Serverless

Introduction

The goal of this project is to deploy a ML inference app on Openwhisk. The app can be viewed as a function

$$\mathbf{Result} = Infer(\mathbf{image})$$

Under the Openwhisk limit, the image should be passed to funtion in URL.

The experiment setting is as follows

- Environment: Ubuntu 20.04LTS
- Platform: Windows Hyper-V, x86_64

> The IP address of target VM is `192.168.1.82`

Getting OpenWhisk ready

In this section, we deploy a **standalone** openwhisk stack on target machine.

Install Docker

We manually install docker using convenient script from get-docker.com

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
```

To accelerate download, we add our own registory to `/etc/docker/daemon.json`

```
$ sudo tee /etc/docker/daemon.json <<-'EOF'
{
```

```
  "registry-mirrors": ["https://p3e80qty.mirror.aliyuncs.com"]
}
EOF
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

Install JAVA

```
$ sudo apt-get install openjdk-11-jdk
```

Install NodeJS, NPM

```
$ sudo apt-get install nodejs
$ sudo apt-get install npm
```

Get OpenWhisk source code and Configure

You will find OpenWhisk's source code on Github. We clone the code and build it with gradlew.

```
$ git clone --recursive https://github.com/apache/openwhisk.git
$ cd openwhisk
$ ./gradlew core:standalone:build
```

Trouble shooting: Cannot connect to docker daemon due to privillege

```
$ sudo usermod -aG docker $USER
$ newgrp docker
```

If the `docker` group does not exist:

```
$ sudo groupadd docker
```

Restart the docker service

```
$ sudo systemctl restart docker
```

Verify normal user can use docker

```
$ docker run hello-world
```

Stop the gradle daemon, then try again to deploy

```
$ ./gradlew --stop
$ ./gradlew core:standalone:build
```

Get OpenWhisk CLI

To connect and manipulate OpenWhisk, `wsk` CLI is needed. We install `wsk` from openwhisk-cli repository:

```
$ wget https://github.com/apache/openwhisk-cli/releases/download/1.2.0/OpenWhisk_CLI-1.2.0-linux-amd64.tgz
$ tar -xzvf OpenWhisk_CLI-1.2.0-linux-amd64.tgz
$ mv wsk /usr/local/bin/
```

> The `amd64` postfix should be replaced with proper platfrom

Launch OpenWhisk

After the build process has finished, launch openwhisk using `java -jar`:

```
$ java -jar ./bin/openwhisk-standalone.jar
```

> This is a foreground task, do not close the terminal or the OpenWhisk will stop

Credential of our OpenWhisk instance is printed to `stdout` (as highlighted in the picture above). Use this credential to setup `wsk` cli.

```
$ wsk property set --apihost 'http://172.17.0.1:3233' \
                   --auth ...
```

Now we can manipulate OpenWhisk with `wsk` command.

Verify OpenWhisk Installation

We create a `hello.js` file to test if our OpenWhisk installation is operational

```
$ echo "function main(params){" \
        "var name = params.name || 'World';" \
        "return {payload:  'Hello, ' + name + '\!'};}" > hello.js
```

We then create an OpenWhisk Action that binds to this script:

```
$ wsk action create helloworld hello.js
ok: created action helloworld
```

To invoke this action, simply run:

```
$  wsk action invoke helloworld --result --param name david
{
    "playload": "Hello, david!"
}
```

Deploy MinIO, an Open-source Object Storage Service

To store pictures of handwritten digits, we need some sort of storage services. The `MinIO` server is a decent choice. It's an object storage that support url sharing.

```
$ docker run \
    --name minio \
    --net=host \
```

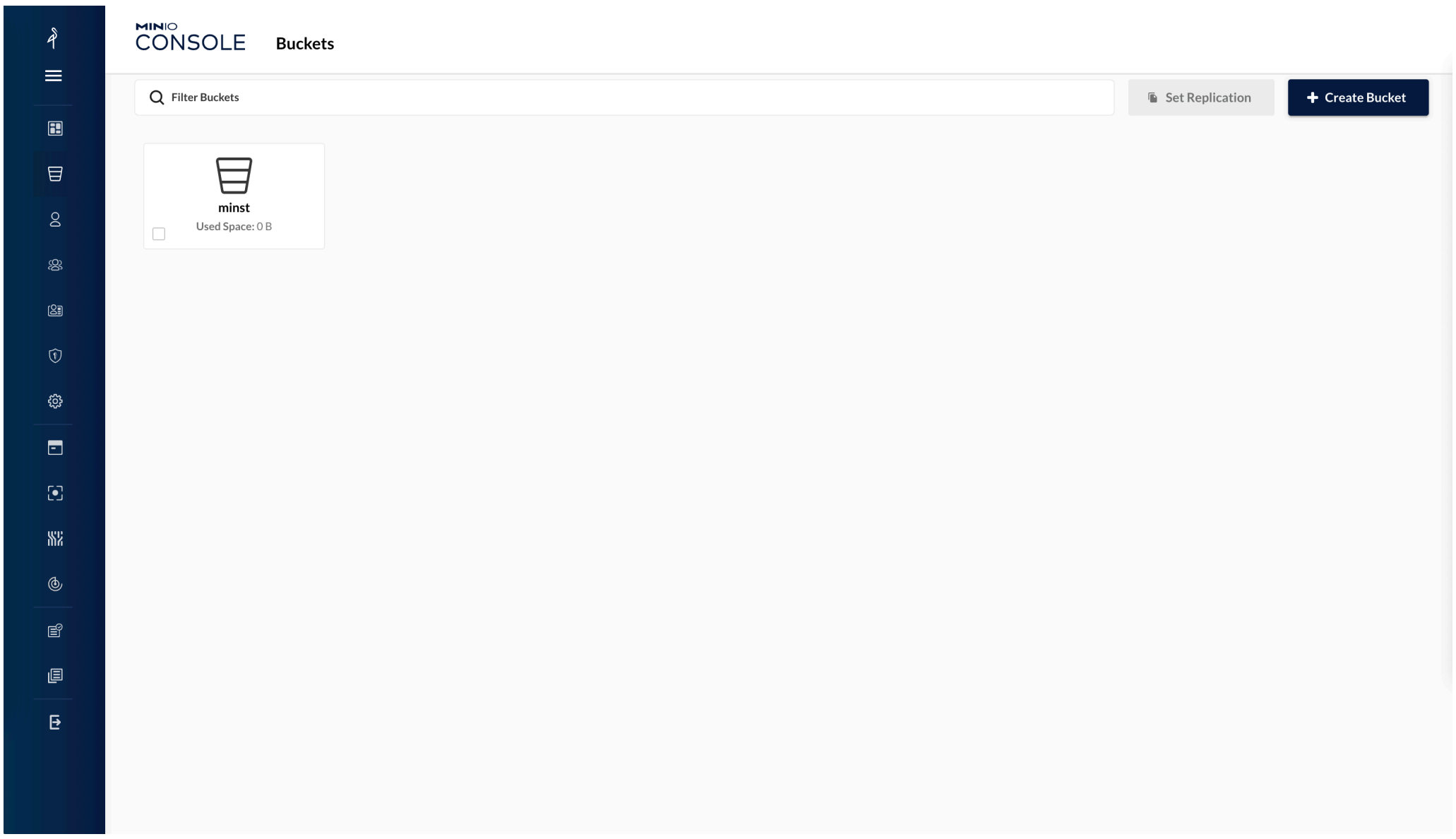
```
    -d \
    -v $PATH_TO_DATA:/data \
    -v $PATH_TO_CONFIG:/root/.minio \
    -e "MINIO_ROOT_USER=$CUSTOM_ROOT_USER" \
    -e "MINIO_ROOT_PASSWORD=$CUSTOM_ROOT_PASSWORD" \
    minio/minio \
    server /data --console-address $ADDRESS
```

For example:

```
$ docker run \
    --name minio \
    --net=host \
    -d \
    -v /home/speit/minio-data:/data \
    -v /home/speit/minio-config:/root/.minio \
    -e "MINIO_ROOT_USER=root" \
    -e "MINIO_ROOT_PASSWORD=password" \
    minio/minio \
    server /data --console-address "0.0.0.0:41309"
```

Using the WebUI of MinIO, we create :

1. A bucket named `minist`
2. A new credential `testAccessKey:testSecretKey`



An upload function is created to upload photos to this OSS service:

```
"""file_uploader.py
"""
from minio import Minio
import argparse
import uuid
from typing import Union, Dict
import os
from datetime import timedelta


def oss_upload(endpoint: str,
               bucket_name: str,
```

```
            access_key: str,
            secret_key: str,
            file: str,
            object_name: str = '') -> Union[None, Dict[str, str]]:
    """Upload a file to MinIO OSS

    Args:
        args (ArgumentParser): should contain following attributes:
            - args.endpoint: str, url
            - args.access_key: str
            - args.secret_key: str

    Returns:
        Dict[str, str]: {$BUCKET:$NAME}
    """
    minioClient = Minio(endpoint,
                        access_key=access_key,
                        secret_key=secret_key,
                        secure=False)  # Create MinIO client
    try:
        if len(object_name) <= 0:
            object_name = str(uuid.uuid1()) + os.path.splitext(file)[
                -1]  # Generate unique object name
        else:
            object_name = object_name

        minioClient.fput_object(bucket_name, object_name,
                                file)  # Upload object
        url = minioClient.presigned_get_object(bucket_name,
                                               object_name,
                                               expires=timedelta(days=2))
        ret = {
            "bucket_name": bucket_name,
            "object_name": object_name,
            "url": url
        }  # Return the object info
        return ret

    except Exception as err:
        print(err)
        return None


if __name__ == '__main__':
    """Usage

    python file_upload.py --endpoint=192.168.1.82:9000 \
                          --access_key=testAccessKey \
                          --secret_key=testSecretKey \
                          --bucket=mnist \
                          --file=mnist.png
    """
    parser = argparse.ArgumentParser()
    parser.add_argument('--endpoint', type=str)
    parser.add_argument('--access_key', type=str)
    parser.add_argument('--secret_key', type=str)
    parser.add_argument('--bucket_name', type=str)
    parser.add_argument('--file', type=str)
    parser.add_argument('--object_name', type=str, default='')
    args = parser.parse_args()
    print(
        oss_upload(endpoint=args.endpoint,
                   bucket_name=args.bucket_name,
                   access_key=args.access_key,
                   secret_key=args.secret_key,
                   file=args.file,
                   object_name=args.object_name))
```

It returns the name of bucket and the name of file as a dictionary, together with an encrypted url:

```
{
    "bucket_name": "mnist",
    "object_name":"082d97b2-19f1-11ec-a558-1e00d10c4441.png",
    "url":"http://192.168.1.82:9000/mnist/082d97b2-19f1-11ec-a558-1e00d10c4441.png?..."
}
```

Deploy ML Applications

Train the Model

We follow this paper to build a multi-layer CNN model using Pytorch. The model is trained with MNIST dataset.

Convert the model to ONNX

With the help of `torch.onnx.export`, we convert our trained model to ONNX format for deployement.

Wrap the ML model as a WebApp

With the help of Flask, the ML model can be wrapped as a WebApp.

> See `deploy-flask.py` for details

The WebApp we designed can be accessed by HTTP GET/POST actions. For example:

```
$ curl -X POST -d '{"value":{"url":"http://192.168.1.82:9000/mnist/test_picture.png"}}' \
       -H 'Content-Type: application/json' http://localhost:8080/run
{"code":200,"res":6}
```

The app

1. Init the model
2. Get image from url
3. Generate prediction

These information can be obtained from `upload()` function.

If the requested object does not exist, or is not readable, the service returns `{"code":500, "res":-1}`

Create Docker Image for this ML application

OpenWhisk support Docker actions, but has several limits. According to OpenWhisk Docker Actions:

- The action should offer a RESTful API at port `8080`
- The action should offer two api
    i. HTTP POST at `/init`
    ii. HTTP POST at `/run` with JsonData

`/init` API will be called to set envrionment variables. It is automatically called by OpenWhisk in the prewarm stage. After the prewarm, each invoke of action will simple pass parameters to `/run` in a specific JSON format:

```
{
    "value":{ // OpenWhisk default
        "param1":"value1",
        "param2":"value2"
    }
}
```

We come up with the following Dockerfile to build our custom ML image:

- The image is based on python:3.8.8-slim to reduce size
- It installs flask, pytorch and other neccessary packages
- It launches our flask ML application

```
FROM python:3.8.8-slim

ENV LANG=C.UTF-8

WORKDIR /opt/app/

RUN pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple && \
    pip install minio pillow tqdm flask gevent requests onnx onnxruntime

ENV PATH_TO_ONNX_MODEL=/opt/app/model.onnx

COPY ["model.onnx", "deploy-flask.py", "/opt/app/"]

EXPOSE 8080

CMD ["/bin/bash", "-c", "python /opt/app/deploy-flask.py"]
```

> Known Issus:
>
> opehwhisk/python3action-base is based on `alphine`, which is not shipped with `glibc` but with `muslc`, an replacement of standard C library. Therefore we cannot easily run `miniconda` (which requires `glibc`) on it.

We build the image using this command:

```
$ docker build . -t python3action-mnist
```

Test run:

```
$ docker run -it --rm --net=host python3action-mnist
```

OpenWhisk docker actions requires docker image to be public, therefore we need to push our image to docker hub:

```
$ docker tag python3action-mnist $USER_NAME/python3action-mnist:1.0
$ docker login
$ docker push $USER_NAME/python3action-mnist:1.0
```

> It is extremely dangerous to store credentials in python script. Consider using environment variables to store access_key and secret_key

> This image runs on `x86_64`. **NOT** `arm`.
>
> For docker cross-platform build, visit this site
>
> ```
> docker buildx create --use --name m1_builder
> docker buildx build . --platform linux/amd64 -t python3action-mnist --load
> ```

Test API

```
$ curl http://localhost:8080/init
OK
```

```
$ curl -X POST -d '{"value":{"url": "http://192.168.1.82:9000/mnist/test_picture.png"}}' \
       -H 'Content-Type: application/json' http://localhost:8080/run
{"code":200,"res":6}
```

Configure OpenWhisk docker action

Create action by:

```
$ wsk action create mnist --docker natrium233/python3action-mnist:1.0
```

If the action is created, use the following command to update

```
$ wsk action update mnist --docker natrium233/python3action-mnist:1.2
```

Invoke action with following command

```
$ wsk action invoke mnist --result --param url "http://192.168.1.82:9000/mnist/test_picture.png"
{
    "code": 200,
    "res": 6
}
```

Creating OpenWhisk API

> Reference adobeio-runtime

To use OpenWhisk via RESTful api, we have to launch our OpenWhisk stack with `--api-gw` enabled:

```
java -jar openwhisk/bin/openwhisk-standalone.jar --api-gw -p 3233
```

The api gateway will be created on port `3234` by default unless `--api-gw-port` is specified

We also need to modify action creation to support web interface:

```
$ wsk action create mnist --docker natrium233/python3action-mnist:1.2 --web true
ok: got action mnist
```

By running `wsk api` command, we enble RESTful access to our action:

```
$ wsk api create /ml /mnist post mnist --response-type json
ok: created API /ml/mnist POST for action /_/mnist
http://172.17.0.1:3234/api/23bc46b1-71f6-4ed5-8c54-816aa4f8c502/ml/mnist
```

> If the action is already created without `--web true`, `wsk action update "/_/mnist" --web true` need to be executed to update it

> `172.17.0.1` is docker bridge network IP. However, We can also access to API via `localhost` since API gateway is enabled.

Test this api with curl

```
$ curl -X POST -d '{"url":"http://192.168.1.82:9000/mnist/test_picture.png"}' \
       -H 'Content-Type: application/json' \
       http://localhost:3234/api/23bc46b1-71f6-4ed5-8c54-816aa4f8c502/ml/mnist
{
  "code": 200,
  "res": 6
}
```

Once firewall rules are configured, this service will be externally accessiable.

Summary

The workflow of using his ML service is:

1. Upload the image to OSS, get access url
2. Invoke OpenWhisk action with url
3. Process with reesult

4. Remove the uploaded image if needed.

A python script is created to accomplish this workflow

```
"""mnist_demo.py
"""
import argparse
from typing import Union, Dict
from file_uploader import oss_upload
import requests
import json
from minio import Minio
import uuid
import os
from datetime import timedelta


def invoke_openwsk_action_mnist(wsk_apihost: str, image_url: str):
    """Invoke openwsk action via RESTful API

    Args:
        wsk_apihost (str): url like string
        image_url (str): url to image
    """
    headers = {'Content-Type': 'application/json'}
    response = requests.post(url=wsk_apihost,
                             headers=headers,
                             data=json.dumps({"url": image_url}))
    return response.json()


if __name__ == '__main__':
    """Usage

    python file_upload.py --endpoint=192.168.1.82:9000 \
                          --access_key=testAccessKey \
                          --secret_key=testSecretKey \
                          --bucket=mnist \
                          --file=mnist.png
    """
    parser = argparse.ArgumentParser()
    parser.add_argument('--endpoint', type=str)
    parser.add_argument('--access_key', type=str)
    parser.add_argument('--secret_key', type=str)
    parser.add_argument('--bucket_name', type=str)
    parser.add_argument('--file', type=str)
    parser.add_argument('--object_name', type=str, default='')
    parser.add_argument('--mnist_api', type=str)
    args = parser.parse_args()

    minioClient = Minio(args.endpoint,
                        access_key=args.access_key,
                        secret_key=args.secret_key,
                        secure=False)  # Create MinIO client

    if len(args.object_name) <= 0:
        # Generate unique object name
        object_name = str(uuid.uuid1()) + os.path.splitext(args.file)[-1]
    else:
        object_name = args.object_name

    # Upload image
    minioClient.fput_object(args.bucket_name,
                            object_name,
                            args.file)
    img_url = minioClient.presigned_get_object(args.bucket_name,
                                               object_name,
                                               expires=timedelta(days=2))
```

```
    # Invoke OpenWhisk via api
    res = invoke_openwsk_action_mnist(args.mnist_api, img_url)
    print(res)

    # Delete the image
    minioClient.remove_object(args.bucket_name, object_name)
```

We can run the demo by passing arguments:

```
$ python mnist_demo.py \
    --endpoint=192.168.1.82:9000 \
    --access_key=testAccessKey \
    --secret_key=testSecretKey \
    --bucket_name=mnist \
    --file=test_picture2.png \
    --mnist_api=http://192.168.1.82:3234/api/23bc46b1-71f6-4ed5-8c54-816aa4f8c502/ml/mnist
{'code': 200, 'res': 4}
```

Appendix - Convenient setup script

see ICE6405P-260-M01

Distributed Training

Preface

> 这是一个示例应用程序 Dark vision，它就是这样做的。在此应用程序中，用户使用 Dark Vision Web 应用程序上载视频或图像，该应用程序将其存储在 Cloudant DB 中。视频上传后，OpenWhisk 通过听 Cloudant 更改（触发）来检测新视频。然后，OpenWhisk 触发视频提取器操作。在执行过程中，提取器将生成帧（图像）并将其存储在 Cloudant 中。然后使用 Watson Visual Recognition 处理帧，并将结果存储在同一 Cloudant DB 中。可以使用 Dark Vision Web 应用程序或 iOS 应用程序查看结果。除 Cloudant 外，还可以使用对象存储。这样做时，视频和图像元数据存储在 Cloudant 中，媒体文件存储在对象存储中。
>
> *from* *【无服务器架构】openwhisk 经典使用案例*

Project Design

We intend to use the OpenWhisk serviceless computing framework for distributed machine learning. In order to do this, we first need to set up a parameter server for saving model parameters. This server provides parameters for worker. This server can be implemented on a cluster for maximum performance, but given the simplicity of the experiment, we deploy it to a single machine.

Second, we need to define a serviceless function for training:

$$\mathtt{NewParameters} = Train(\mathtt{Model}, \mathtt{Parameters}, \mathtt{Dataset})$$

This function accepts the model, the model parameters, and the dataset as input and outputs the updated parameters. Considering the limitations of the OpenWhisk framework, in practice we fix the model to `LeNet5`, locate the dataset with a URL and pull the parameters from an external parameter server, hence the following pseudo code.

```
def Train(Model=LeNet5, ParameterServerURL, DatasetURL, TrainSettings):
    Dataset = GetFromURL(DatasetURL)
    Parameters = PullFromParameterServer(ParameterServerURL)
    TrainWithModel(Model, Parameters, Dataset, TrainSettings)
```

The Parameter Server

A parameter server is created by `Flask` to serve model parameters to workers. Its mission is to:

- Store model parameters in `model`, keep it versioned by updating `model.version`
- Respond to workers' request of latest model parameters via `/getParameter`
- Acceput gradient uploaded from workers from `/putGradient`
- Optimize model parameters according to gradient received

The api of this server is summarized as bellow:

API	Method	Type	Example
`/getVersion`	GET	application/json	{"code":200, "accuracy":0.0}
`/getParameter`	GET	application/octec_stream	serialized state dict {"code":200,"param":Dict[str, torch.Tensor]}
`/putGradient`	POST	application/octec_stream	serialized gradient dict {"id":0,"param":Dict[str, torch.Tensor]}
`/registerWorker`	POST	application/json	(beta){ "id":0,"description":"worker_0"}
`/unregisterWorker`	POST	application/json	(beta){ "id":0}

> `/registerWorker` and `/unregisterWorker` is under development. The interface should be protected by some sort of access/secret key pair and SSL encryption to avoid abuse of model parameters. On the other hand, sending pickle serialized object over http is not secure. We are aware of these vulnerabilities

Once updated, `model.version` will increase. The workers can check version of model via `/getVersion` api. If the model is updated on the parameter server, workers can choose to download latest parameters to local

Full code of parameter server can be found in `run_server.py`

The Docker Contained Worker

The worker will be launched by Openwhisk. Therefore, we need to create: two api `/init` and `/run`

API	Method	Type	Significance
`/init`	POST	None	Init NeuralNet
`/run`	POST	application/json	Execute trainning

Since we are building our own python action, `/init` api is insignificant:

```
@app.route("/init", methods=['GET', 'POST'])
def init():
    global g_net, g_train_dataset
    g_net = Net()
    g_net.train()
```

The `/run` api, on the other hand, need parameters that are critical to training. After experiments, we decideded that `/run` api will accept a json dictionary like this:

```
{
    "value": {
        "batch_sz_train": 32,
        "epoch_n": 32,
        "apihost": "http://192.168.1.131:29500",
        "update_intv": 8,
        "dataset_url": "http://192.168.1.131:9000/mnist-dataset/dataset_dl_1.tar.gz",
        "device": "cpu"
    }
}
```

Where the `value` key is **automatically added by OpenWhisk**.

Full code of worker can be found in `run_worker.py` and `worker_utils.py`

Preparing Dataset

The goal of this section is to make datasets accessible via URL

First, we split mnist dataset into 6 parts. A `dataset_dl` module is organized as follows :

```
./dataset_dl
├── __dataset__.py
├── __init__.py
└── data
    └── MNIST
        └── raw
            ├── t10k-images-idx3-ubyte
            ├── t10k-images-idx3-ubyte.gz
            ├── t10k-labels-idx1-ubyte
            ├── t10k-labels-idx1-ubyte.gz
            ├── train-images-idx3-ubyte
            ├── train-images-idx3-ubyte.gz
            ├── train-labels-idx1-ubyte
            └── train-labels-idx1-ubyte.gz
```

Explanation

The `__init__.py` import dataset object from `__dataset__.py`

```python
"""__init__.py"""
from .__dataset__ import dataset as Dataset
```

In `__dataset__.py`, the full MNIST dataset is truncated by index. The example below shows a subset of MNIST from 50000 to 60000 (10000 elements)

```python
"""__dataset__.py"""
import torch
import torchvision

_raw_dataset = torchvision.datasets.MNIST(
        './dataset_dl/data/',
        train=True,
        download=True,
        transform=torchvision.transforms.Compose([
            torchvision.transforms.ToTensor(),
            torchvision.transforms.Normalize((0.1307, ), (0.3081, ))
        ]))
# Split dataset using torch.utils.data.Subset api
dataset = torch.utils.data.Subset(_raw_dataset, range(50000, 60000))
```

> We admit this format as a paradigm. Any dataset can be wrapped and accessed by:
>
> ```python
> import dataset_dl
> dataset = dataset_dl.Dataset # a torch.utils.data.Datasset object
> ```
>
> without knowiing the detail of implementation

We then create 6 dataset, each dataset is a different subset of the whole MNIST dataset

```
$ tar -zcvf dataset_dl_1.tar.gz ./dataset_dl # Subset(0,10000)
$ tar -zcvf dataset_dl_2.tar.gz ./dataset_dl # Subset(10000,20000)
$ tar -zcvf dataset_dl_3.tar.gz ./dataset_dl # Subset(20000,30000)
$ tar -zcvf dataset_dl_4.tar.gz ./dataset_dl # Subset(30000,40000)
$ tar -zcvf dataset_dl_5.tar.gz ./dataset_dl # Subset(40000,50000)
$ tar -zcvf dataset_dl_6.tar.gz ./dataset_dl # Subset(50000,60000)
```

We upload these datasets to OSS with the help of previously created `file_uploader.py`, or via web interface of MinIO.

```
$ python file_uploader.py \
        --endpoint=192.168.1.131:9000 \
        --access_key=testAccessKey \
        --secret_key=testSecretKey \
        --bucket_name=mnist-dataset \
        --file=dataset_dl_1.tar.gz
```

```
{
  'bucket_name': 'mnist-dataset',
  'object_name': 'd96e3d6c-1ddf-11ec-9aeb-c3cd4bc871fd.gz',
  'url': 'http://192.168.1.131:9000/mnist-dataset/d96e3d6c-1ddf-11ec-9aeb-c3cd4bc871fd.gz...'
}
```

Summary of urls

Finally, we have a list of dataset that can be accessed via URL.

Volume	URL
1	http://192.168.1.131:9000/mnist-dataset/dataset_dl_1.tar.gz
2	http://192.168.1.131:9000/mnist-dataset/dataset_dl_2.tar.gz
3	http://192.168.1.131:9000/mnist-dataset/dataset_dl_3.tar.gz
4	http://192.168.1.131:9000/mnist-dataset/dataset_dl_4.tar.gz
5	http://192.168.1.131:9000/mnist-dataset/dataset_dl_5.tar.gz
6	http://192.168.1.131:9000/mnist-dataset/dataset_dl_6.tar.gz

> The dataset on OSS (data and `.py` descriptors) must be tar.gz archive.

Training

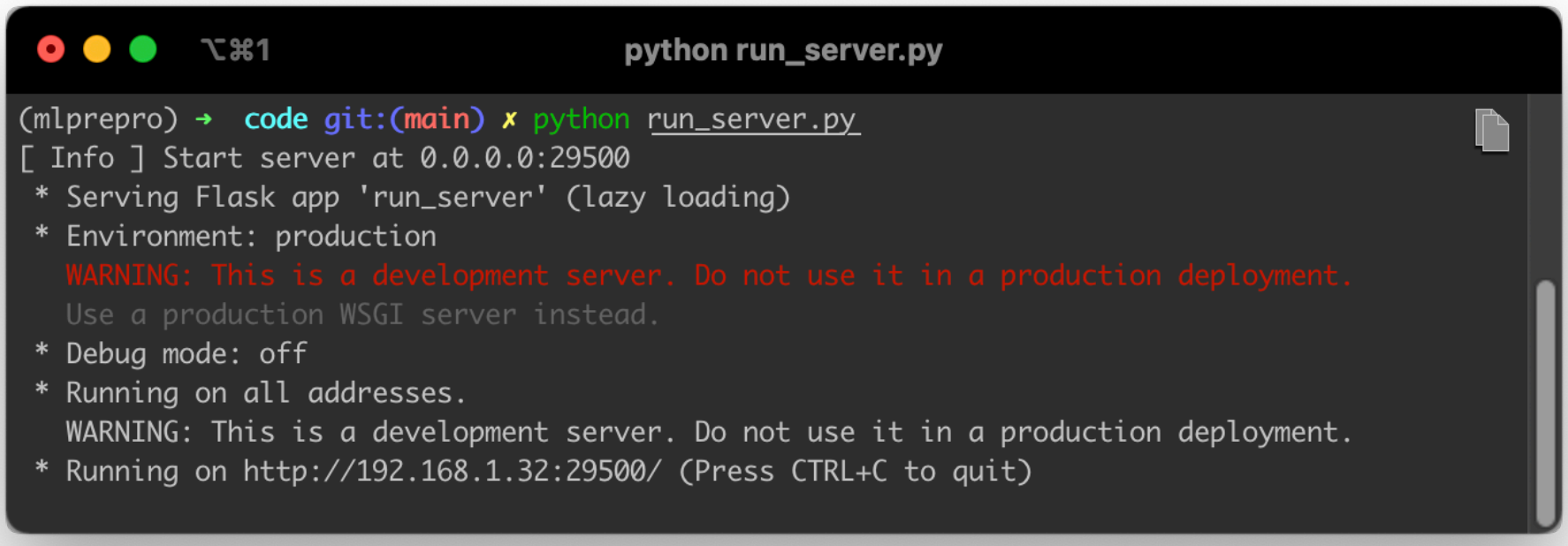
API test with curl

Before deployment, we first test distributed workers with `curl`.

We first fire up the MinIO instance with `docker start minio`. Then the parameter server via

```
$ python run_server.py
```

The parameter server will listen at port 29500



Then, we start worker via:

```
$ python run_worker.py
```

The `run_worker.py` will listen at port 8080. We need to use this API via `curl`.

```
#!/bin/bash
SERVER_HOST=http://172.17.0.1:29500
DATASET_URL=http://172.17.0.1:9000/mnist-dataset/dataset_dl_1.tar.gz
WORKER_HOST=http://localhost:8080
curl -X POST \
     -d '{"value":{"batch_sz_train": 32,
     "epoch_n": 32,
     "apihost": "'$SERVER_HOST'",
     "update_intv": 8,
     "dataset_url": "'$DATASET_URL'",
     "device": "cpu"}}' \
     -H 'Content-Type: application/json' $WORKER_HOST/run
```

> See `test-worker.sh`

The training loop should start. And we can observe client activities from server side

Build Docker Image

After local test is completed. We construct the docker image in which the worker runs

```
$ cd ./code # root of code
$ docker build . -t python3action-dist-train-mnist
```

After the build, we test the containner locally with curl

```
$ docker run --rm --net=host python3action-dist-train-mnist
$ curl ...
```

This will create a docker container that use host network. We test this container with the same method.After test, we tag the image and push it to docker registery

```
$ docker login
$ docker tag python3action-dist-train-mnist natrium233/python3action-dist-train-mnist:1.0
$ docker push natrium233/python3action-dist-train-mnist:1.0
```

Configure OpenWhisk

Create OpenWhisk action

```
$ wsk action create dist-train --docker natrium233/python3action-dist-train-mnist:1.0 \
                               --web true \
                               --timeout 300000
```

> This command assumes that openwhisk is configured with api-gateway, otherwise you should remove the `--web true` flag

To invoke this action, run

```
#!/bin/bash
SERVER_HOST=http://172.17.0.1:29500
DATASET_URL=http://172.17.0.1:9000/mnist-dataset/dataset_dl_1.tar.gz
wsk action invoke dist-train \
    --param batch_sz_train 32 \
    --param epoch_n 8 \
    --param apihost $SERVER_HOST \
    --param update_intv 8 \
    --param dataset_url $DATASET_URL \
    --param device cpu
```

The action will create workers, and we can monitor worker activity via parameter server.

> Modify SERVER_HOST and DATASET_URL on demand

> See `test-wsk-action.sh` for details

Changing server configuration

When setting local batch size to 32, the memory consumption is 166MiB, which does not exceed the default memory limit of OpenWhisk

```
$ docker stats
CONTAINER ID   NAME                       CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
5ceb5ca66318   wsk0_13_guest_disttrain    57.77%    165.8MiB / 256MiB     64.76%    33.7MB / 35.4MB
560ba865b754   wsk0_9_prewarm_nodejs14    0.00%     10.16MiB / 256MiB     3.97%     4.08kB / 0B
```

When invoking two actions at same time, multiple containers will be created:

```
CONTAINER ID   NAME                       CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
c35b1fee108c   wsk0_14_guest_disttrain    33.04%    166.5MiB / 256MiB     65.04%    30.6MB / 13.8MB
9cb88380585d   wsk0_15_guest_disttrain    33.64%    167.1MiB / 256MiB     65.28%    30.5MB / 13.6MB
15e384116b43   wsk0_16_prewarm_nodejs14   0.00%     10.39MiB / 256MiB     4.06%     3.28kB / 0B
```

When creating action, you can use `--memory` to limit the maximum memory a function can use, this value is `256MB` by default

```
wsk action create dist-train --docker natrium233/python3action-dist-train-mnist:1.0 \
                             --web true \
                             --timeout 300000 \
                             --memory 512
```

Summary

The 4GB virtual machine I created have 2.5GiB of free memory at idle stat. But since each activation consumes 33% of CPU, the virtual machine can only hold up to 3 workers at one time

If we insist to add workers on this machine, the trainning will be bottenecked by CPU

If we could deploy OpenWhisk to a Kubernetes cluster, the number of workers can increase. What is more, if the cluster had GPU installed and `nvidia-docker` installed, the traininig would be accelerated by GPU.

However, configuring `nvidia-docker` on a Kubernetes cluster is tredious.

Pay attention to OpenWhisk timeout policy. On a standalone server, an action must finish within its timeout (which is 300000 milliseconds maximum).