточно данных для того, чтобы хорошо натренировать все слои нейросети. Transfer Learning решает эту проблему.

Рассмотрим кратко главы, которые представлены в туториале:

1. Load Data: загрузка данных с помощью пакетов torchvision и torch.utils.data. Выбранный в примере датасет включал в себя изображения муравьёв и пчёл. Обученная модель должна была по картинке опредлять, какое насекомое представлено на изображении. Также данная часть статьи включает в себя функции демонстрации изображений.

2. Training the model: здесь происходит демонтрация функции, с помощью которой далее модель будет обучаться. Затем определяется функция для демонстрации результатов лучшей модели

3. Finetuning the ConvNet: на данном этапе объявляются все необхожимые инструменты для обучения и их параметры: оптимизатор, шедулер, функция ошибки и другое. После обучения модели сохраняются показатели, давшие лучший результат.

4. ConvNet as fixed feature extractor: происходит заморозка всех слоёв сети кроме последнего. После этого происходит снова обучение модели с помощью немного изменнённого алгоритма.

5. Inference on custom images: демонстрация работы с обученной моделью на произвольно взятой картинке из стороннего источника.

Для реализации безлайна, сначала подготовим и загрузим наши данные. Далее создадим отдельную функцию для обучения модели. В ней будет происходить поиск лучшего набора параметров. Затем зададим параметры для обучения: шедулер, оптимизатор, критерий и так далее. После обучение модели попробуем улучшить результаты. Для этого воспользуемся аугментацией: "заморозим" все слои сети, кроме последнего и узнаем, к чему приведёт данное решение. После очередного обучения формируем выводы.

Затем реализуем самостоятельно нейросетевую архитектуру, которая также позволит решать задачу классификации объектов. Выберем для этих целей архитектуру **ResNet18** от Microsoft Labs, которая в своё время выиграла конкурс ImageNet.

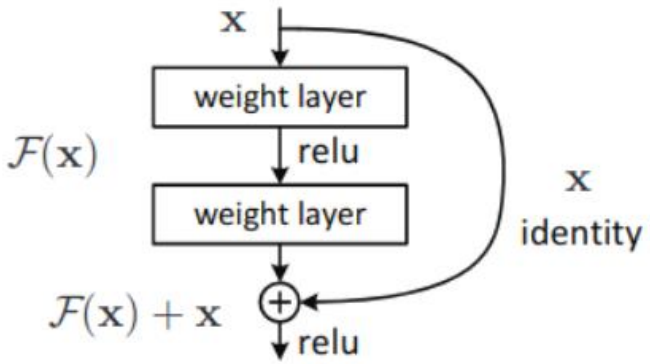
Специалисты отмечают, что глубокие стопки свёрточных слоёв приводят к сильному затуханию градиента, а как следствие, к плохой обучаемости. Чтобы решить эту проблему, они ввели "остаточные" (residual) пути по которым градиенту при

обратном распространении легче проходить. В результате, даже сети с 1000 слоями достаточно успешно учатся, приводя к дополнительному улучшению точности модели.

ResNet-18 - это сверточная нейронная сеть глубиной 18 слоев. Можно загрузить предварительно подготовленную версию сети, обученную на более чем миллионе изображений из базы данных ImageNet. Предварительно обученная сеть может классифицировать изображения по 1000 категориям объектов, таким как клавиатура, мышь, карандаш и множество животных. В результате сеть изучила богатые представления объектов для широкого спектра изображений. Размер входного изображения в сети составляет 224 на 224.

Если углубляться в суть, то **ResNet** применяет сопоставление идентификаторов между уровнями для достижения архитектуры сокращенных соединений. А уровни/блоки в архитектуре, состоящие из этих сокращенных соединений, известны как блоки остаточного обучения.

Рисунок ниже более конкретно описывает суть данной архитектуры:



После реализации и обучения данной нейросетевой архитектуры формируем выводы.

Описание программы

Вначале требуется подключить все необходимые библиотеки для работы с данными. Затем с помощью функций *data_transforms, image_datasets* и *dataloaders* происходит извлечение и нормализация данных для обучения. Для иллюстрации изображений из нашего датасета реализуем функцию *imshow*, которая выводит картинки и их принадлежность к конкретному классу.

Для реализации безлайна будет создана функция *train_model,* с помощью которой будет происходить обучение модели. Внутри неё будет происходить:

- Создание временного каталога для хранения контрольных точек при обучении (*with TemporaryDirectory() as tempdir:*);
- Обучение и оценка на каждой эпохе (*for phase in ['train', 'val']:*);
- Итерирование по данным (*for inputs, labels in dataloaders[phase]:*);

- Сохранение лучших весов модели (*model.load_state_dict(torch.load(best_model_params_path))*).

Далее для визуализации работы лучшей модели напишем функцию *visualize_model*, которая определяет принадлежность к классам для 6-ти случайных картинок.

Перед обучением зададим значение критерия (*criterion = nn.CrossEntropyLoss()*), оптимизатора (*optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.002, momentum=0.9)*) и шедулера (*exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)*), после чего обучим нашу модель.

Далее, для улучшения безлайна, воспользуемся аугментацией. В этот раз установим *require_grad = False*, чтобы заморозить параметры. В таком случае градиенты не будут вычислятся в *Back()*.

Теперь реализуем архитектуру ResNet-18. Данная архитектура строится из блоков, поэтому их можно описать в отдельном классе *BasicBlock*. Реализация самой архитектуры будет происходить в классе *ResNet18*. Затем снова задаём параметры модели: в качестве оптимизатора выступает *SGD* (он лучше всего подходит для обучения *ResNet-18*), шедулером будет *CosineAnnealingLR* (изменяет шаг обучения по косинусу). После обучения демонстрируем работу полученной модели с помощью *visualize_model*.

Результаты

Безлайн:

Training complete in 2m 60s
Best val Acc: 0.894737

Демонстрация работы:

Оценка качества: как видим, модель дала весьма неплохие результаты - почти 90% правильно предсказанных результатов на валидационной выборке.

Улучшение безлайна:

Training complete in 3m 44s
Best val Acc: 1.000000

Сравнение с результатами из пункта 4 и формирование выводов: как видим, новый метод обучения дал безупречные результаты: все картинки были правильно распределены по классам на валидационно выборке. Причина того, что "заморозка" слоёв сработала успешно, скорее всего кроется в выбранном мною датасете. Признаки, по котором оцениваются изображения, здесь не очень сложные, поэтому дообучивание предыдущих слоёв приводит к тому, что такие признаки обнаруживаются значительно лучше.

Нейросетевая архитектура (ResNet-18):

Training complete in 3m 39s
Best val Acc: 1.000000

Демонстрация работы:

Сравнение результатов: судя по результатам, лучшая модель, полученная с помощью архитектуры *ResNet-18*, справляется с задачей также, как и свёрточные сети из предыдущего пункта.

Исходный код

Подключаем все необходимые библиотеки для работы с данными

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory

cudnn.benchmark = True
plt.ion()
```

Извлечение и нормализация данных для обучения

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    ]),
}

data_dir = '/content/drive/MyDrive/archive'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
batch_size=4,
                                              shuffle=True,
num_workers=4)
              for x in ['train', 'val']}
```

```
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

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

Функция для иллюстрации изображений из нашего датасета

```
def imshow(inp, title=None):
    """Display image for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)   # небольшая пауза для обновления графиков

# Получаем пакет обучающих данных
inputs, classes = next(iter(dataloaders['train']))

# Создание сетки из данных
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])
```

Функция для обучения модели

```
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    since = time.time()

    # Создание временного каталога для хранения контрольных точек при
обучении
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir,
'best_model_params.pt')

        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0

        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Обучение и оценка на каждой эпохе
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Перевод модели в режим обучения
                else:
```

```
                    model.eval()   # Перевод модели в режим оценки

                running_loss = 0.0
                running_corrects = 0

                # Итерирование по данным
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    # Обнуление параметров градиента
                    optimizer.zero_grad()

                    # Шаг вперёд градиента
                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs)
                        _, preds = torch.max(outputs, 1)
                        loss = criterion(outputs, labels)

                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    # Статистика
                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds == labels.data)
                if phase == 'train':
                    scheduler.step()

                epoch_loss = running_loss / dataset_sizes[phase]
                epoch_acc = running_corrects.double() /
dataset_sizes[phase]

                print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}')

                if phase == 'val' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    torch.save(model.state_dict(),
best_model_params_path)

            print()

        time_elapsed = time.time() - since
        print(f'Training complete in {time_elapsed // 60:.0f}m
{time_elapsed % 60:.0f}s')
        print(f'Best val Acc: {best_acc:4f}')
```

```
        # Сохранение лучших весов модели
        model.load_state_dict(torch.load(best_model_params_path))
    return model
```

Функция для визуализации работы лучшей модели

```
def visualize_model(model, num_images=6):
    was_training = model.training
    model.eval()
    images_so_far = 0
    fig = plt.figure()

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['val']):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            for j in range(inputs.size()[0]):
                images_so_far += 1
                ax = plt.subplot(num_images//2, 2, images_so_far)
                ax.axis('off')
                ax.set_title(f'predicted: {class_names[preds[j]]}')
                imshow(inputs.cpu().data[j])

                if images_so_far == num_images:
                    model.train(mode=was_training)
                    return
        model.train(mode=was_training)
```

Необходимые параметры для обучения модели

```
model_ft = models.resnet18(weights='IMAGENET1K_V1')
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 6)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.002, momentum=0.9)

exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,
gamma=0.1)
```

Обучение модели

```
model_ft = train_model(model_ft, criterion, optimizer_ft,
exp_lr_scheduler,
```

```
                        num_epochs=25)
```

Демонстрация работы лучшей модели

```
visualize_model(model_ft)

plt.ioff()
plt.show()
```

Применение аугментации

```
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')
for param in model_conv.parameters():
    param.requires_grad = False

num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 6)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001,
momentum=0.9)

exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7,
gamma=0.1)
```

Обучение новой модели

```
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=25)
```

Класс *BasicBlock* для архитектуры ResNet-18

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=3,
            stride=(1 + (in_channels != out_channels)),
            padding=1,
            bias=False,
        )
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(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 = None
        if in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(
                    in_channels, out_channels, kernel_size=1, stride=2,
bias=False
                ),
                nn.BatchNorm2d(out_channels),
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        if self.downsample is not None:
            x = self.downsample(x)
        return self.relu(x + out)
```

Класс самой сети

```
class ResNet18(nn.Module):
    def __init__(self, block=BasicBlock, num_classes=1000):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2,
padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = nn.Sequential(block(64, 64), block(64, 64))
        self.layer2 = nn.Sequential(block(64, 128), block(128, 128))
        self.layer3 = nn.Sequential(block(128, 256), block(256, 256))
        self.layer4 = nn.Sequential(block(256, 512), block(512, 512))

        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)

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

        x = self.avgpool(x)
        return self.fc(x.flatten(1))
```

Параметры модели

```
my_model_ft = ResNet18(num_classes=len(class_names))

my_model_ft = my_model_ft.to(device)

my_optimizer_ft = optim.SGD(
    my_model_ft.parameters(),
    lr=0.001,
    weight_decay= 0.0001,
    momentum=0.9,
)

my_cos_lr_scheduler = lr_scheduler.CosineAnnealingLR(my_optimizer_ft,
T_max=25)
```

Обучение модели

```
my_model_ft = train_model(
    my_model_ft, criterion, my_optimizer_ft, my_cos_lr_scheduler,
num_epochs=25
)
```

Работа лучшей модели

```
visualize_model(model_ft)

plt.ioff()
plt.show()
```

Выводы

Работа с Transfer Learning в компьютерном зрении показалась довольно интересной и удобной в данной лабораторной работе. Простота пользования позволяет применять данный алгоритм любому простому человеку, который не является специалистом в данной сфере, для своих личных целей. При этом, качество работы модели, полученной в процессе обучения, впечатляет. Что же касается архитектуры ResNer-18, то с поставленной задачей она справляется также хорошо, как и сверточные сети из прошлого метода, но требуют более деликатного и серьёзного подхода к реализации. Также не исключаю того факта, что подобранный мною датасет очень хорошо подходит для обеих нейросетевый архитектур, из-за чего результаты оказались безупречными.