cifar10の動かし方メモ

前提条件

実行ディレクトリの移動

現状のディレクトリが以下の通りになっておりことを想定。

```
ubuntu@ip-172-31-44-249:~/tmp/ml-system-in-actions$ pwd
/home/ubuntu/tmp/ml-system-in-actions
```

cifar10のフォルダがあるところまでディレクトリの移動

```
ubuntu@ip-172-31-44-249:~/tmp/ml-system-in-actions$ cd ./chapter2_training/cifar10
ubuntu@ip-172-31-44-249:~/tmp/ml-system-in-actions/chapter2_training/cifar10$ pwd
/home/ubuntu/tmp/ml-system-in-actions/chapter2_training/cifar10
```

##　解説

このディレクトリで主に扱うコマンドは、makefileに記されている。

例えば、

```
make dev
```

これで、makefileの中にある、"pip install -r requirements.txt"が実行される。

Makefileで用意されているのは以下の四つのコマンド

```
# pip install を自動で行う
make dev

#コンテナの立ち上げなど
make d_build

#モデルの学習
make train

# mlflowのUIをブラウザで立ち上げることができる
make ui
```

ここでは、4つのコマンドの動き方について記録を残しておく。

"make dev"の中身

これについては、pip install -r ~.txtと同様なので、特に情報はない

"make d_build"の中身

実行するコマンドとしては、以下の通り。

```
docker build -t [イメージ名]:[タグ名] -f [Dockerfileのパス] .
```

(詳細については、こちらのリンクを参照のこと
https://docs.docker.jp/engine/reference/commandline/build.html)

基本的に、Dockerを現場で使うには、imageと呼ばれるものから、Docker containerを立ち上げる必要がある。この際、やり方としては、以下の二つがある。

コンテナから作る場合

Docker Hubからイメージを"docker run"(もしくは docker pull)コマンドよりインストールしてきて、コンテナを起動し、そのコンテナに対して、"docker exec"などでシェルに入って操作したり、"docker cp"でファイルのコピーを行い、調整を行う。そして、最終的に"docker commit"コマンドによって、イメージ化を行うことで変更を記録する方法である。

Dockerfileを作成する場合

ベースとなるイメージと、そのイメージに対してどのような操作をするかを記録した、Dockerfileと呼ばれるファイルを用意して、Dockerfile通りにコンテナに対して変更やファイルのコピーを加えることによってイメージの作成を行う。(Docker Hubなどからインストールしたイメージに対して、事前にどのような変更を加えるかをDockerfileで決めておくということ、そしてその変更が定式化されるということ)

つまり、Dockerfileを作成するということは、材料が何で、どのような調理をするかを記した、レシピに該当し、変更などについて一覧性を持つということ。

基本的には、Dockerfileを作成しましょう。(なんの変更が加わったのか、確認出来ないのは怖い)

Dockerfileの中身

```
FROM python:3.8-buster  #ベースとなるイメージの指定

ENV PROJECT_DIR /mlflow/projects #環境変数の設定
ENV CODE_DIR /mlflow/projects/code #同じく
WORKDIR /${PROJECT_DIR}

# 環境変数"PROJECT_DIR"の中のディレクトリに移動

ADD requirements.txt /${PROJECT_DIR}/

# ローカル(EC2)上の"requirement.txt"を"python:3.8-buster"のディレクトリに移動

#以下のコマンドを実行
RUN apt-get -y update && \
    apt-get -y install apt-utils gcc && \
```

```
    pip install  --no-cache-dir -r requirements.txt

#作業ディレクトリの指定
WORKDIR /${CODE_DIR}
```

"make d_build"を実行すると、以下のエラー文が表示されるときがある。特に問題はない。
(https://mokuzine.net/ubuntu-apt-util-warn/)

make train

学習パイプラインの実行

```
$ make train
# 実行されるコマンド
# mlflow run . --no-conda
```

"mlflow run . --no-conda"を実行する。"--no-conda"を追加するのは、"mlflow"は"conda"でインストールされたパッケージをデフォルトで実行するから。pipでインストールされたパッケージを使用する際にはこちらを利用する。

ここで実行しているのは、同じディレクトリ内にある、"MLproject"のファイルである。“MLproject”の中には、学習手順などの一連の動作について一括で実行できるように記述されており、これを"mlflow run"で呼び出すことで、他の実験者が中のコードを意識することなく実行することができる。

"mlflow run . --no-conda"実行後

このコマンドを実行すると、MLprojectのファイルを実行する。

```
name: cifar10_initial

entry_points:
  main:
    parameters:
      preprocess_data: {type: string, default: cifar10}
      preprocess_downstream: {type: string, default: /opt/data/preprocess/}
      preprocess_cached_data_id: {type: string, default: ""}

      train_downstream: {type: string, default: /opt/data/model/}
      train_tensorboard: {type: string, default: /opt/data/tensorboard/}
      train_epochs: {type: int, default: 1}
      train_batch_size: {type: int, default: 32}
      train_num_workers: {type: int, default: 4}
      train_learning_rate: {type: float, default: 0.001}
      train_model_type: {type: string, default: vgg11}

      building_dockerfile_path: {type: string, default: ./Dockerfile}
      building_model_filename: {type: string, default: cifar10_0.onnx}
      building_entrypoint_path: {type: string, default:
```

```
  ./onnx_runtime_server_entrypoint.sh}

      evaluate_downstream: {type: string, default: ./evaluate/}
    command: |
      python -m main \
        --preprocess_data {preprocess_data} \
        --preprocess_downstream {preprocess_downstream} \
        --preprocess_cached_data_id {preprocess_cached_data_id} \
        --train_downstream {train_downstream} \
        --train_tensorboard {train_tensorboard} \
        --train_epochs {train_epochs} \
        --train_batch_size {train_batch_size} \
        --train_num_workers {train_num_workers} \
        --train_learning_rate {train_learning_rate} \
        --train_model_type {train_model_type} \
        --building_dockerfile_path {building_dockerfile_path} \
        --building_model_filename {building_model_filename} \
        --building_entrypoint_path {building_entrypoint_path} \
        --evaluate_downstream {evaluate_downstream}
```

これは、main.pyを実行する上で、必要な引数を指定したうえで実行している。具体的には、

```
command: | python -m main ...
```

によって、pythonを呼び出し、mainを実行することを示している。また、"..."のところでは、オプションの引数を指定している。(ArgumentParser (argparse)の使い方については、こちらを参照：https://qiita.com/kzkadc/items/e4fc7bc9c003de1eb6d0#%E5%9F%BA%E6%9C%AC%E5%BD%A2)

main.pyの実行内容

MLprojectから呼び出す形でmain.pyが実行された場合、引数を".add_argument"で読み取りつつ、以下のコマンドを実行する。

```
    # 引数の引き渡し及び実験IDの習得
    args = parser.parse_args()
    mlflow_experiment_id = int(os.getenv("MLFLOW_EXPERIMENT_ID", 0))

    # preprocessフォルダの実行
    with mlflow.start_run() as r:
        preprocess_run = mlflow.run(
            uri="./preprocess",
            entry_point="preprocess",
            backend="local",
            parameters={
                "data": args.preprocess_data,
                "downstream": args.preprocess_downstream,
                "cached_data_id": args.preprocess_cached_data_id,
            },
        )
```

```
        preprocess_run =
mlflow.tracking.MlflowClient().get_run(preprocess_run.run_id)

        dataset = os.path.join(
            "/tmp/mlruns/",
            str(mlflow_experiment_id),
            preprocess_run.info.run_id,
            "artifacts/downstream_directory",
        )

        train_run = mlflow.run(
            uri="./train",
            entry_point="train",
            backend="local",
            parameters={
                "upstream": dataset,
                "downstream": args.train_downstream,
                "tensorboard": args.train_tensorboard,
                "epochs": args.train_epochs,
                "batch_size": args.train_batch_size,
                "num_workers": args.train_num_workers,
                "learning_rate": args.train_learning_rate,
                "model_type": args.train_model_type,
            },
        )
        train_run = mlflow.tracking.MlflowClient().get_run(train_run.run_id)
        print("status: {}".format(train_run.info.status))

        building_run = mlflow.run(
            uri="./building",
            entry_point="building",
            backend="local",
            parameters={
                "dockerfile_path": args.building_dockerfile_path,
                "model_filename": args.building_model_filename,
                "model_directory": os.path.join(
                    "mlruns/",
                    str(mlflow_experiment_id),
                    train_run.info.run_id,
                    "artifacts",
                ),
                "entrypoint_path": args.building_entrypoint_path,
                "dockerimage": f"shibui/ml-system-in-
actions:training_pattern_cifar10_evaluate_{mlflow_experiment_id}",
            },
        )
        building_run = mlflow.tracking.MlflowClient().get_run(building_run.run_id)
        print("status: {}".format(building_run.info.status))

        evaluate_run = mlflow.run(
            uri="./evaluate",
            entry_point="evaluate",
            backend="local",
            parameters={
```

```
                "upstream": os.path.join(
                    "../mlruns/",
                    str(mlflow_experiment_id),
                    train_run.info.run_id,
                    "artifacts",
                ),
                "downstream": args.evaluate_downstream,
                "test_data_directory": os.path.join(
                    "../mlruns/",
                    str(mlflow_experiment_id),
                    preprocess_run.info.run_id,
                    "artifacts/downstream_directory/test",
                ),
                "dockerimage": f"shibui/ml-system-in-
actions:training_pattern_cifar10_evaluate_{mlflow_experiment_id}",
                "container_name":
f"training_pattern_cifar10_evaluate_{mlflow_experiment_id}",
            },
        )
        evaluate_run = mlflow.tracking.MlflowClient().get_run(evaluate_run.run_id)
        print("status: {}".format(evaluate_run.info.status))
```

このmain.pyを実行することで、一連の処理を完結させている。 具体的に中身についてみてみる。

preprocess フォルダの実行について

まずは、実行手順の最初に行われる、preprocessフォルダの中身について確認する。

```
    # preprocessフォルダの実行
    with mlflow.start_run() as r:
        preprocess_run = mlflow.run(
            uri="./preprocess",
            entry_point="preprocess",
            backend="local",
            parameters={
                "data": args.preprocess_data,
                "downstream": args.preprocess_downstream,
                "cached_data_id": args.preprocess_cached_data_id,
            },
        )
        preprocess_run =
mlflow.tracking.MlflowClient().get_run(preprocess_run.run_id)
```

ここでは、mlflow.start_run()がwith構文で実行されており、これはrunIDと呼ばれるものを発行している。その下のmlflow.run()は前述の"mlflow run ."と同じで、中身についてはMLprojectの書き方と同じ。簡単なTrackingについてのチュートリアルはこちらを参照のこと(https://future-architect.github.io/articles/20200626/)

ここでは、./preprocessのディレクトリにある、MLprojectを実行している。

./preprocess/MLprojectの概要

コードは以下の通り。 "docker_env"の項目では、今回コンテナ側で実行することを見越して、imageやボリュームの指定を行っている。

```
name: cifar10_initial

docker_env:
  image: shibui/ml-system-in-actions:training_pattern_cifar10_0.0.1
  volumes: ["$(pwd)/data:/opt/data", "/tmp/ml-system-in-
actions/chapter2_training/cifar10/mlruns:/tmp/mlruns"]

entry_points:
  preprocess:
    parameters:
      data: {type: string, default: cifar10}
      downstream: {type: string, default: /opt/data/preprocess/}
      cached_data_id: {type: string, default: ""}
    command: |
      python -m src.preprocess \
        --data {data} \
        --downstream {downstream} \
        --cached_data_id {cached_data_id}
```

指定されたイメージ（shibui/ml-system-in-actions:training_pattern_cifar10_0.0.1）は、すでに"docker d_build"の際に作ったもの。

```
ubuntu@ip-172-31-35-241:~/tmp/ml-system-in-actions/chapter2_training/cifar10$ docker image ls
REPOSITORY                    TAG                          IMAGE ID       CREATED       SIZE
cifar10_initial               4fb1b3e                      e4e1fa743a00   2 days ago    3.51GB
shibui/ml-system-in-actions   training_pattern_cifar10_0.0.1   d57ed627bfc7   2 days ago    3.51GB
python                        3.8-buster                   b5e1eb14396b   10 days ago   884MB
```

その後、"src"フォルダ内のpreprocessファイルの中身にアクセス。

注意事項

基本的にはCPUの性能と、メモリが一定以上必要。また、デフォルトでUbuntu状で動かす際には、trainの際に使うMLprojectのファイルを書き換える必要がある。

```
docker_env:
  image: shibui/ml-system-in-actions:training_pattern_cifar10_0.0.1
  volumes: ["$(pwd)/data:/opt/data", "$(builtin cd ..; pwd)/mlruns:/tmp/mlruns"]
```

このように変更して、bindのマウントの設定を変更

参考文献