Deep Learning Model Training with FfDL running on Minishift

FfDL, fabric for deep learning, is an open source collaboration platform runs on public or private cloud and provides training, monitoring and management of deep learning models. FfDL runs as scalable microservices with Kubernetes, including RESTful API service, trainer service, training data service, and lifecycle management and learner pods. In fact, FfDL is one of the 2018 Best of Open Source Software Award winners in machine learning and deep learning. It is also the core of IBM Watson Machine Learning service.

With FfDL's rich support of deep learning frameworks, including TensorFlow, PyTorch, ONNX and more, data scientists can focus on developing the model training code, as well as compare the models trained with different frameworks. Better yet, FfDL can run locally on a Kubernetes cluster inside a VM. This gives users great flexibility and cost efficiency to develop the deep learning models before going to cloud.

In this tutorial, we will show how to deploy FfDL on Minishift.

Minishift runs Red Hat OpenShift Kubernetes platform locally. It uses OpenShift Origin to run the cluster. Data scientists can try out the development and training on a Minishift cluster before deploying and serving the models to an OpenShift cluster.

- Install Minishift cluster

We start with installation of a Minishift cluster on a local host. Our host machine has **CentOS 7.6** base Linux operating system.

The Getting Started with MiniShift has detailed documentation on how to install a Minishift cluster. It also includes the instructions to set up `virtualbox` or `KVM` hypervisors. Depends on the different base Linux platforms, installation of hypervisor are different. Sometime either the hypervisor installation fails or Minishift cluster fails to start with one of the hypervisor.

So instead of running a Minishift cluster with a hypervisor, following provides the step by step instruction to run the cluster with a `generic` hypervisor, also supported by Minishift.

1. Install docker

```
yum install docker -y
systemctl restart docker
```

2. Generate ssh key if not yet. The ssh key is needed to start the Minishift cluster

```
ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
ssh-copy-id root@localhost
```

3. Install Minishift executable following the link and choose a release, for example, *1.31.0*.

```
curl -L -o minishift.tgz
https://github.com/minishift/minishift/releases/download/v1.31.0/minishift-1.31.0-
linux-amd64.tgz
tar zxvf minishift.tgz
cp minishift*linux*/minishift /usr/local/bin
```

Once complete the simple steps above, go forward to starting the Minishift cluster with following command:

```
export VM_IP=<vm_ip_address>
minishift start --vm-driver=generic --remote-ipaddress=$VM_IP --remote-ssh-
user=root --remote-ssh-key=/root/.ssh/id_rsa --cpus <num_cpus> --memory
<memory_size> --disk-size <disk_size>
```

replace `<vm_ip_address>` with the IP address for the host, `<num_cpus>` with the number of cores to reserve for the cluster, `<memory_size>` with the max memory to be consumed by the cluster and `<disk_size>` with the size of the disk space for the cluster.

Note, during Minishift cluster is starting, it needs to download from some github repo, and if `403 API rate limit` error is observed, setting following environment variable and submit the command to start the cluster again.

```
export MINISHIFT_GITHUB_API_TOKEN=<github_access_token>
```

replace the `<github_access_token>` with a real github personal access token. Follow this link to create one if needed.

As part of the Minishift start process, OpenShift `oc` command is added to the `/var/lib/minishift/bin` directory, add the directory to the `PATH` environment variable.

```
export PATH=$PATH:/var/lib/minishift/bin
```

- Prepare one PersistentVolume for FfDL trainer pod

Minishift creates 100 `PersistentVolume`s when the cluster starts. We will claim one of these PVs to store the FfDL training job info.

To work around an OpenShift `subPath` issue as described here, we will remount the local `tmpfs` file system used by Minishift for storing data.

```
mount --bind --make-rshared /var/lib/minishift/base/openshift.local.pv/pv0001
/var/lib/minishift/base/openshift.local.pv/pv0001
```

Restart the Minishift cluster

```
minishift stop
minishift start --vm-driver=generic --remote-ipaddress=$VM_IP --remote-ssh-
```

```
user=root --remote-ssh-key=/root/.ssh/id_rsa --cpus <num_cpus> --memory
<memory_size> --disk-size <disk_size>
```

- Deploy FfDL service

After the Minishift cluster starts, the default project `myproject` is created. This is also the namespace where the FfDL service will be deployed. We need to make some tweaks to the instructions in the FfDL project link so that FfDL can be deployed on a Minishift cluster. Following are step by step instructions.

1. Login as admin

```
oc login -u system:admin
```

2. Install `helm` as FfDL uses `helm charts` to install its services

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get > get_helm.sh
chmod 700 get_helm.sh
./get_helm.sh
```

3. Create the `tiller` service account

```
cat <<EOF > sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
EOF

oc apply -f sa.yaml
helm init --service-account tiller --upgrade
```

4. Create permissive RBAC policy to allow all service accounts to act as cluster administrators

```
oc create clusterrolebinding permissive-binding \
  --clusterrole=cluster-admin \
  --user=admin \
  --user=kubelet \
  --group=system:serviceaccounts
```

5. Install s3fs as it is used by FfDL to mount S3 buckets

```
yum install -y epel-release
yum install -y s3fs-fuse
ln -s /usr/bin/s3fs /usr/local/bin/s3fs
```

6. Clone the FfDL repository since we will run the `helm charts` from within the repository

   Install `git` if not yet installed:

   ```
   yum install -y git
   ```

   Clone FfDL github repo and checkout `helm-patch` branch:

   ```
   git clone https://github.com/IBM/FfDL.git
   cd FfDL
   git checkout helm-patch
   ```

7. Deploy FfDL service

   Before deploy, we need to set two environment variables.

   ```
   export NAMESPACE=<myproject>
   export SHARED_VOLUME_STORAGE_CLASS=""
   ```

   replace `<myproject>` with the namespace where FfDL will be installed.

   - Install object storage plugin

   ```
   helm install docs/helm-charts/ibmcloud-object-storage-plugin-0.1.tgz --name
   ibmcloud-object-storage-plugin --set namespace=$NAMESPACE,cloud=false
   ```

   - Install FfDL helper

   ```
   helm install docs/helm-charts/ffdl-helper-0.1.1.tgz --name ffdl-helper --set
   namespace=$NAMESPACE,shared_volume_storage_class=$SHARED_VOLUME_STORAGE_CLASS
   ,localstorage=true,prometheus.deploy=false --wait
   ```

   - Install FfDL core pods

   ```
   helm install docs/helm-charts/ffdl-core-0.1.1.tgz --name ffdl-core --set
   namespace=$NAMESPACE,lcm.shared_volume_storage_class=$SHARED_VOLUME_STORAGE_C
   ```

```
LASS --wait
```

8. Modify FfDL PVC to consume resource from Minishift's PV

   After FfDL is installed, it by default creates a PersistentVolume `local-volume-1` and a bound PersistentVolumeClaim `static-volume-1`. Train jobs won't be able to start with this PVC due to the `subPath` issue mentioned above. So we need to recreate the PersistentVolumeClaim to use the remounted `pv0001`.

   First delete the default PV and PVC.

   ```
   oc delete pvc static-volume-1
   oc delete pv local-volume-1
   ```

   Then recreate the `static-volume-1` PVC.

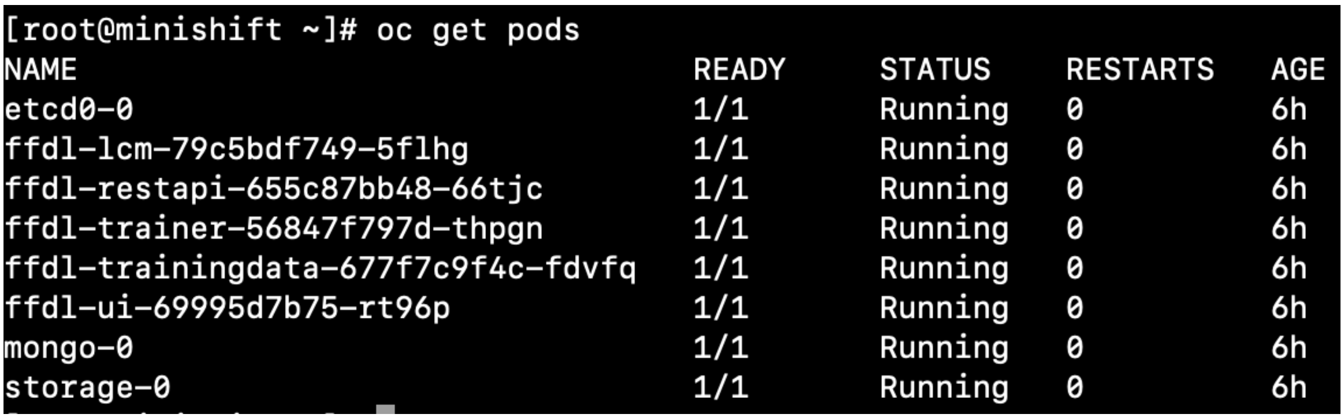
   ```
   cat <<EOF > pvc-config.yaml
   kind: PersistentVolumeClaim
   apiVersion: v1
   metadata:
       name: static-volume-1
       namespace: myproject
       annotations:
           volume.beta.kubernetes.io/storage-class:
       labels:
           type: dlaas-static-volume
   spec:
       selector:
           matchLabels:
               volume: pv0001
       accessModes:
           - ReadWriteMany
       resources:
           requests:
               storage: 20Gi

   ---
   kind: ConfigMap
   apiVersion: v1
   data:
       PVCs-v2.yaml: |
           static-volumes-v2:
               - name: static-volume-1
                 zlabel: static-volume-1
                 status: active
   metadata:
       name: static-volumes-v2
       namespace: myproject
   EOF
   ```

Note: if the namespace to deploy FfDL is different than `myproject`, replace the `namespace` field in the `pvc-config.yaml` file above with your namespace value.

Now create the PVC as follow:

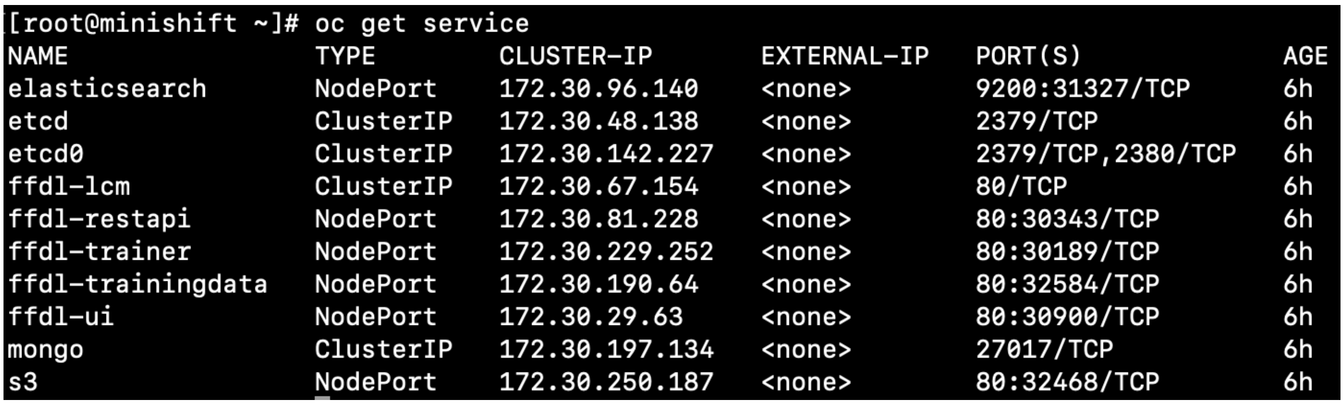
```
oc apply -f pvc-config.yaml
```

Once all steps succeed, you should see following pods and services are running:



```
[root@minishift ~]# oc get pods
NAME                                 READY     STATUS    RESTARTS   AGE
etcd0-0                              1/1       Running   0          6h
ffdl-lcm-79c5bdf749-5flhg            1/1       Running   0          6h
ffdl-restapi-655c87bb48-66tjc        1/1       Running   0          6h
ffdl-trainer-56847f797d-thpgn        1/1       Running   0          6h
ffdl-trainingdata-677f7c9f4c-fdvfq   1/1       Running   0          6h
ffdl-ui-69995d7b75-rt96p             1/1       Running   0          6h
mongo-0                              1/1       Running   0          6h
storage-0                            1/1       Running   0          6h
```

There are three infrastructure pods (`etcd0-0`, `mongo-0`, `storage-0`) and five service pods (`ffdl-ui`, `ffdl-restapi`, `ffdl-trainer`, `ffdl-trainingdata`, `ffdl-lcm`) running after FfDL is deployed. `Note names for FfDL pods are randomly generated, so you may see different names than what are in above screenshot.



```
[root@minishift ~]# oc get service
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)             AGE
elasticsearch       NodePort    172.30.96.140    <none>        9200:31327/TCP      6h
etcd                ClusterIP   172.30.48.138    <none>        2379/TCP            6h
etcd0               ClusterIP   172.30.142.227   <none>        2379/TCP,2380/TCP   6h
ffdl-lcm            ClusterIP   172.30.67.154    <none>        80/TCP              6h
ffdl-restapi        NodePort    172.30.81.228    <none>        80:30343/TCP        6h
ffdl-trainer        NodePort    172.30.229.252   <none>        80:30189/TCP        6h
ffdl-trainingdata   NodePort    172.30.190.64    <none>        80:32584/TCP        6h
ffdl-ui             NodePort    172.30.29.63     <none>        80:30900/TCP        6h
mongo               ClusterIP   172.30.197.134   <none>        27017/TCP           6h
s3                  NodePort    172.30.250.187   <none>        80:32468/TCP        6h
```

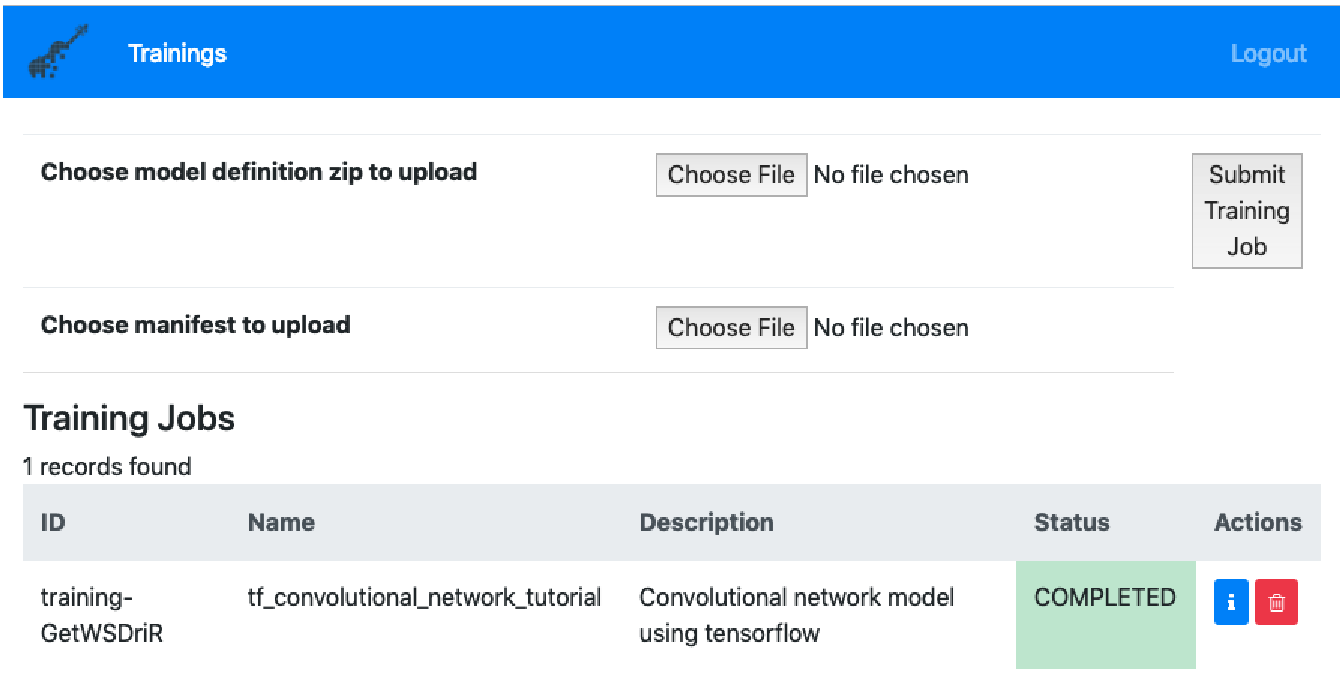
Accordingly there are five FfDL services are running, including `ffdl-ui`, `ffdl-restapi`, `ffdl-trainer`, `ffdl-trainingdata` and `ffdl-lcm`. Note the port number for each service may be different than what is in above screenshot. A local S3 object store service with `s3` name is also deployed.

Now the FfDL is fully installed and is ready to accept model training jobs.

Run following commands to get the URL link to the FfDL UI:

```
ui_port=$(oc get service ffdl-ui -o jsonpath='{.spec.ports[0].nodePort}')
restapi_port=$(oc get service ffdl-restapi -o
jsonpath='{.spec.ports[0].nodePort}')
node_ip=$(minishift ip)
echo "Web UI: http://$node_ip:$ui_port/#/login?
endpoint=$node_ip:$restapi_port&username=test-user"
```

Paste the URL to a web browser and you are ready to submit your first training job through FfDL.



Training Jobs

1 records found

ID	Name	Description	Status	Actions
training-GetWSDriR	tf_convolutional_network_tutorial	Convolutional network model using tensorflow	COMPLETED	

- Obtain access to a cloud object store

FfDL loads train data from and stores the models to a S3 cloud object store. Collect the object store credentials including `endpoint url`, `access key id` and `secret access key` from the AWS or any other S3 cloud object store you will load data from and save models to.

On the other hand, if it is just for model development and experiments, you can also use the S3 object store service deployed with FfDL above as storage. Run following command to retrieve the `endpoint-url` of this S3:

```
s3_port=$(oc get service s3 -o jsonpath='{.spec.ports[0].nodePort}')
node_ip=$(minishift ip)
s3_endpoint_url="http://$node_ip:$s3_port"
echo $s3_endpoint_url
```

the `access_key_id` and `secret_access_key` are both `test`.

To access this S3, install AWS CLI with `pip` install as follow:

```
# if python or pip is not installed, install them first
# yum install -y python python-pip
pip install awscli
```

Once AWS CLI is installed, run following command to configure the access

```
aws configure
## AWS Access Key ID [None]: test
## AWS Secret Access Key [None]: test
## Default region name [None]: us-east
## Default output format [None]:
```

Refer to `Using FfDL Local S3 Based Object Storage` section in FfDL project for more info.

- Model training with FfDL

With all preparation done above, we can now run some serious deep learning model training. Refer to the `User Guide` for how to train deep learning models on FfDL.

Generally, there are four steps to take for starting a training job:

1. upload the train data to the cloud object storage
2. develop the model training code using one of the supported deep learning frameworks, and pack the code into a model definition file in `zip` format
3. create a `manifest.yaml` file describing the deep learning framework and resource requirement
4. upload the model definition file and `manifest.yaml` file to FfDL UI and submit the job

This tutorial will end with a model training example using `TensorFlow` framework.

- Train handwriting recognition model with TensorFlow through FfDL

This deep learning model train code can be retrieved from FfDL github or here.

First thing is to upload the training data to S3 object store. When using the S3 service deployed by FfDL, follow the instructions in the link to upload the image files from MNIST datasets, as follow:

```
aws --endpoint-url=$s3_endpoint_url s3 mb s3://tf_training_data
aws --endpoint-url=$s3_endpoint_url s3 mb s3://tf_trained_model
mkdir tmp
for file in t10k-images-idx3-ubyte.gz t10k-labels-idx1-ubyte.gz train-images-idx3-
ubyte.gz train-labels-idx1-ubyte.gz;
do
  test -e tmp/$file || wget -q -O tmp/$file http://yann.lecun.com/exdb/mnist/$file
  aws --endpoint-url=$s3_endpoint_url s3 cp tmp/$file s3://tf_training_data/$file
done
```

Two buckets are created above, `tf_training_data` bucket is for storing data and `tf_trained_model` bucket is to store the training results including model itself.

Now that the training data is uploaded, continue to prepare the model train code. Users can develop model train code in any language that is supported by the deep learning framework they choose. For example, for `TensorFlow`, the model train code may be written in Python.

FfDL UI takes a model train code in a `zip` format file and a `manifest` file in `yaml` format to kick off a training job. To try out this example, run following command to compress the TensorFlow code:

```
pushd tf-model
zip -j tf-model convolutional_network.py input_data.py
popd
```

Similar command can be run to pack your own model train code.

The `manifest.yml` specifies the `cpus` and `memory` resources, the object store credentials, the deep learning framework and other info related to the model training job. Modify the example `manifest.yml` to use the FfDL S3 service by replacing the value of `auth_url` key to your cluster's `s3_endpoint_url`.

If you are using other S3 cloud object store, update the `training_data`, `training_results` and `connection` sections accordingly.

Last step is to choose the `tf-model.zip` and `manifest.yml` files from above in FfDL UI, click on the Submit Training Job to start the model training. The training job should be showed up in the `Training Jobs` list. Wait for the `Status` to change from `PENDING` to `COMPLETED` until the model training finishes. The trained model is saved in the `tf_trained_model` bucket.

- Conclusion

This tutorial provides step-by-step guidance on running a Minishift cluster locally on a VM or host, then deploying FfDL on Minishift, and finally running model training on FfDL. Specifically this tutorial shows how to work around the known issue on the current Minishift release. It should hopefully help data scientists eliminate the frustration with the deployment of a powerful fabric for deep learning on a Kubernetes containerized environment, while enjoy the better user experience on deep learning model training with FfDL.