МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Инженерная школа информационных технологий и робототехники
Направление подготовки 09.04.01 Информатика и вычислительная техника
Отделение Информационных технологий

Отчет по курсовому проекту
по дисциплине «Параллельные и высокопроизводительные вычисления»

Тема работы
Реализация алгоритма классификации посуды с использованием архитектуры ResNet и применением параллельных вычислений на графическом процессоре

Студент

Группа	ФИО	Подпись	Дата
8ВМ22	Ямкин Н.Н.		

Руководитель

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент ОИТ	Аксёнов С.В.	к.т.н., доцент		

Томск – 2023

Оглавление

Цель работы: реализовать алгоритм классификации посуды по двум классам (чистая или грязная) с применением параллельных вычислений на GPU.

Задачи:

1. Разбить обучающую выборку на обучающую и валидационную. Каждому изображению присвоить метку класса, к которому оно относится.
2. Подготовить данные для обучения: преобразовать исходные изображения в тензор, сделать их аугментацию.
3. Разбить обучающую выборку на батчи.
4. Обучить модель ResNet на подготовленных данных и применить её к тестовой выборке.
5. Сравнить время обучения модели на CPU и на GPU.

Ход работы

Начнем с набора данных. Датасет взят Kaggle, это соревнование, где предлагается отклассифицировать тарелки – либо они грязные, либо они уже помытые. Соревнование интересно тем, что в нем очень мало обучающих данных. В датасете всего 40 примеров тарелок с разметкой (то есть, 20 будет примеров с грязной посудой, 20 примеров с чистой), а огромная часть тарелок находится в отложенной выборке, и их метки необходимо предсказать.

Хорошо: у нас есть данные, теперь напишем некоторый код, который будет предсказывать правильный класс.

Суть предлагаемого решения в следующем: берем некоторую архитектуру (в данном случае возьмем ResNet), дообучаем её на новых изображениях и классифицируем весь тестовый датасет. Таким образом, для решения исходной задачи классификации изображений будет применен метод Transfer Learning.

Получаем выборку данных для валидации

Для начала разделим train dataset на две части – на обучающую часть и на валидационную. На трейне будем обучать нейронную сеть, а на валидации будем смотреть, насколько она хорошо работает, какую точность показывает на тех данных, которые она не видела.

Для этого создадим две новые папки: train и val, и каждую шестую фотографию из train переложим в папку с валидацией. Код ниже выполняет ровно эту задачу: здесь берётся каждая шестая фотография и кладется в валидацию. И теперь фотографии лежат в корневой директории, в папках train и val.

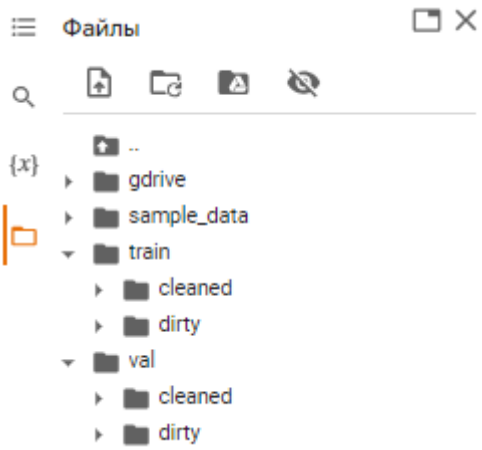

Рисунок 1 – Результат разбиения датасета

```
import numpy as np
import pandas as pd
from tqdm import tqdm
import time
import os
import shutil

from google.colab import drive
drive.mount('/content/gdrive')

data_root = '/content/gdrive/MyDrive/platesv2/plates/plates'
print(os.listdir(data_root))

train_dir = 'train'
val_dir = 'val'
class_names = ['cleaned','dirty']

for dir_name in [train_dir, val_dir]:
  for class_name in class_names:
    os.makedirs(os.path.join(dir_name, class_name), exist_ok = True) # Создаем папки для трейна и валидации в
корневой папке

for class_name in class_names:
  source_dir = os.path.join(data_root,'train',class_name)
  for i, file_name in enumerate(os.listdir(source_dir)):
    if i % 6 != 0:                                        # каждую шестую фотку копируем в валидационную выборку,
остальные копируем в трейн
      dest_dir = os.path.join(train_dir, class_name)     #  создаем директорию для трейна
    else:
      dest_dir = os.path.join(val_dir,class_name)        # создаем директорию для валидации
    shutil.copy(os.path.join(source_dir,file_name),os.path.join(dest_dir,file_name))   # копируем файлы из корневой
папки в новую в с помощью shutil.copy
```

Подготавливаем данные для обучения

Теперь нужно создать некоторые итераторы по этим директориям, чтобы они получали картинки, формировали из этих картинок тензоры (батчи с изображениями) и их можно было бы уже передавать в нейронную сеть. К счастью, в PyTorch уже есть функциональность ImageFolder, которая выполняет такую задачу.

ImageFolder предоставляет возможность итерироваться по изображением и получает некоторые пары (изображение и его метка). Откуда же он берёт метки? Метки копируются из названий тех папок, в которых лежат изображения.

Для того, чтобы перевести изображения в тензоры нужно сделать их некоторую трансформацию.

Для начала уменьшим изображение до размера 224 на 224. В принципе, ResNet инвариантен к размерам, но, конечно же, обучать его нужно на объектах одинакового размера. Заданный размер выбран не случайно – он совпадает с размером изображений из датасета ImageNet, на котором предобучен ResNet.

После этого нужно превратить модифицируемое изображение в тензор.

И есть ещё третья трансформация, которая уже не такая тривиальная – этот нормировка.

Данные трансформации нужны, чтобы привести все изображения к одному виду.

Кроме того, мы можем увеличивать train dataset за счёт аугментации. Аугментация – это процесс изменения или расширения данных путем добавления искусственно созданных или измененных элементов. В контексте решаемой задачи, создадим новые обучающие примеры из существующих данных путем применения масштабирования и зеркального отражения. Это позволит увеличить размер обучающего набора данных, улучшить общую обобщающую способность модели и уменьшить риск переобучения.

```
# Функция transforms.Compose() используется для создания последовательности преобразований. Переданные в эту функцию преобразования будут применяться последовательно к образцам обучающих данных.
train_transforms = transforms.Compose([
    transforms.RandomResizedCrop(224),        # Обрезаем изображение в случайном месте, а затем изменяем размер обрезки до заданного размера (224 x 224).
    transforms.RandomHorizontalFlip(p = 0.5),        # Выполняет зеркальный переворот изображения по горизонтали с заданной вероятностью.
    transforms.ToTensor(),                    # Преобразует изображение в тензор PyTorch.
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) #  Нормализует значения пикселей изображения.
                                                                       # Значения пикселей нормализуются с использованием заданных средних значений и стандартных отклонений для каждого цветового канала RGB.
   # Нормализация помогает в обучении модели, облегчая оптимизацию и уменьшая влияние различных масштабов значений пикселей на результаты.
])

val_transforms = transforms.Compose([
    transforms.Resize((224, 224)),            # просто изменяем размер изображения до заданных размеров (224x224 пикселей)
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#  С помощью класса torchvision.datasets.ImageFolder загружаем данные изображений, представленных в папке train_dir,
#  и применяем указанные преобразования train_transforms к каждому образцу данных во время загрузки
train_dataset = torchvision.datasets.ImageFolder(train_dir, train_transforms)

val_dataset = torchvision.datasets.ImageFolder(val_dir, val_transforms)
```

```
# Создадим итератор train_dataloader, который предоставляет доступ к обучающему набору данных в партиях (batch) для использования в процессе обучения модели.
# train_dataset - набор данных, который будет использоваться для создания итератора
# batch_size - размер партии данных (количество образцов в каждой партии)
# shuffle=True гарантирует, что данные будут перемешиваться перед каждой эпохой обучения. Это помогает предотвратить запоминания порядка обучения моделью  и сделать процесс обучения более стохастическим.
# num_workers=batch_size определяет количество фоновых рабочих процессов, которые будут использоваться для загрузки данных параллельно.
# Это может ускорить загрузку данных, позволяя параллельно выполнять операции загрузки и предварительной обработки данных.

batch_size = 8
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=batch_size)
val_dataloader = torch.utils.data.DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=batch_size)
```

Далее напишем функцию, которая бы выводила изображения из батча.

```
def show_input(input_tensor, title=''):
    image = input_tensor.permute(1, 2, 0).numpy()
    image = std * image + mean
    plt.imshow(image.clip(0, 1))
    plt.title(title)
    plt.show()
    plt.pause(0.001)

X_batch, y_batch = next(iter(train_dataloader))

for x_item, y_item in zip(X_batch, y_batch):
    show_input(x_item, title=class_names[y_item])
```

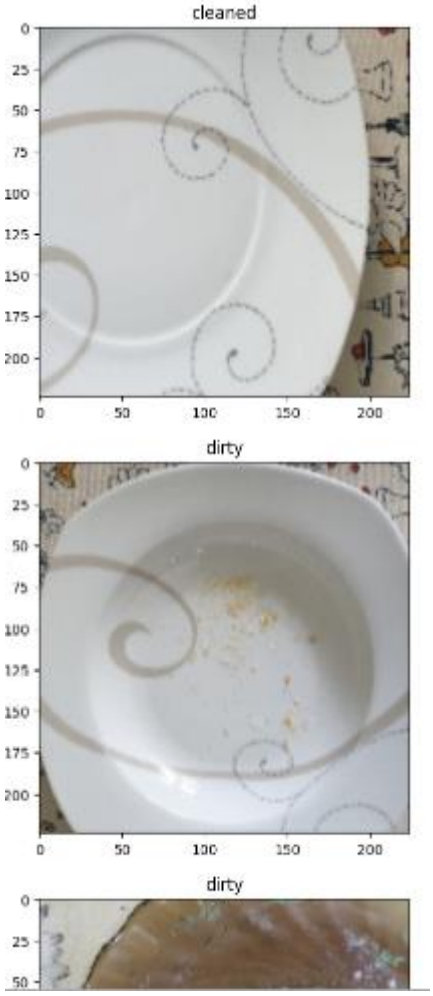

Рисунок 2 – Результат работы функции

Обучаем модель ResNet18

Теперь, когда все данные подготовлены, следует написать некоторую функцию train_model, которая будет на вход получать саму модель, лосс-функцию, оптимизатор, планировщик (scheduler), а также количество эпох обучения.

Функция train_model итерируется по эпохам, в каждой из которых она итерируется по батчам. Если активна фаза обучения, то функция считает backward, а optimizer делает шаги градиентного спуска, если же активна фаза валидации, то функция считает точность на валидационных данных.

Итак, каждую эпоху будем делать следующее: сперва попадаем в фазу «train», и тогда нам нужен train DataLoader, который формирует батчи из обучающего датасета, из папки «train». Затем планировщик делает шаг, который будет говорить о том, что произошла одна эпоха. Каждые 7 эпох scheduler должен уменьшать градиентный спуск. Таким образом, один степ, для него – это одна эпоха. В конце нужно перевести модель в режим «train», чтобы веса нейросети могли обновляться.

Далее итерируемся по DataLoader, который отдает батч с изображениями одним тензором (один батч – один тензор), и с лейблами. Затем переводим полученные данные на GPU. Кроме того, следует обнулить градиент, чтобы он не накапливался (в PyTorch, по умолчанию, он накапливается). После этого мы считаем предсказание, ошибку и определяем класс, с помощью функции argmax(). Если у нас фаза «train», то вычисляем градиент и обновляем веса нейросети.

```
def train_model(model, loss, optimizer, scheduler, num_epochs):
    for epoch in range(num_epochs):
        print('Epoch {}/{}:'.format(epoch, num_epochs - 1), flush=True)

        # Каждая эпоха имеет обучающий и валидационный (оценочный) режим работы
        for phase in ['train', 'val']:
            if phase == 'train':
                dataloader = train_dataloader  # обучающие данные (как он видит train_dataloader внутри функции и
не возникает ошибки???? )
                scheduler.step()      # обновляем скорость обучения в соответствии с заданным расписанием
                model.train()  # переключаем модель в режим обучения, в котором веса могут обновляться во время
обратного распространения ошибки
            else:
                dataloader = val_dataloader  # валидационные данные (как он видит train_dataloader внутри функции и
не возникает ошибки???? )
                model.eval()   # переключаем модель в режим оценки, в котором веса НЕ могут обновляться во время
обратного распространения ошибки. В этом режиме модкль используется только для оценки, а не для обучения

            running_loss = 0
            running_acc = 0

            # Итерируемся по данным
            for inputs, labels in tqdm(dataloader):
                inputs = inputs.to(device)        # перемещаем изображения на GPU
                labels = labels.to(device)        # перемещаем метки на GPU

                optimizer.zero_grad()   # Обгуляем градиенты оптимизатора

                # forward и backward
                with torch.set_grad_enabled(phase == 'train'):
                    preds = model(inputs)             # Прямой проход. Модель получает данные и выполняет
предсказания по ним
```

```
                    loss_value = loss(preds, labels)  # Вычисляем ошибку между предсказаниями и метками
                    preds_class = preds.argmax(dim=1) # Вычисляем классы, выбирая индекс максимального предсказания

                    #  Если текущая фаза равна 'train', выполняется обратный проход (backward pass) и обновление
весов модели
                    if phase == 'train':
                        loss_value.backward() # Вычисляем градиенты потери по весам модели
                        optimizer.step() # Обновляем веса модели на основе вычисленных градиентов

                # statistics
                running_loss += loss_value.item()
                running_acc += (preds_class == labels.data).float().mean()

            epoch_loss = running_loss / len(dataloader)
            epoch_acc = running_acc / len(dataloader)

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss, epoch_acc), flush=True)

    return model
```

Теперь зададим параметры функции train_model: модель, loss-функция, метод градиентного спуска и планировщик того, как будет уменьшаться шаг градиентного спуска.

Итак, сперва нам нужна модель. В данной работе будем использовать ResNet. В PyTorch есть много разных ResNet, например есть ResNet-18, самая маленькая модель из тех, которые доступны в библиотеке. Число «18» говорит о том, что в архитектуре модели 18 слоёв. Выбранная модель нужна в формате pre-trained, то есть нам нужны веса, которые получились вследствие обучения этого ResNet, на датасете ImageNet.

Так как используем предобученную модель, то необходимо обучить один полносвязанный слой, который находится на последнем слое. В этом случае следует «заморозить» все остальные веса нейросети, чтобы они не использовались в обучении. После этого поменяем последний полносвязанный слой, потому что по умолчанию он классифицирует тысячу классов, а у нас класса всего два.

Далее переложим объявленную ранее модель на GPU.

Следующий шаг – определяем лосс-функцию. В данном случае это бинарная кросс-энтропия, так как имеем всего два класса.

Теперь нужно определить метод градиентного спуска. В данном случае будем использовать Adam – самый лучший алгоритм, который хорошо работает «из коробки» и в нем практически ничего не нужно изменять.

Также объявим планировщик, который позволяет изменять скорость обучения.

```
model = models.resnet18(pretrained=True)
# "Замораживаем" уже обученые на датасете ImageNet веса НС, так, чтобы они не изменялись при обучении для нашей задачи
for param in model.parameters():
    param.requires_grad = False

# Заменим последний полносвязный слой в ResNet18, который по умолчанию классифицирует 1000 классов, на свой, который классифицирет изображение по 2 классам (грязная/чистая посуда)
# Будем обучать только последний слой ResNet18
# Создадим слой torch.nn.Linear, это полносвязный слой, на вход он принимает model fc_in features.
# Давайте посмотрим, сколько там будет фичей (это вот, как раз, признаковое описание объекта). Ну вот, 512 фичей. То есть 512, нейронов на вход приходит, и на выход нам нужно два нейрона, то есть два класса.
model.fc = torch.nn.Linear(model.fc.in_features, 2)

# Таким образом, сначала мы запретили модели обучаться, отключив все градиенты, потом создали последний слой заново, а у него веса по-умолчанию не заморожены, то есть обучаемы.
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu") # проверяем, доступно ли устройство с поддержкой CUDA (GPU).  Если устройство CUDA доступно, то переменная device устанавливается на "cuda:0", указывая использовать GPU с индексом 0
model = model.to(device) # модель model перемещается на указанное устройство

loss = torch.nn.CrossEntropyLoss()  # Функция потерь (Кросс-энтропия)
optimizer = torch.optim.Adam(model.parameters(), lr=1.0e-3) # Обучающий алгоритм. Оптимизатор Адам

# scheduler реализует так называемый "шаговый" (step) планировщик, который позволяет изменять скорость обучения с определенным шагом во время обучения
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

#Параметры конструктора torch.optim.lr_scheduler.StepLR:
#- optimizer: Это объект оптимизатора, для которого будет применяться планировщик.
#- step_size: Через сколько эпох нужно снижать скорость обучения.
#- gamma: Множитель, на который будет умножаться скорость обучения при снижении.
```

На данный момент создана нейросеть и определено всё, что требуется. Запустим функцию train_model на 100 эпох и зафиксируем время обучения.

```
start_time = time.time()
train_model(model, loss, optimizer, scheduler, num_epochs=100)
elapsed_time = time.time() - start_time
print('Время обучения:', elapsed_time)
```

Подготовим тестовую выборку и получим предсказания на ней.

```
test_dataset = ImageFolderWithPaths('/content/test', val_transforms)

test_dataloader = torch.utils.data.DataLoader(
    test_dataset, batch_size=batch_size, shuffle=False, num_workers=0)

model.eval()  # переводим модель в режим оценки. Модель используется только для предсказаний и не обновляет веса

test_predictions = []
test_img_paths = []

start_time = time.time()
#  В этом цикле проходятся данные тестового набора данных (test_dataloader). В каждой итерации цикла извлекаются входы (inputs), метки (labels) и пути к изображениям (paths).
for inputs, labels, paths in tqdm(test_dataloader):
    inputs = inputs.to(device)      # Входные данные перемещаются на GPU
    labels = labels.to(device)
    with torch.set_grad_enabled(False):
        preds = model(inputs)      # Предсказываем
    test_predictions.append(       # Полученные предсказания подвергаются обработке и сохраняются в список.

 # В данном случае, предсказания подвергаются softmax-преобразованию и выбирается вероятность, соответствующая классу с индексом 1 (в случае, если у вас два класса). Затем значения переводятся на CPU и сохраняются в виде массива numpy.
        torch.nn.functional.softmax(preds, dim=1)[:,1].data.cpu().numpy())
    test_img_paths.extend(paths)               # Осуществляется сохранение путей к изображениям в список

test_predictions = np.concatenate(test_predictions)      # Накопленные предсказания объединяются в единый массив
elapsed_time = time.time() - start_time
print('Время работы нейросети на тестовой выборке:', elapsed_time)
```

Выведем результат работы нейросети на тестовой выборке.

```
inputs, labels, paths = next(iter(test_dataloader))

for img, pred in zip(inputs, test_predictions):
    show_input(img, title=pred)
```

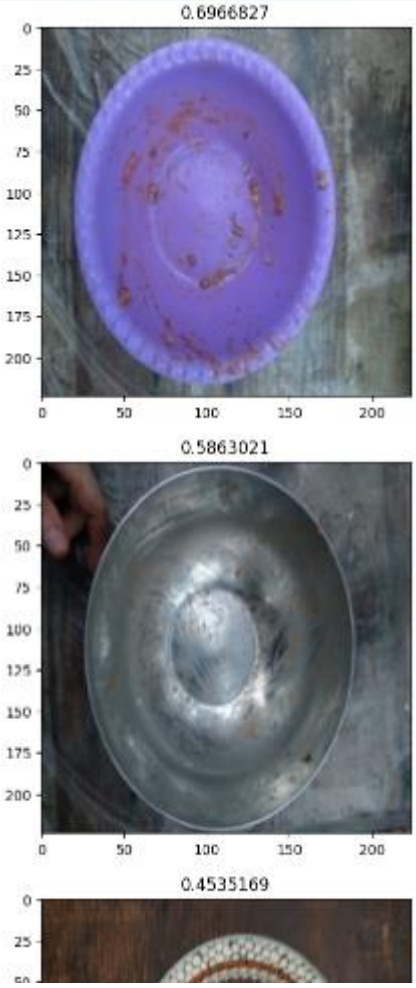

Рисунок 2 – Результат работы нейросети на тестовой выборке

Сравнение быстродействия

В указанном выше коде параллельная обработка данных на GPU осуществляется следующим образом:

1. Переменная device определяется наличием графического процессора (GPU) с поддержкой CUDA, с помощью команды torch.cuda.is_available(). Если GPU доступно, переменная device устанавливается в «cuda:0», указывая использовать GPU с индексом 0. Если GPU недоступно, переменная device устанавливается на «cpu», указывая на то, что используется центральный процессор (CPU).

2. Затем модель перемещается на указанное устройство с помощью команды .to(device). Это позволяет выполнить операции обучения и предсказания на указанном устройстве.

3. Во время обработки данных (в прямом и обратном проходах) данные inputs и labels также перемещаются на устройство с помощью .to(device). Это позволяет выполнить вычисления и операции обратного распространения на GPU.

Общий процесс обработки данных на каждом потоке GPU включает операции, связанные с вычислением предсказаний модели, вычислением потери, расчетом градиентов и обновлением весов модели во время обратного прохода. Каждый поток будет независимо выполнять данные операции на своей части данных, чтобы максимально использовать параллелизм и вычислительную мощность, доступную на GPU.

Количество тензоров изображений, которые обрабатывает каждый поток на GPU, определяется размером пакета данных (batch size), который передается модели.

Работа была выполнена в облачной среде Google Colaboratory. Расчеты выполнялись на CPU Intel(R) Xeon(R) с тактовой частотой 2.20 ГГц и на GPU Nvidia Tesla T4, который содержит 2560 CUDA-ядер для параллельных вычислений.

Исследуем быстродействие программы на CPU и на GPU. Зафиксируем время обучения нейросети на 100 эпохах на каждом из устройств.

Таблица 1 – Оценка быстродействия

	CPU Intel(R) Xeon(R) @ 2.20GHz	GPU Nvidia Tesla T4
Время обучения, с	454.933	141.777

Из таблицы 1 видно, что вычисления на GPU произвелись примерно в 3 раза быстрее, чем на CPU.

Заключение

В ходе выполнения курсового проекта были получены навыки работы с модулем PyTorch для реализации параллельной обработки набора данных с изображениями на языке Python.

Исходный датасет был разбит на обучающую и валидационную выборку изображений, которые впоследствии были трансформированы и преобразованы в тензоры. Затем, предобученная на датасете ImageNet нейронная сеть ResNet18 была дообучена на подготовленных данных на CPU и на GPU соответственно. В результате оценки быстродействия, убедились в том, что на GPU нейросеть обучилась быстрее, чем на CPU.

Исходный код доступен по ссылке на GitHub: https://github.com/yamichnikita/Parallel-computing-Course-project .

Список источников

1. Community Prediction Competition. Cleaned vs Dirty V2// Kaggle.URL: https://www.kaggle.com/competitions/platesv2/data (дата обращения: 16.07.2023).

2. Курс: «Нейронные сети и компьютерное зрение»// Stepik.URL: https://stepik.org/course/50352/info (дата обращения: 20.07.2023).

3. PYTORCH DOCUMENTATION // pytorch.org. URL: https://pytorch.org/docs/stable/index.html (дата обращения: 22.07.2023).