Лабораторная работа 1

Попов Илья, 406

Задание

1. Подробно изучить туториал и написать комментарий к каждой строчке кода
2. Выбрать наборы данных и обосновать его выбор (реальная практическая задача)
3. Выбрать метрики качества и обосновать их выбор
4. Создание бейзлайна и оценка качества
5. Улучшение бейзлайна

- Сформулировать гипотезы (подбор аугментаций и других гиперпараметров, подбор архитектуры нейронной сети)
- Проверить гипотезы
- Сравнить результаты моделей в сравнении с результатами из пункта 4
- Сделать выводы

1. Реализация нейросетевой архитектуры

- Самостоятельно реализовать выбранную нейронную сеть
- Сравнить результаты моделей в сравнении с результатами из пункта 5
- Сделать выводы

1. Изучение туториала

После изучения материала стало понятно, что основная цель трансферного обучения заключается в передаче знаний, полученных на одной задаче, на другую. Это достигается путем замораживания первых N слоев нейронной сети, ответственных за извлечение базовых признаков, и обучения новых слоев для решения конкретной задачи, например, классификации.

Разберем, что значат основные шаги в этом плейбуке для колаба:

1. Аугментация данных. Здесь мы применяем несколько методов для увеличения разнообразия обучающего набора: случайное обрезание изображения до размера 224x224 пикселей и случайное отражение по горизонтали. Это помогает модели лучше обучиться на различных вариантах изображений.

2. Нормализация данных. Мы меняем размер изображения до 256x256 пикселей, а затем вырезаем центральную область размером 224x224 пикселей. Это стандартная практика для подготовки данных перед подачей их на вход нейронной сети.

3. Загрузка предварительно обученной модели ConvNet. Мы используем модель ResNet-18, обученную на наборе данных IMAGENET1K_V1. Это позволяет нам использовать заранее обученные

веса, захватывающие общие признаки изображений, которые могут быть адаптированы к нашей конкретной задаче.

4. Отключение обучения для некоторых слоев. Это достигается установкой параметра requires_grad в False для этих слоев. Этот шаг полезен, когда мы хотим использовать предобученные веса и избежать их изменения.

5. Применение регуляризации весов с помощью lr_scheduler.StepLR. Этот метод позволяет регулировать скорость обучения в зависимости от эпох обучения. Уменьшение скорости обучения через каждые 7 эпох с коэффициентом 0.1 помогает предотвратить переобучение модели.

2. Выбор набора данных

Выбрана задача классификации изображений муравьев и пчел с использованием датасета с Kaggle. Она подходит для задачи классификации изображений с использованием transfer learning.

3. Выбор метрики качества

При выборе Accuracy я сравнивал ее с другими метриками, такими как Precision, Recall и F1-score. Хотя каждая из этих метрик имеет свои преимущества, мне показалось, что Accuracy лучше всего подходит для моей конкретной задачи классификации изображений муравьев и пчел. В отличие от Precision, которая фокусируется на точности предсказаний положительного класса, и Recall, которая измеряет способность модели обнаруживать все положительные примеры, Accuracy предоставляет более общую оценку производительности модели, что было важно для меня в данной ситуации. Кроме того, в отличие от F1-score, который учитывает как Precision, так и Recall, Accuracy была более простой и интуитивно понятной для интерпретации, что соответствовало моим целям в выборе метрики.

4. Создание бейзлайна

Выбор использовать ResNet-18 с весами IMAGENET1K_V1 в качестве основной модели был сделан после тщательного анализа доступных архитектур и ресурсов. ResNet-18 представляет собой компактную и эффективную архитектуру, позволяющую достичь хороших результатов в задачах классификации изображений, что делает ее привлекательным выбором для моего проекта.

Использование предобученных весов IMAGENET1K_V1 обусловлено желанием скорее запустить процесс обучения и сэкономить время, чем начинать с нуля. Предобученные веса содержат информацию о широком спектре изображений, что может помочь модели лучше обобщать и адаптироваться к новым данным.

Кроме того, ResNet-18 с весами IMAGENET1K_V1 предоставляет отличную отправную точку для дальнейших экспериментов и улучшений. После оценки базовой модели я планирую исследовать различные методы дообучения и адаптации модели к моей конкретной задаче, такие как fine-tuning и аугментация данных.

Download dataset

```
! mkdir dataset && wget -P dataset/
https://download.pytorch.org/tutorial/hymenoptera_data.zip
! unzip -q dataset/hymenoptera_data.zip -d dataset/
```

```
--2024-05-11 17:35:39--
https://download.pytorch.org/tutorial/hymenoptera_data.zip
Resolving download.pytorch.org (download.pytorch.org)...
18.239.225.41, 18.239.225.75, 18.239.225.61, ...
Connecting to download.pytorch.org (download.pytorch.org)|
18.239.225.41|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 47286322 (45M) [application/zip]
Saving to: ‘dataset/hymenoptera_data.zip’

hymenoptera_data.zi 100%[===================>]  45.10M  64.3MB/s    in
0.7s

2024-05-11 17:35:40 (64.3 MB/s) - ‘dataset/hymenoptera_data.zip’ saved
[47286322/47286322]
```

Useful imports

```
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 tqdm.auto import tqdm, trange
from PIL import Image
from tempfile import TemporaryDirectory

cudnn.benchmark = True
plt.ion()   # interactive mode
```

```
<contextlib.ExitStack at 0x7ffba6f9e170>
```

Process dataset

Тут происходят преобразования изображений, такие как обрезка до определенного размера и преобразование в формат тензора, затем создаются объекты для загрузки данных, учитывая пакетную обработку, перемешивание и количество рабочих процессов

для ускорения. Размеры наборов данных и список классов изображений также определяются здесь.

```
data_transforms = {
    'train': transforms.Compose([
        transforms.CenterCrop(224),
        transforms.ToTensor(),
    ]),
    'val': transforms.Compose([
        transforms.CenterCrop(224),
        transforms.ToTensor(),
    ]),
}

data_dir = 'dataset/hymenoptera_data'
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")
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/
dataloader.py:558: UserWarning: This DataLoader will create 4 worker
processes in total. Our suggested max number of worker in current
system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to
avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

Prepare network

Здесь происходит инициализация модели ResNet-18 с предварительными весами изображений ImageNet. Последний полносвязанный слой модели адаптируется под количество классов в наборе данных. Модель перемещается на выбранное устройство для обучения. Для расчета потерь используется кросс-энтропия, а для обновления весов модели используется оптимизатор Stochastic Gradient Descent (SGD). Устанавливается уменьшение скорости обучения с помощью StepLR для динамической адаптации скорости обучения.

```
model_ft = models.resnet18(weights='IMAGENET1K_V1')
num_ftrs = model_ft.fc.in_features
```

```
model_ft.fc = nn.Linear(num_ftrs, 2)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001,
momentum=0.9)

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

```
Downloading: "https://download.pytorch.org/models/resnet18-
f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-
f37072fd.pth
100%|██████████| 44.7M/44.7M [00:00<00:00, 105MB/s]
```

Fit model

Внутри функции идет цикл по эпохам, в каждой эпохе обучение и валидация модели.

```
import time
from tempfile import TemporaryDirectory
import os
import torch
from tqdm import tqdm

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

    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

        pbar = tqdm(total=num_epochs, desc='Training Progress',
position=0)

        for epoch in range(num_epochs):
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()
                else:
                    model.eval()

                running_loss = 0.0
                running_corrects = 0
```

```
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    optimizer.zero_grad()

                    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()

                    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]

                pbar.set_description(f'Epoch {epoch}/{num_epochs - 1}
{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)

            pbar.update(1)

        pbar.close()

        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}')

        model.load_state_dict(torch.load(best_model_params_path))
    return model

model_ft= train_model(model_ft, criterion, optimizer_ft,
exp_lr_scheduler, num_epochs=25)
```

```
Epoch 24/24 val Loss: 0.1863 Acc: 0.9216: 100%|| 25/25 [00:25<00:00,
1.01s/it]
```

```
Training complete in 0m 25s
Best val Acc: 0.934641
```

Вывод

По результатам двух проходов обучения с постоянным числом эпох (25) и теми же параметрами, модель достигла достаточно высокой точности на валидационном наборе данных. Обученная сеть показала отличные результаты - 93% на выборке валидации.

5. Улучшение бейзлайна

Для улучшения модели планируется использовать аугментацию данных и заморозить все слои, кроме последнего. Это достигается путем установки параметра requires_grad в False для всех параметров модели, что предотвратит обновление их значений во время обратного распространения ошибки (backward()). Такой подход поможет сосредоточиться на обучении только последнего слоя, что может привести к улучшению обобщающей способности модели и предотвращению переобучения.

Process dataset

```
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 = 'dataset/hymenoptera_data'
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")

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, 2)

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)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-
f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-
f37072fd.pth
100%|██████████| 44.7M/44.7M [00:01<00:00, 44.4MB/s]
```

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

```
 Training Progress:   0%|          | 0/25 [00:00<?,
?it/s]/usr/lib/python3.10/multiprocessing/popen_fork.py:66:
RuntimeWarning: os.fork() was called. os.fork() is incompatible with
multithreaded code, and JAX is multithreaded, so this will likely lead
to a deadlock.
  self.pid = os.fork()
Epoch 24/24 val Loss: 0.1855 Acc: 0.9412: 100%|██████████| 25/25
[18:32<00:00, 44.50s/it]

Training complete in 18m 33s
Best val Acc: 0.954248
```

Вывод

После внесения улучшений в модель, включая аугментацию данных и замораживание параметров слоев, мы достигли заметного увеличения точности на валидационном наборе данных с 93.4% до 95.42%. Это демонстрирует эффективность выбранных улучшений и подтверждает их значительное влияние на результаты модели.

6. Реализация архитектуры

В качестве архитектуры выбран ResNet с использованием блоков Residual Block. Каждый Residual Block состоит из двух сверточных слоев с функцией активации ReLU и слоев Batch Normalization.

Класс ResNet определяет архитектуру сети, включая сверточные слои, слои пулинга, а также последовательность блоков Residual Block. Здесь используются четыре последовательных слоя блоков Residual Block с различным количеством фильтров.

Функция _make_layer создает последовательность блоков Residual Block для каждого слоя ResNet, принимая на вход количество фильтров и количество блоков в слое. Если изменяется размерность входных данных или количество фильтров, применяется слой с 1x1 сверткой для подстройки размерности.

В методе forward определяется последовательное выполнение сверточных слоев, пулинга, блоков Residual Block и классификационного слоя. Размерность выхода усредняется по пространственным размерам с помощью слоя AvgPool2d и подается на классификационный слой для получения предсказаний.

Почему выбор пал на ResNet

Выбор модели ResNet для решения задачи обычно обусловлен ее хорошей производительностью в различных компьютерных зрении задачах. Вот несколько причин, по которым ResNet может быть выбрана:

Глубина сети: ResNet имеет глубокую архитектуру, что позволяет ей изучать сложные иерархии признаков на изображениях. Это особенно полезно для задач, где необходимо извлечь высокоуровневые признаки, такие как семантическая сегментация или классификация изображений.

Skip connections: ResNet включает в себя "skip connections" или "residual connections", которые позволяют градиентам более эффективно распространяться по сети во время обратного распространения ошибки. Это помогает в борьбе с проблемой затухания градиентов и позволяет обучать глубокие сети более успешно.

Производительность и эффективность: ResNet обеспечивает хорошую производительность на различных наборах данных и демонстрирует небольшое количество параметров по сравнению с некоторыми другими архитектурами. Это делает его привлекательным выбором для решения различных задач компьютерного зрения.

Предварительно обученные модели: ResNet представлен в предварительно обученной форме на больших наборах данных, таких как ImageNet. Это позволяет использовать предварительно обученные веса для переноса обучения, что ускоряет процесс обучения и улучшает общую производительность модели.

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride = 1,
downsample = None):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Sequential(
```

```
                        nn.Conv2d(in_channels, out_channels,
kernel_size = 3, stride = stride, padding = 1),
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())
        self.conv2 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels,
kernel_size = 3, stride = 1, padding = 1),
                        nn.BatchNorm2d(out_channels))
        self.downsample = downsample
        self.relu = nn.ReLU()
        self.out_channels = out_channels

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.conv2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes = 10):
        super(ResNet, self).__init__()
        self.inplanes = 64
        self.conv1 = nn.Sequential(
                        nn.Conv2d(3, 64, kernel_size = 7, stride = 2,
padding = 3),
                        nn.BatchNorm2d(64),
                        nn.ReLU())
        self.maxpool = nn.MaxPool2d(kernel_size = 3, stride = 2,
padding = 1)
        self.layer0 = self._make_layer(block, 64, layers[0], stride =
1)
        self.layer1 = self._make_layer(block, 128, layers[1], stride =
2)
        self.layer2 = self._make_layer(block, 256, layers[2], stride =
2)
        self.layer3 = self._make_layer(block, 512, layers[3], stride =
2)
        self.avgpool = nn.AvgPool2d(7, stride=1)
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes:

            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes, kernel_size=1,
```

```
stride=stride),
                nn.BatchNorm2d(planes),
            )
        layers = []
        layers.append(block(self.inplanes, planes, stride,
downsample))
        self.inplanes = planes
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)


    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)

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

        return x
```

Лосс-функция определяется как CrossEntropyLoss(), которая часто используется для задач классификации.

Оптимизатор настроен как стохастический градиентный спуск (SGD) с параметрами скорости обучения (lr=0.001) и момента (momentum=0.9). Он используется для обновления параметров модели в процессе обучения.

Для управления скоростью обучения используется шедулер StepLR, который уменьшает скорость обучения на заданный коэффициент gamma=0.1 каждые step_size=7 эпох.

```
model = ResNet(ResidualBlock, [3, 4, 6, 3]).to(device)
criterion = nn.CrossEntropyLoss()

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

scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

my_model = train_model(model_conv, criterion, optimizer, scheduler,
num_epochs=100)
```

```
Epoch 99/99 val Loss: 0.7176 Acc: 0.6013: 100%|█| 100/100
[01:32<00:00,  1.09it/
```

```
Training complete in 1m 32s
Best val Acc: 0.601307
```

Вывод

После введения архитектуры ResNet и обучения модели в течение 100 эпох, мы получили точность на валидационном наборе данных около 60.13%. Это может свидетельствовать о том, что модель не смогла достаточно хорошо обобщить обучающие данные и показать хорошие результаты на новых данных.

Общий вывод

В небольших наборах данных обучать модель с нуля нецелесообразно из-за вероятности получить низкое качество результатов. Трансферное обучение представляет собой более эффективный подход, поскольку предварительно обученная модель уже обладает опытом работы с изображениями, и дообучение ее для конкретной задачи в таком случае является более оптимальным.