NOTES ABOUT IMPLEMENTING TECHNOLOGIES ADOPTED:

TENSORFLOW FEDERATED notes: 08/11/2021

Tensorflow Federated is a framework by Google that let researchers define Federated Learning algorithms and test they are execution in simulated environments (at this date the version available cannot be used in real end-devices such as Android one, but it is one of the goals of Google).

When coding in TFF we are practically using 3 different languages:

-TFF:

higher-level language that expresses the federated operation's logic functions ( like broadcast,aggregation) using the @tff.federated.computation tag or some @tff.method that will act as a wrapper for lower-level TF code.

This is a kind of abstract language (glue-language) because for research purposes it is intended to be platform/language indipendent because the client code may practicallt be deployed over mobile nodes or some other distributed system.

In TFF we don't have the concept of state.

We can think of a fed.computation as a distributed computation that doesn't only take place at the server or in a particular place at all but needs a lot of actors to be executed .

So we can in a sense map the "state" in to those input and outputs , in fact we usually have to specify both the itemtype and location in a {T}@G fashion were T it's the itemtype ( e.g Float32) and G is the item's location in the system ( e.x @Client , @Server ).

-TF:

this is used as classical TensorFlow code , encapsulated in TFF functions to implement to implement the core functionalities such as (crete the model , train locally... ) .

This is expressed through the tag @tf.function and @tf.computations annotations that expects pure TF code internal to them .

-Python:

It is used to simulate (start) the algorithm, iterate for some training round or similar.

Behind the curtains Pyhton will call TFF frawemork ton execute the Fed Training.

KUBERNETES NOTES 08/11/21

A kubernetes cluster is formed usually by a lot of nodes that in general can oly be of two tipes:

- Master node

- Worker node (slave)

Each worker node has to support :

- a container runtime ( e.g Docker )

- a kubelet process : a process that it is used to interact voth with the container and the node;

Kubelets starts the pods ( Abstraction of a container) within a container inside the worker node and later it assigns resources from the node to the containers ( cpus, Rams , Volumes ).

More worker nodes can ( and usually does) present replicas of the same pods that can btw them using the "Service " pods ( Load Balancer that catches the application requests and forward them).

In order to implement efficiently this kind of features on the master node 4 prcesses needs to run:

- Api Server : element used to interact with some client ( UI , API or CLI as kubectl).

- Scheduler : it is the second actor in the "request workflow" and has the ability to schedule some process over a worker node. It has the intelligent ability of deciding which one is the best pod to host the process so it acts in a sense also as a Load Balancer ( it schedules based on the resource occupation over a node ).

-Controller manager : it detects state changes like crashing of pods or similar over the cluster, and tries to resume the system's state as soon as possible ( ex. by contacting the scheduler to restart the crashed pods ).

As before once the scheduler has select the best node to which it will assign the process it will contact its kubelet that it is in charge of restarting the selected pod.

-ETCD: It is a key-value storage of the cluster state. We can in a sense imagine it as the cluster brain.

All the changes that takes place in the system will be stored here as a key-value pair ( for ex when a new service is deployed, when a Pod crashes..).

The whole mechanism at the master node it's based upon the data stored here in the ETCD.

This key-value storage anyway does not store application-specific data too as databases or similar.

Given this important processes that needs to run on the Master node to assure a good overrall performance of the cluster in many K8s real distributions the cluster is composed not by a single master -multy slave , but both the master and the slave nodes are replicated for the sake of the availability and scalability of the cluster.

In this kind of deployment the API servers are load balanced in ingress requests ( if we have more than 1 Api-server the total request towards the cluster will be load balanced btw the copies of the API server) and the ETCD forms a distributed storage across the master nodes.

An important thing to point out is that usually th master nodes has less resources than the worker one because in the worker it resides the application logic that is usually more resource demanding.

An useful tool to cope with a kubernetes cluster is the Kubectl CLI , a Command Line Tool for k8.

Practically when we interact with the clusters for example to start a process , we don't manage directly the pods ( that are k8s smallest units ) but we will create deployments ( abstractions over pods )

EX:

kubectl create deployment DEP\_NAME --image=IMAGE

this is the basic (without any optional paramenter) command to create a deployment from an existing image ( either in the dockerhub or some repository or local)

A deployment it's the blueprint for creating Pods ( similar to what are the docker files for the Containers).

In the middle btw deployments and pods there is an additional layer called "Replica Set".

This layer is the one in charge of managing the replicas of a pod over the cluster, but in practical terms we will NEVER deal directly with them.

kubectl logs PODNAME

This command it's useful for debugging pods.

kubectl exec -it PODNAME --bin/bash ( or --bin/sh if bash is not available in the container)

This commands gets a terminal of the PODNAME container , -it stands for "Interactive Terminal" .

If we want to express different and numerous features explicit them all at once in a command would be tricky so to configure Kubernetes element we exploit the so called " configuration files" usually YAML files and use them with the command :

kubectl apply -f FILE\_NAME

A configfile has a fixed structure with its elements that changes depending on the "kind" of component we are configuring ( Service , ConfigMap , Deployment ... )

The stucture is the following :

api Version:

kind: ( Service , ConfigMap , Deployment )

metadata : this field contains infos like name and labels used to identify the component from others

spec: this field contains kind-specific attributes and additional infos such as replicas . Internal to this component ( in the case of kind=Deployment) we find the template field that represents the blueprint of the pod and it is in turn formed by metadata and spec fields.

Another part of the configuration file not specified at deployment time is the status and represents the status of the pod while it is running.

All the orchestration and similar mechanisms in kubernetes are based on the equation:

-Desired State (the one specified in the YAML) = Actual state?

If so, no action needs to be taken otherwise as soon as the system notices that this equation, is not satisfied (by looking up the data at ETCD) it will take some action to reestablish this balance.

Another important concept in k8 is the concept of " Namespaces" since all the resources of the cluster are organized within them.

In short, a namespace represents a virtual cluster inside of a real one.

By default, we have available already 4 different namespaces:

- Kube-system: it is a namespace to not modify that contains system processes such as master and kubelet processes.

- Kube-public : it's the namespace which contains publicly accessible data that is stored in a configMap for cluster information.

- Kube-node-lease: it's the namespace that contains healt information about each node,

To do so each node has associated a lease object in this namespace that determines the availability of a node.

- default : it's the namespace used to locate the resources we create.

Namespaces can also by created and used in different scenarios, for ex if two teams (test and deployment) are working over the same k8cluster at two different versions of an application.

It would be useful to define two different namespace and let each team use their own namespace.

KUBEFLOW NOTES: 08/11/21

KubeFLow is an open-source platform to easily manage and deploy ML (Machine Learning) end to end procedures and run them without the need of explicitly create each underline fundamental component.

An important abstaction available in this platform is the one of the "Pipelines" that let the user define a custom pipeline composed by basic block operations ( fct. with some input and output) such as Data preprocessing, model creation, train-test phase ...

The main idea that drives this technology is that once we have a KubeFLow ML procedure we can deploy it over different architectures (e.g phisical devices,edges,cloud ) that hosts a k8 cluster without adding additional low level infos regarding the target platform.

At the bottom of the KubeFlow architecture there are the available Platforms who host the kubernetes Cluser to which we can deploy a kubeflow process (GCP, AWS, Azure, On prem platforms).

On top of that there are a lot of KubeFlow applications and scaffolding ( e.g Katib, KubeFlow UI, KFServing , Pipelines ...) that represent a common set of tools available independly of the type of ML procedure we are going to implement.

At the upper layer based upon those kubeflow applications we find all the most adopted tools/platform for ML such as Jupyter notebooks (we have the possibility to define Jupyter Notebook Servers), PyThorch ,Tensorflow and other popular frameworks.

This kind of platform is very important,especially from now on that AI will be used more and more,because represents for the ML community a standard and complete tool for deploying ML enterprise Apps and manage they'r entire lifecycle.

So to repeat briefly the main aim of the platform is to easily let users develop, deploy and manage ML Apps all in one place on top of a Kubernetes container orchestration platform.

This solution presents 3 important properties:

-COMPOSABILITY:

it means that we can choose what is right for our project from a wide range of common building blocks ( Jupyter, Pythorc , TF). For example, we may use different versions of the TF plaform in distinct parts of the project (or of the Pipeline).

-PORTABILITY:

it means that we can run every piece of the project (pipeline/flow) anywhere we are running KubeFlow.

-SCALABILITY:

it means that we can access more resources if they are needed and release them when they are not. Every environment may also have different computing resources (CPUs, GPUs, TPUs).

To scale it exploits the ability of the underlying kubernetes orchestrator.

Istio is one of the core components of the Kubeflow architecture.

In the whole architecture, it resides on top of the k8 cluster too and it is used by the upper KubeFlow layer.

Istio simplifies the traffic management of our service-API, by using Envoy proxies that are deployed along with our services.

In this way all the incoming and outcoming data plane traffic is proxied through Envoy so that we don't need to change anything in the services itself to change the traffic control logic.

To do so Istio exploits an internal Service registry, connects itself to a service discovery system like the one that already exists in K8 clusters.

By default, the Envoy proxies distribute traffic across each service’s load balancing pool using a round-robin model, where requests are sent to each pool member in turn, returning to the top of the pool once each service instance has received a request.

WIth Istio we can also realize A/B testing policies or a particular traffic policy by selecting the percentage of the messages that needs to be routed to a new/outdated version of the application.

An essential element for Istio is the so-called Virtual services.

Virtual Services let you configure how requests are routed to a certain service within an Istio service mesh,each of them consists of a set of routing rules that are evaluated in order, letting Istio match each given request to the virtual service to a specific real destination within the mesh. Your mesh can require multiple virtual services or none depending on your use case.

On top of Istio stands another key component for the overall behavior of the kubeflow platform: Knative.

## Knative is Kubernetes-based platform to deploy and manage modern serverless workloads.

Its main components are the “ Knative serving” and “Knative eventing”.

With Knative serving we can define services (as the classical K8 one) using mainfest or the Kn CLI that are like classical services with some additional functionality.

For example, knative services has built in automatic auto-scaling, following the serverless paradigm so “scale to zero” enabled, will scale the service dependently of the incoming requests down to zero if no request arrives.

Another important feature that comes with knative services is the traffic splitting, in which we can select a specific percentage of traffic to direct to, for example, an old or an updated version of the same service to test it.

For what regards Knative eventing instead it is important to point out that it lets define event-driven applications/components (all concepts reused and exploited to realize kubeflow pipelines).

Once the kubeflow platform is installed over the K8s cluster and hany tool (specifically a CLI) to interact with it and deploy kubeflow applications is the kfctl , in a sense the counterpart of kubectl but intended for kubeflow deployment.

Through kfctl in fact we can deploy similarly to kubectl components from their manifest (YAML) files using the “kfctl build” and “kfctl aplly –f CONFIGFILENAME”.

One other available and useful tool offered by kubeflow is Katib. The goal of this tool is to ease the challenging task to set the optimum hyper-parameters of some ML algorithm.

*Hyperparameter tuning* is the process of optimizing the hyperparameter values to maximize the predictive accuracy of the model. If you do not use Katib or a similar system for hyperparameter tuning, you need to run many training jobs yourself, manually adjusting the hyperparameters to find the optimal values.

In fact, Katib let us visualize in a global and intuitive manner all the possible couples, sets of values for the hyperparameters to tune to select the best one without the need of train a lot of times and tune them a global optimum way.

In addition to that it also presents functionalities to infer the best architecture (optimal number of layers, optimal number of nodes per layer) for a given neural network.

Katib runs several training jobs (known as *trials*) within each hyperparameter tuning job (*experiment*). Each trial tests a different set of hyperparameter configurations. At the end of the experiment, Katib outputs the optimized values for the hyperparameters.

OTHER NOTES ABOUT KUBEFLOW: PIPELINES (10/01/2022)

Important note: some of the components such as “Kale” and “Rok” are available in the MiniKf distribution only, may be integrated soon to the open source ones.

All the general( and related to other tools such as “Katib” and “Kfserving” ) considerations remains valid though.

The ML workflow using Kubeflow has many operative advantages between which we have:

* Simplify the Hyperparameters tuning and Serving workflows using an intuitive UI
* Accelerate the time to production by reducing the needed time for training and from training to serving
* collaborate faster within the team especially between Data scientists and the MLOps team.

Kubeflow offers some tool to ease the process of mapping complex ML tasks into its fundamentals and simple operations in order to create a Kubeflow Pipelines that meet up the objectives of the project or the business in case.

In particular among the plethora of ML frameworks and technologies widely used to create and test ML algorithms we have the Jupyter Notebooks.

It is possible to decompose an algorithm developed locally using Jupyter Notebooks using “Kale” by annotating notebook cells and convert the notebook into a scalable pipeline.

Afterwards we can spin up a Hyper parameter tuning Job using “Katib”to run possibly hundreds or thousands of parallel execution of the scalable pipeline obtained in the previous step.

One another important feauture of “Kale” is that after this parallel training to search the best HP it help us to select the best models on massive workloads and then with a very simple API we are able using “KFServing” to serve these models to anyone who need to use it ( maybe another section of the ML team).

Each step of the described process is tracked by an MLMD (or ML Metadata) so we can have an end to end succession( lineage ) of the workflow.

In addition each step is backed up by PVCs (Persistent Volume Claims) and another tool called “Rok” can take a snapshot of these PVCs to also have a complete time machine of the workflows.

The Kubeflow pipelines were born based upon the idea that Data Science and ML are inherently pipeline processes given the number of steps usually involved :

* Data ingestion
* Data Analysis
* Data Transformation
* Data validation
* Data splitting
* Trainer
* Building a model
* Validate the model
* Train the model at scale
* Serving it
* Monitor and Log the processes

Convert a notebook to a Pipeline has several benefits :

* notebooks allow us to define what is the structure of the resulting pipeline
* since we have multiple cells we can easily parallelize and isolate them
* once we have a pipeline we can apply data versioning and different hardware requirements for running them ( for ex : run training steps in GPUs and data processing one in CPUs).

If we don’t make this notebook to pipeline convertion it is possible to create custom pipelines by using Kubeflow Pipelines SDK but the whole process is more complicated and it usually consist in:

* Write your own ML code and test it locally
* Containerize it by creating a docker image to be used by the underlying kuberntes infrastructure
* Write DomainSpecificLanguage in KFP( Kubeflow Pipeline SDK)
* Compile DSL KFP code
* Upload the pipeline to KFP
* Run the pipeline

The downside of the whole process is that if some kind of bug appears while the pipeline is running since we had containerized manually the ML code we have always the need to rebuild and re-containerize the new version of the algorithm resulting in a longer process.

Whit “Kale “ and “Rok” ( the two components mentioned before) the whole process is reduced to the following steps:

* Write your own ML code and test it locally
* Tag your notebook cells
* Run the pipeline through the UI by clicking a button

The second way of creating custom pipelines it’s obviously faster and more business-driven, in the sense that we can focus on working over the ML code and leave out all deployment procedures such as containerize the code and make it compatible as a Pipeline through KFP.

It is very useful because dramatically reduce the speed to production and iteration.

The concept of pipeline inherits two more basical but still important concenpts: input and outptus!.

When defining pipelines it is possible to link input parameters to pipelines (for example constants used for HP tuning , artifacts containg datasets, URI from which we can read the content of a file..)

and define specific output parameters (such as digested data to be used by the next pipeline steps, a path to which save the intermediary results, and more important output metrics used to evaluated the evolution of the model while training or testing it )

Kale ease a lot the definition of this input and output parameters because you just need to create a notebook cell with some variable assignments and annotate it with the pipelines parameters annotation through the UI, this translates in a parametrization of these tagged valued by kale obtained the desired pipeline schema.

In the same way through kale we may tag the variables ( with the metric annotation) in our notebook that holds metrics info and we want to print during the ML process.

The “Kfserving” component helps us in deliver to production the best model after a parallel run to tune our HP, it is based on Knative and it allows serverless inferencing in kubernetes.

It provides several abstractions on top of different ML frameworks and provides interesting features like “Canary deployment” ,“Scale to Zero” and more.

The process of delivering to process the best trial consist on restore a notebook from the Rok snapshot related to the trial