Homework 2

Task 1: DIY loss scaling

Задание: необходимо реализовать масштабирование лосса при обучении модели в AMP режиме. Необходимо достичь точности ≥ 0.985 за 5 эпох обучения.

Подготовительные действия

1. Скачивание данных

```
cd week03_fast_pipelines/homework/task1
./download_data.sh
```

2. Дополнение кода для запуска без ошибок

Необходимо добавить в файл train.py запуск обучения:

```
if __name__ == "__main__":
    train()
```

А также добавить код для оптимизатора:

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

3. Получение бейзлайна

После этого можно ставить обучение:

```
python3 train.py
```

Полученная точность: 93.49%

Loss scaling

Я реализовал сразу класс для скейлинга лосса в соответсвии с инструкцией от nvidia. Также добавил возможность задавать стратегию скейлинга через командную строку.

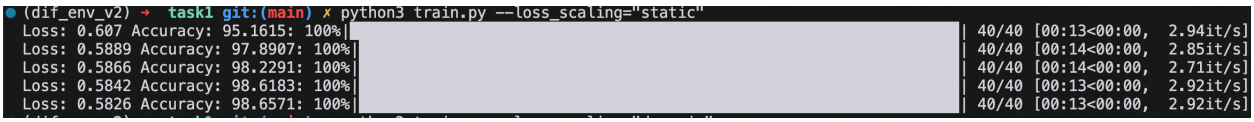
Получился следующий код (новый код выделен зеленым цветом):

```
import torch
from torch import nn
from tqdm.auto import tqdm
from argparse import ArgumentParser
from unet import Unet

from dataset import get_train_data


class GradScaler:
    def __init__(self, loss_scaling, init_scale=2**16, growth_factor=2.0,
    backoff_factor=0.5, growth_interval=100):
        self.loss_scaling = loss_scaling
        self.scale = init_scale
        self.growth_factor = growth_factor
        self.backoff_factor = backoff_factor
        self.growth_interval = growth_interval
        self.steps_since_inc = 0

    def scale_loss(self, loss):
        return loss * self.scale

    def unscale_grads(self, model):
        for param in model.parameters():
            if param.grad is not None:
                param.grad.data.div_(self.scale)

    def check_overflow(self, model):
        for param in model.parameters():
            if param.grad is not None:
                if torch.isnan(param.grad).any() or torch.isinf(param.grad).any():
                    return True
        return False

    def update(self, model):
        if self.loss_scaling == "static":
            return True
        if self.check_overflow(model):
            self.scale *= self.backoff_factor
            self.steps_since_inc = 0
            return False
        else:
            self.steps_since_inc += 1
            if self.steps_since_inc >= self.growth_interval:
                self.scale *= self.growth_factor
                self.steps_since_inc = 0
```

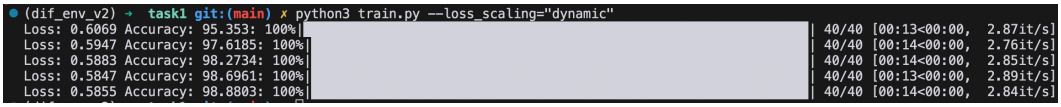
```
            return True


def train_epoch(
    train_loader: torch.utils.data.DataLoader,
    model: torch.nn.Module,
    criterion: torch.nn.modules.loss._Loss,
    optimizer: torch.optim.Optimizer,
    device: torch.device,
    scaler: GradScaler,
) -> None:
    model.train()

    pbar = tqdm(enumerate(train_loader), total=len(train_loader))
    for i, (images, labels) in pbar:
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        with torch.amp.autocast(device.type, dtype=torch.float16):
            outputs = model(images)
            loss = criterion(outputs, labels)

            scaled_loss = scaler.scale_loss(loss)
            scaled_loss.backward()
            scaler.unscale_grads(model)
            if not scaler.update(model):
                continue
            optimizer.step()

        accuracy = ((outputs > 0.5) == labels).float().mean()

        pbar.set_description(f"Loss: {round(loss.item(), 4)} " f"Accuracy: {round(accuracy.item() * 100, 4)}")


def train(loss_scaling: str):
    device = torch.device("cuda:0")
    model = Unet().to(device)
    criterion = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
    scaler = GradScaler(loss_scaling)

    train_loader = get_train_data()

    num_epochs = 5
    for epoch in range(0, num_epochs):
        train_epoch(train_loader, model, criterion, optimizer,
                            device=device, scaler=scaler)
```

```
if __name__ == "__main__":
    parser = ArgumentParser()
    parser.add_argument("--loss_scaling", type=str, default="static")
    args = parser.parse_args()
    train(args.loss_scaling)
```

Полученная точность с статичным скейлингом: 98.65%



Полученная точность с динамическим скейлингом: 98.88%



Task 2: efficient batching for language modeling

В данной секции описаны эксперименты по сравнению разных подоходов к паддингу текстовых данных.

Для начала, все эксперименты проводились на 100000 текстов для ускорения сравнения, это составляет 8.5% всего датасета, что достаточно для хорошей оценки времени.

Наиболее важный кусок кода здесь — это замер времени. Привожу его ниже:

```
# Start timing
start_event = torch.cuda.Event(enable_timing=True)
end_event = torch.cuda.Event(enable_timing=True)

start_event.record()
with torch.no_grad(), torch.autocast(device_type='cuda', dtype=torch.bfloat16):
    model(input_ids, src_mask)
end_event.record()

# Synchronize CUDA and record time
torch.cuda.synchronize()
batch_time = start_event.elapsed_time(end_event)
batch_times.append(batch_time)
```

Я добавил torch.autocast(device_type='cuda', dtype=torch.bfloat16) для ускорения, так как его довольно часто используют при обучении моделей.

batch_size = 256

Время замеряется в миллисекундах.

Для разогрева GPU я прогоняю модель на 10 батчах, состоящих из случайных индексов.

BRAIN

В данном эксперименте все последовательности добиваются паддингами до максимальной длины.

	MIN	MAX	MEAN	MEDIAN
BRAIN	65.33	149.06	104.12	104.01

BIG_BRAIN

В данном эксперименте все последовательности добиваются паддингами до максимальной длины в батче, при этом не больше MAX_LENGTH.

	MIN	MAX	MEAN	MEDIAN
BIG_BRAIN	37.15	178.99	75.03	72.13

ULTRA_BIG_BRAIN

В данном эксперименте необходимо реализовать сэмплер данных, при котором в батче разница между самым длинным и самым коротким текстами будет не больше k.

Я сделал довольно понятную и простую реализацию.

1. Сначала токенизировал данные.
2. После этого построил хэштаблицу, в которой каждой длине соответствовал список индексов текстов данной длины.
3. После чего просто я прошелся в цикле по отсортированному списку длин и соблюдая условие на k и объединил все тексты по группам. Причем таким образом, что количество текстов в группе кратно размеру батча. Если получалась такая группа, в которой элементов меньше, чем в батче и их нельзя объединить со следующими текстами, так как те сильно длиннее, то такие группы выбрасывались.
4. После внутри групп разбил случайно тексты на батчи.

	MIN	MAX	MEAN	MEDIAN
ULTRA_BIG_BRAIN	2.82	108.45	18.50	15.35

Зависимость от k

k	1	5	10	20	50	640
MEDIAN	12.24	13.76	15.35	16.98	17.32	67.85

Результаты соотносятся с логикой, чем больше k тем больше время. Понятно, что из-за этого должно проседать качество, так как сэмплы становятся все более скоррелированы.

ULTRA DUPER BIG BRAIN

В данном эксперименте необходимо реализовать стратегию сэмплирования данных, при котором полностью отсутствуют паддинги. Другими словами, абстрактно, необходимо вытянуть все тексты в одну строку и нарезать ее на батчи.

Для реализации этой идеи я сделал следующее:

1. Токенизировал все данные и перемешал
2. После этого в цикле шел по всем последовательностям и динамически наполнял последовательность до максимальной длины.
3. Одновременно с этим я запоминал сегменты отдельных текстов для каждой такой последовательности.
4. После я формировал блочно диагональную каузальную маску, чтобы разные тексты при вычислении attention не влияли друг на друга
5. Дополнительно пришлось изменить подачу маски в трансформер. Так как он принимал на вход маску n_head * batch_size, seq_len, seq_len.

Я отказался от идеи использования IterableDataset, так как такой подоход у меня работал дольше.

Понятно, что время подросло, так как теперь последовательности полностью наполнены.

	MIN	MAX	MEAN	MEDIAN
ULTRA_BIG_BRAIN	49.77	119.38	116.35	117.63

Task 3

В данной задаче требуется реализовать профайлер для измерения производительности модели на уровне слоев. Профайлер должен отслеживать время выполнения каждого слоя во время прямого и обратного проходов.

Реализация профайлера:

```
import json
import time
import torch
import os
from collections import defaultdict


class Profile:
    def __init__(self, model, name: str = "model", schedule: dict = None):
        self.name_map = self._build_name_map(model, name)
        self.events = []
        self.schedule = schedule or {"wait": 1, "warmup": 1, "active": 3}  # Default schedule
        self.active = False
        self.current_step = 0

    def _build_name_map(self, model, name="model"):
        # Build a mapping from module objects to their display names
        name_map = {}

        # Iterate through all modules in the model, getting their full hierarchical names
        for full_name, module in model.named_modules():
            # For the root module, use the provided name parameter
            if full_name == "":
```

```
            full_name = name

        # For leaf modules just use the class name
        if self._is_leaf(module):
            name_map[module] = module.__class__.__name__
        # For non-leaf modules, include the full hierarchical path and class name
        else:
            name_map[module] = f"{full_name}: {module.__class__.__name__}"

    return name_map

def _is_leaf(self, module):
    return len(list(module.children())) == 0

def __enter__(self):
    for module in self.name_map.keys():
        # Runs before the forward pass of each module
        module.register_forward_pre_hook(self._forward_pre_hook)
        # Runs after the forward pass of each module
        module.register_forward_hook(self._forward_post_hook)

        module.register_full_backward_pre_hook(self._backward_pre_hook)
        module.register_backward_hook(self._backward_post_hook)
    return self

def __exit__(self, type, value, traceback):
    pass

def _forward_pre_hook(self, module, inputs):
    if self.active:
        module._start_event = torch.cuda.Event(enable_timing=True)
        module._end_event = torch.cuda.Event(enable_timing=True)
        module._start_event.record()

def _forward_post_hook(self, module, inputs, outputs):
    if self.active:
        module._end_event.record()
        torch.cuda.synchronize()  # Ensure completion
        elapsed_time = module._start_event.elapsed_time(module._end_event)
        self.events.append({"name": self.name_map[module], "type": "forward", "time": elapsed_time})

def _backward_pre_hook(self, module, grad_output):
    if self.active:
        module._start_event = torch.cuda.Event(enable_timing=True)
        module._end_event = torch.cuda.Event(enable_timing=True)
        module._start_event.record()

def _backward_post_hook(self, module, grad_input, grad_output):
```

```
        if self.active:
            module._end_event.record()
            torch.cuda.synchronize()  # Ensure completion
            elapsed_time = module._start_event.elapsed_time(module._end_event)
            self.events.append({"name": self.name_map[module], "type": "backward", "time": elapsed_time})

    def step(self):
        self.current_step += 1
        if self.current_step > self.schedule["wait"] + self.schedule["warmup"]:
            self.active = True

    def summary(self):
        print("Summary:")
        for event in self.events:
            print(event)

    def to_perfetto(self, path="trace.json"):
        trace_events = []
        pid = os.getpid()  # Process ID
        tid = 0  # Single-threaded

        for event in self.events:
            trace_events.append({
                "name": event["name"],
                "cat": event["type"],
                "ph": "X",
                "ts": event["time"] * 1e3,  # Convert milliseconds to microseconds
                "dur": event["time"] * 1e3,
                "pid": pid,
                "tid": tid,
            })

        with open(path, "w") as f:
            json.dump({"traceEvents": trace_events}, f, indent=4)

        print(f"Perfetto trace saved to {path}")
```

Я реализовал только базовый профилер без ядер куды.

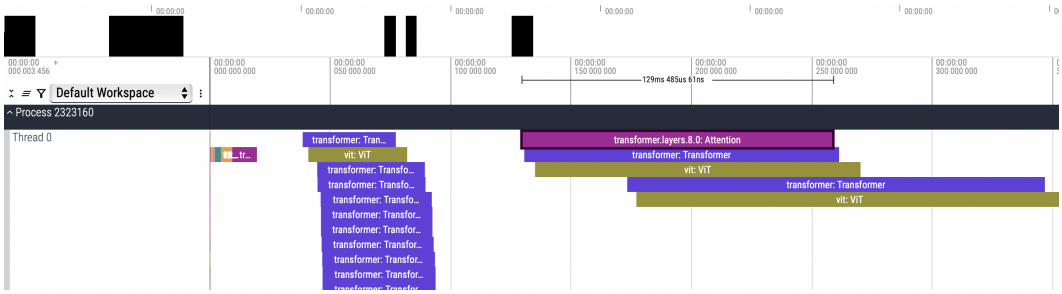
Список неэффективностей:

1. В Vit размерность *dim*=255 не является степенью двойки. Использование 256 будет более эффективно для параллелизации на куде.
2. В аугментации изображений излишне делать transforms.CenterCrop(224), так как после него опять идет кроп такого же размера, можно оставить только второй.

```
def get_train_transforms() -> tp.Any:
    return transforms.Compose(
        [
            transforms.Resize((320, 320)),
            transforms.CenterCrop(224),
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.AugMix(),
            transforms.ToTensor(),
        ]
    )
```

3. Еще одна очевидная ошибка — в dataloader не используются num_workers
4. Attention реализован неправильно

Во-первых, он работает очень долго.



Во-вторых, сама реализация даже не multi head

```
def forward(self, x):
    q = self.queries(x)
    k = self.keys(x)
    v = self.values(x)

    dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale

    attn = self.attend(dots)
    attn = self.dropout(attn)

    out = torch.matmul(attn, v)

    return self.to_out(out)
```

Я заменил его на:

```
def forward(self, x):
    x = self.norm(x)
    b, n, _ = x.shape
```

```
    # More efficient combined QKV projection
    qkv = self.qkv(x).chunk(3, dim=-1)
    q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h=self.heads), qkv)

    output = torch.nn.functional.scaled_dot_product_attention(
        q, k, v,
        dropout_p=self.dropout if self.training else 0.0,
        is_causal=False
    )
    output = rearrange(output, 'b h n d -> b n (h d)')
    return self.to_out(output)
```

5. Валидация происходит без torch.no_grad()
6. Неоптимальный способ считывания изображений

На семинаре разбирали, что быстрее всего читать с помощью cv2

```
img = cv2.imread(f"{self.folder_path}/{img_name}.jpg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = Image.fromarray(img)
```

7. Также я обнаружил баг на валидации

```
val_accuracy += acc.item() / len(train_loader)
val_loss += loss.item() / len(train_loader)
```