МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Факультет (институт, филиал) №8 **Кафедра** №814

Специальность 02.04.02 **Группа** М8О-114М-22

Отчет по курсу
«Параллельные и распределенные вычисления»

Выполнила: Коротина Д. Е.

Группа: М8О-114М-22

Преподаватель: Кондаратцев В.Л.

Москва 2022

Функция Create_Model создается модель resnet18 с предтренировочными весами (predtrained=True). Затем, мы получаем число входных нейронов в последнем классифицирующем слое (num_ftrs). После мы меняем последний слой, делая число выходных нейронов равным числу классов.

Дальше создается функция ошибки и задается оптимизатор.

```
def Create_Model():
    model_ft = models.resnet18(pretrained=True) # загрузили модель
    num_ftrs = model_ft.fc.in_features # получили число выходных
нейронов в последнем слое
    model_ft.fc = nn.Linear(num_ftrs, 2) # заменили последний слой
(число входных нейронов осталось темже, а выход стал 2мя нейронами
поскольку у нас всего два класса)
    criterion = nn.CrossEntropyLoss() # создали функцию ошибки
    optimizer_ft = optim.Adam(model_ft.parameters(), lr=0.01) # создали
оптимизатор
    return model_ft, criterion, optimizer_ft
```

Класс MyDataset формирует кастомный датасет, как изначальные параметры задаются тип данных (type_data), image_dir – общий путь к данным, transforms – трансформации (аугментации) к картинкам, data – создаем pandas таблицу для дальнейшего доступа к изображениям и их меткам (лэйблам).

Generate_Dataframe – парсит исходные данные. work_dir - создали путь до директории с данными, data_frame - создали пустой лист, в него будем заполнять данными.

__getitem__ – из сформированного дата фрейма вытаскиваем пути до изображения и лэйбл текущей картинки. Затем применяем трансформации.

```
class MyDataset(Dataset):

    def __init__(self, type_data, img_dir, transforms=None):
        """
        type_data - тип формируемых данных (тренировочные или тестовые)
        img_dir - общий путь к данным
        transforms - трансформации (аугментации) к изображениям
        data - создаем pandas таблицу для дальнейшего доступа к
изобраднениям и их меткам (лэйблам)
        """
        self.type_data = type_data
        self.img_dir = img_dir
        self.transforms = transforms
        self.data = self.Generate_Dataframe()

    def Generate_Dataframe(self):
        """
        work_dir - создали путь до даирректории с данными
        data_frame - создали пустой лист, в него будем пихать данные
        all_paths - названия всех картинок в папке с медведями
        """
        work_dir = self.img_dir + '/' + self.type_data + '/'
        data_frame = list()

        all_paths = os.listdir(work_dir + 'Bears') #Названия всех
картинок в папке с медведями и добавили каждую картинку с ее меткой в
наш будующий датафрейм
        for it in all_paths:
            data_frame.append([work_dir + 'Bears' + '/' + it, 0])

        all_paths = os.listdir(work_dir + 'Pandas') # названия всех
картинок в папке с пандами и добавили каждую картинку с ее меткой в наш
будующий датафрейм
        for it in all_paths:
            data_frame.append([work_dir + 'Pandas' + '/' + it, 1])

        return pd.DataFrame(data_frame, columns=[0, 1]) # Возвращем то
что насобирали в фомате pandas

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
```

```
        path = self.data.iloc[idx, 0] # из первой калонки в нешй
таблице получили путь к изображению
        image = Image.open(path).convert('RGB') # загрузили изображение
        label = self.data.iloc[idx, 1] # из второй колонки поличили
метку этого изображения
        if self.transforms: # применили преобразования
            image = self.transforms(image)
        return image, label
```

Функция Create_Dataloader – создает наш кастомный датасет, при помощи заранее определенного класса MyDataset. Затем создается дата лоадер (позволяет вытаскивать сформированные данные из датасета) и наконец получаем конечный размер данных.

```
def Create_Dataloader(type_data, img_dir, transform=None,
shuffle=None):
    dataset = MyDataset(type_data=type_data, img_dir=img_dir,
transforms=transform) # сформировали кастомный датасет при помощи
класса MyDataset
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=30,
shuffle=shuffle)  # создали даталоадер
    dataset_sizes = len(dataset) # получили размер датасета
    return dataloader, dataset_sizes
```

Функция train – тренируем и валидируем нашу модель с использованием MPI. При тренировке данные перекидываются с нулевого процесса на остальные следующим образом:
по все процессам раскидывается по ⅓ данных из батча, каждый процесс принимает свои данные, прогоняет их через модель, на выходе получаем матрицу с вероятностями для каждого класса, для каждой картинки. Затем при помощи torch.max получаем максимальные индексы с вероятностями, эти индексы соответствуют нашим классам. Считается точность и ошибка модели на каждом батче, а затем по всем батчам.

Подробнее про даталоадер: даталоадер чтобы выдать батч размером 30 условно вызывает 30 раз метод __getitem__ из класса MyDataset, последовательно итерируясь по сформированным внутри данным, перед тем как выдать склеивает их вместе.

```
def train(model, criterion, optimizer, dataloader_train,
dataset_sizes_train, dataloader_test, dataset_sizes_test, my_rank,
num_epochs=10):

    best_score = 0.0 #лучшая точность модели
    for epoch in range(num_epochs): # итерируемся по заданному числу
эпох
        model.train() # перевод модели в состояни тренировки
        runing_loss = 0.0 # текущая ошибка
        score = 0 # текущая точность

        if my_rank == 0: # если ранг процесса 0, то отправляем данные
трем оставшимся процессам
            for image, label in tqdm(dataloader_train):
                for procid in range(1, p):
                    N = int(image.size(0) / 3) # чиселка при помощи
которой мы делим данные на три процесса
                    comm.send(image[(procid-1) * N : procid * N],
dest=procid, tag=0) # срезами отправляем частичку изображений каждому
процессу
                    comm.send(label[(procid-1) * N : procid * N],
dest=procid, tag=1) # срезами отправляем частичку меток каждому
процессу

        if my_rank != 0: # если ранг процесса не нулевой
            for _ in range(len(dataloader_train)): # поскольку мы
отправили какое то количество раз изображения и метки, то должны
столько же раз принять эти данные
                image = comm.recv(source=0, tag=0) # приняли
изображения
                label = comm.recv(source=0, tag=1) # приняли метки
                optimizer.zero_grad() # занулил градиенты ()
                out = model(image) # прогнали изображения через модель
                _, preds = torch.max(out, 1) # выбрали максимальную
вероятность для кажого изображения
```

```
                loss = criterion(out, label) # посчитали ошибку на
данном батче
                loss.backward() # посчитали градиенты для каждого
параметра сети
                optimizer.step() # сделали шаг в нашем многомерном
пространстве + изменили веса(парамтры) сети
                runing_loss += loss.item() * image.size(0) # посчитали
текущую ошибку
                score += torch.sum(preds == label.data) # посчитали
текущую точность
            epoch_acc = score.double() / (dataset_sizes_train / 3) #
посчитали точность на данной эпохе
            runing_loss = runing_loss / (dataset_sizes_train / 3) #
посчитали ошибку на данной эпохе
            print("Epoch of train:", epoch + 1, "score: [",
epoch_acc.item(), "], loss: [", runing_loss, "]", my_rank) # вывели
ранее расчитанные значения

        MPI.Comm.Barrier(MPI.COMM_WORLD) # собрали все процессы вместе

        score = 0
        runing_loss = 0.0
        model.eval() # перевели модель в режим тестирования

        with torch.no_grad():
            if my_rank == 0:
                for image, label in tqdm(dataloader_test):
                    for procid in range(1, processes):
                        N = int(image.size(0) / 3)
                        comm.send(image[(procid-1) * N : procid * N -
1], dest=procid, tag=0)
                        comm.send(label[(procid-1) * N : procid * N -
1], dest=procid, tag=1)

            if my_rank != 0:
                for _ in range(len(dataloader_test)):
                    image = comm.recv(source=0, tag=0)
                    label = comm.recv(source=0, tag=1)
                    out = model(image)
                    _, preds = torch.max(out, 1)
                    loss = criterion(out, label)
                    runing_loss += loss.item() * image.size(0)
                    score += torch.sum(preds == label.data)
```

```
                epoch_acc = score.double() / (dataset_sizes_test / 3)
                runing_loss = runing_loss / (dataset_sizes_test / 3)
                print("Epoch of val:", epoch + 1, "score: [",
epoch_acc.item(), "], loss: [", runing_loss, "]", my_rank)
            MPI.Comm.Barrier(MPI.COMM_WORLD)

        if my_rank != 0: # если ранг не 0, то сохраняем модели с лучшей
точностью и минимальной ошибкой
            if epoch == 0:
                best_loss = runing_loss
            if epoch_acc > best_score and runing_loss <= best_loss:
                best_score = epoch_acc
                best_loss = runing_loss
                torch.save(model.state_dict(),
f"./weights/model_{my_rank}.pth")
    return model
```

Собирает процессы вместе, пока все процессы не будут в этой точке код не продолжит выполняться. Смысл здесь в том, что один процесс может улететь вперед и либо выдать полную кашу, либо сломать программу.

```
MPI.Comm.Barrier(MPI.COMM_WORLD)
```

Создаем модель, на этот раз с непред тренировочными весами и изначально заданным количеством классов. Загружаем ранее сохраненные веса. Создаем функцию ошибки.

```
def LoadModel():

    model_ft = models.resnet18(pretrained=False, num_classes=2) #
создали непредобученную модель

model_ft.load_state_dict(torch.load(f'/content/weights/model_{my_rank}.
pth')) # загрузили ранее сохраненные веса
    criterion = nn.CrossEntropyLoss()
    return model_ft, criterion
```

Аналогично функции с тренировкой.

```
def test(model, criterion, dataloader_test, dataset_sizes_test):
    score = 0
    runing_loss = 0.0
    model.eval()
    result = 0

    with torch.no_grad():
        if my_rank != 0:
            for image, label in tqdm(dataloader_test): # прогоняем все
тестовые данные через обученные модели
                out = model(image)
                comm.send(out, dest=0, tag=0) # каждый процесс посылает
выход своей модели нулевому процессу
                if my_rank == 1:
                    comm.send(label, dest=0, tag=1) # т.к. нулевой
процесс не получил лэйблы, то их ему нужно отправить (например первый
мпроцессом)
                _, preds = torch.max(out, 1)
                loss = criterion(out, label)
                runing_loss += (loss.item() / 2) * image.size(0)
                score += torch.sum(preds == label.data)
            epoch_acc = score.double() / dataset_sizes_test
            runing_loss = runing_loss / dataset_sizes_test
            print("Test process ", my_rank, ": score: [",
epoch_acc.item(), "], loss: [", runing_loss, "]", my_rank)

        if my_rank == 0:
            result = 0
            for _ in range(len(dataloader_test)): # принимаем столькоже
раз сколько и послали
                result_tmp = 0
                label = comm.recv(source=1, tag=1) # нулевой процесс
принял лэйблы от первого процесса
                for procid in range(1, p): # здесь принимаются выходы
сетей для кажого из процессов
                    out = comm.recv(source=procid, tag=0)
                    if procid == 1:
                        result_all_models = out
                    else:
                        result_all_models += out # все выходы
сладываются вместе
                result_all_models /= 3 # и устредняются
                _, preds = torch.max(result_all_models, 1)
```

```
                result += torch.sum(preds == label.data)
            result = result.double() / dataset_sizes_test
            print("Test process ", my_rank, ": score: [",
result.item(), "]") # выводится итоговый результат
```