Cloud-agnostic deployment of distributed TensorFlow

Javier Jorge Cano

EuroPython 2018

Highlights

When you leave this poster, you will know more about:

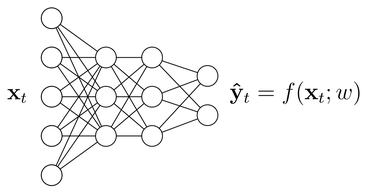
- Infrastructure as code (Infrastructure Manager).
- Automatic SW provisioning and configuration (Ansible).
- Training distributed deep learning models (TensorFlow).

Objectives

- Deploy a transient cluster of nodes to perform the training of a neural network using TensorFlow.
- Create, provision and configure the cluster in an unattended way.
- Perform these steps independently, as much as possible, from the cloud provider.

Motivation

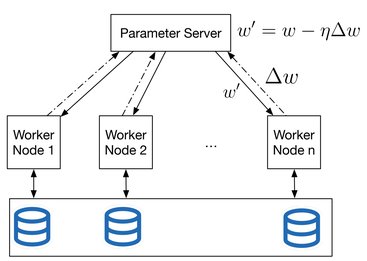
- A typical Neural Network architecture has an input $\mathbf{x}_t$, the output $\mathbf{y}_t$ as the result of the function that it represents $f(\mathbf{x}_t, w)$, with the set of parameters w.



- We compute the gradients (Δw) of this function through Backpropagation, and then we can update the current weights:

$$w' = w - \eta \Delta w \quad (1)$$

- If we want to train this model in a distributed environment, we could use the following architecture:



Where we have

- Parameter server (PS): Node that gathers the gradients and averages them, and scatters the update of the weights.
- Worker node (WN): Node that computes the forward-backward steps, obtaining the gradients to send to the PS.
- Data nodes: Some way of providing the data to the WN, i.e: text files, HDFS, S3 buckets, ...

We want to deploy, configure and run the training automatically, ending up with a persistent trained model available for inference.

Deploying the architecture: Infrastructure Manager

- The Infrastructure Manager (IM) [2] is a general platform to deploy on-demand customizable virtual computing infrastructures.
- With the IM, you can deploy complex and customized virtual infrastructures on multiple back-ends.
- It automates the Virtual Machine Image (VMI) selection, deployment, configuration, software installation, monitoring and update of virtual infrastructures.
- It provides DevOps capabilities, such as roles, based on Ansible.
- "Resource and Application Description Language" (RADL): High-level language to define virtual infrastructures and VM requirements. The general structure of these files is as follows:

```
network <network_id> (<features>)
system <system_id> (<features>)
configure <configure_id> (<Ansible recipes>)
contextualize [max_time] (
system <system_id> configure <configure_id> [step <num>]
...
)
deploy <system_id> <num> [<cloud_id>]
```

- Example of a minimal RADL file:

Defining the network:

```
network public (
        outbound = 'yes' and # Visible from outside
        outports = '2222/tcp-2222/tcp,22/tcp-22/tcp,...' and # List of ports open
        provider_id = 'vpc-vpc_id.subnet-subnet_id' # Net-ID from the cloud provider
)
```

Defining the node's parameters:

```
system node (
        net_interface.0.connection = 'public' and
        net_interface.0.dns_name = 'node0' and
        disk.0.image.url = 'aws://us-east-1/ami-5c66ea23' and
        instance_type = 't2.micro' and
        disk.0.os.name='linux' and
        disk.0.os.credentials.username='ubuntu' and
        disk.0.applications contains (
                name='ansible.modules.git+URL/ansible-role-tf|tf_node')
)
```

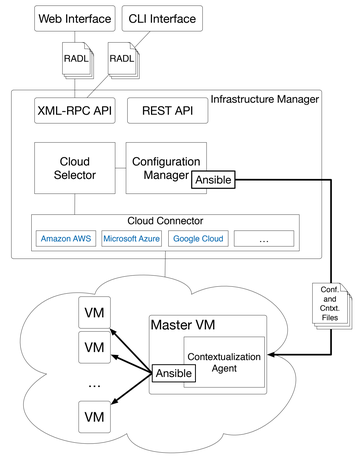
Defining how it is configured, using Ansible roles:

```
configure install_tf (
@begin
---
- vars:
        ...
  tasks:
        ...
  roles:
       - { role: 'tf_node'}
@end
)
configure run_exp (
        ...
)
```

Finally, we specify the number of nodes that we want to deploy, and if we have multiple "`configure`" rules, the order of them:

```
deploy node 2

contextualize (
        system node configure install_tf step 1
        system node configure run_exp step 2
)
```



TensorFlow's Ansible Role

The role organizes and performs the following steps to install TF:

1. Prepare system and Python dependencies, installing them with `apt` module.
2. With the module `pip`, installs `jupyter, Matplotlib, boto3` (AWS interaction) and finally TF.
3. Copy the training script parametrized with Jinja2, a templating language, to provide the parameters for the distributed training.
4. Launch the training process.

Regarding the storage, we have implemented two options:

- Using Hadoop Distributed File System (HDFS): This is configured using another role.
- Using S3 Select: Included in the training script.

Workflow

- Launch the infrastructure with IM (auth_file.dat contains the cloud-provider credentials):

```
im_client.py -a auth_file.dat create dist_tf.radl
```

- Check the status of the deployment (with the infrastructure-id that you got in the creation):

```
im_client.py -a auth_file.dat getstate <inf-id>
```

- Check the status of the contextualization:

```
im_client.py -a auth_file.dat getcontmsg <inf-id>
```

- List your infrastructures:

```
im_client.py -a auth_file.dat list
```

- Destroy the infrastructure:

```
im_client.py -a auth_file.dat destroy <inf-id>
```

- And other commands, such as `addresource`, `removeresource`, `start`, `stop`, ...

TensorFlow Training Script

- TF developers focused their efforts on making distributed TensorFlow fast and easy to use. We should just introduce a few changes in our original code [1].
- First, indicating the nodes' addresses and the tasks for each node. This is introduced in the code when the infrastructure is created thanks to Ansible and Jinja2.

```
# values replaced by Jinja2
parameter_servers =  ['ps-0:2222']
workers = ['worker-0:2222', 'worker-1:2222']
job_name = ''ps''
task_index = 0
```

- And then create the ClusterSpec with these values:

```
cluster = tf.train.ClusterSpec({"ps":parameter_servers, "worker":workers})
# start a server for a specific task
server = tf.train.Server(
        cluster,
        job_name=job_name,
        task_index=task_index)
```

- Now, the code is parametrized depending on who is the node as follows:

```
if FLAGS.job_name == "ps":
        server.join()
elif FLAGS.job_name == "worker":
        with tf.device(tf.train.replica_device_setter(
                worker_device="/job:worker/task:%d" % FLAGS.task_index,
                cluster=cluster)):
                #Model's definition from this point onwards
```

- And finally, in the workers' section, we create a supervisor to coordinate the actual training process:

```
        is_chief = FLAGS.task_index == 0
        sv = tf.train.Supervisor(
                is_chief=is_chief,
                global_step=global_step,
                init_op=init_op)

        with sv.prepare_or_wait_for_session(server.target) as sess:
                # Training loop from this point onwards
                if is_chief:
                        #Do checkpointing and logging
                ...
        #When training is over, we stop the services and the coordinator.
        sv.stop()
```

- When the training is completed, the final model is stored in a S3 Bucket by means of the package `boto3`.

Conclusions

- We have shown how to deploy, configure and launch a TF cluster automatically.
- We have used the Infrastructure Manager tool, with Ansible, to define and create the infrastructure, independently from the cloud provider.
- We have introduced how to change our TF code to a distributed TF version.

References

[1] Configuration files, roles and training scripts are publicly available on: `https://github.com/JJorgeDSIC`.

[2] Infrastructure manager Web page. `http://www.grycap.upv.es/im`.

Special thanks to the support of the EuroPython Society.