Serving Tensorflow Models with Kubernetes

Ron Bodkin, Technical Director, Applied AI
Brian Foo, Senior Software Engineer, Applied AI

Step One: Get An Account
Go To goo.gl/sgZ2Qp to start
Your Instructors

Ron Bodkin
Brian Foo
Cassie Kozyrkov

Set up an account here: goo.gl/sgZ2Qp
Your Instructors

Ron Bodkin
Brian Foo
Cassie Kozyrkov

Set up an account here: goo.gl/sgZ2Qp
Overview

● Tensorflow
  ○ Building and Running Tensorflow graphs & APIs
  ○ Tensorflow Serving for Online Predictions

● Docker and Kubernetes
  ○ A brief history of containers
  ○ Kubernetes: pods and services

● The Marriage of TF Serving and Kubernetes
Building and Running Tensorflow Graphs
Build a Tensorflow Graph

```python
import tensorflow as tf
x = tf.placeholder(tf.float32, shape=(100))
# Some preloaded model weights and biases
w = tf.get_variable('weights', shape=(100))
b = tf.get_variable('bias', shape=[])  
y = tf.tensordot(w, x, 1) + b
```
Run the Tensorflow Graph

```python
import tensorflow as tf

x = tf.placeholder(tf.float32, shape=(100))

w = tf.get_variable('weights', shape=(100))
b = tf.get_variable('bias', shape=[])  
y = tf.tensordot(w, x, 1) + b

with tf.Session() as sess:
    print(sess.run(y, feed_dict={x: input_data}))
```

Data entrypoint for predictions
Different APIs for TF

- Estimator
- Keras model
- Layers
- Python front end
- C++ front end
- TensorFlow Distributed Execution Engine
- CPU
- GPU
- Android
- iOS
- ...
What is Serving?

Serving is how you apply a ML model, after you’ve trained it
What is Serving?

Data → TensorFlow → Model → ? → App

Data Scientist
What is Serving?

Data → TensorFlow → Model → RPC Server → App

Data Scientist
TensorFlow Serving Goals

- Online, low latency
- Multiple models in a single process
- Multiple versions of a model loaded over time
- Compute cost varies in real-time to meet product demand
  - auto-scale with CloudML, Docker & K8s
- Aim for the efficiency of mini-batching at training time ...
  - except with requests arriving asynchronously
TFX: TensorFlow-Based ML deployment at Google

Integrated Frontend for Job Management, Monitoring, Debugging, Data/Model/Evaluation Visualization

Shared Configuration Framework and Job Orchestration

Tuner

Data Ingestion
- Data Analysis
- Data Transformation
- Data Analysis
- Trainer
- Model Evaluation and Validation
- Serving
- Logging

Shared Utilities for Garbage Collection, Data Access Controls

Pipeline Storage
Docker and Kubernetes
A Brief History of Containers

2008

In 2008, Linux introduced containers.

- Isolated environment for running applications, except...
- All applications must have a common OS Kernel (e.g. Ubuntu, Debian, etc.)
In 2013, Docker found a way around the shared kernel problem.

- Application containers have their own OS kernel.
- Flexible resource requirements.
- Perfect for cloud computing!
In 2014, Google open sourced Kubernetes.

- Deploy Docker containers to any number of machines
- Create load balancing and front-end services to handle external requests.
- Automatically restart backend containers when they fail.
Example projects built on top of Kubernetes:

- Monitoring (Heapster)
- Deployment languages (KSonnet)
- Deployment automation (Kubeflow for ML)
- and many more!
Kubernetes in a Nutshell

Kubernetes Pods
collections of containers that are co-scheduled

Pod 1
Pod 2
node
docker
docker
Kubernetes in a Nutshell

How Kubernetes Works?

Manual or auto-scaling!
Tensorflow Serving using Kubernetes
TF Serving on Kubernetes Workflow

What do we want?

- A prediction service that can handle multiple client requests
- Load-balancing across TF model servers
- Ability to scale up
TF Serving on Kubernetes Workflow

Exercises

How do we get there?

- Convert TF training code to model for serving.
- Package model in Docker container and upload to a registry.
- Use *Kubernetes* to:
  - Deploy container on multiple back-end pods.
  - Deploy a front-end service to send client requests to a backend pod.
- Send protobufs (encoded JSONs) containing images to Kubernetes cluster.
- Load testing.
Monitoring, Interpretation, and Keras
Bonus Exercises

Heapster Grafana Dashboard
(Pod and Cluster Resource Usage)

Model Understanding and Visualization using Integrated Gradients
Codelab Time!

Open a Chrome incognito window.

Log in at events.qwiklabs.com

If you don’t have an account register at goo.gl/sgZ2Qp
Recap and Demos
Acknowledgements and Additional Resources

Special thanks to:

- **Kubeflow**: providing Docker images and templates for TF Serving on Kubernetes
  David Aronchick, Jeremy Lewi, Vishnu Kannan (Google); Peng Yu (Shopify)

- **Google Cloud ML**: GPU batch profiling work using Beam and Tensorflow

Reference - Model Visualization:

Thank you

1. Please leave feedback
2. Resources at goo.gl/Sg6ecA
3. Save any work you want to keep
Appendix: pipelines versus Client-Server Architectures
Pipeline Architecture: Batch/Online Processing

- Read offline data from local/HDFS/Google Storage/AWS
- Preprocess (clean, filter, aggregate) using Spark/Beam/Flink
- Create batches to run through a TF graph
- Update model params (training) / Collect inference results (serving)
Pipeline Architecture: Batch/Online Processing

**Benefits:**
- Full control over pipeline application and model!

**Limitations:**
- **Language Dependency:** Requires Python, or Java JNI to C++
- **No Proprietary Models:** Requires graph and model params to be exposed in code.
- **Experience:** Months to years of expertise to build, debug, and manage pipelines effectively.
Client-Server Architecture: Tensorflow Serving

- Asynchronous and Streaming Model Serving
- Efficient implementation in c++
- Server build can be optimized for native environment
  - CPUs or GPUs
  - Just-in-time (JIT) compilation
  - etc.
Client-Server Architecture: Tensorflow Serving

- Asynchronous and Streaming Model Serving
- Efficient implementation in c++
- Build can be optimized for the environment (CPUs or GPUs)
- **Language independent Protobufs!**
  - RESTful API calls using serialized dictionaries
  - Send dictionary of data
  - Receive dictionary of prediction results
Client-Server Architecture: Tensorflow Serving

- Asynchronous and Streaming Model Serving
- Efficient implementation in c++
- Build can be optimized for the environment (CPUs or GPUs)
- **Language independent Protobufs!**
  - RESTful API calls using serialized dictionaries
  - Send dictionary of data
  - Receive dictionary of prediction results
- How do we guarantee identical serving environments?
- How do we scale?
- How do we handle failures gracefully?