caicloud
才云

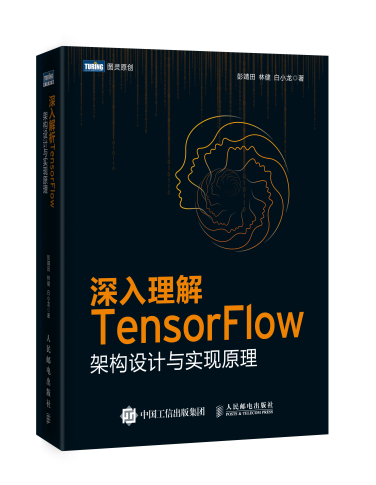
基于 Kubeflow 的云原生机器学习最佳实践

Caicloud 彭靖田

个人介绍

彭靖田

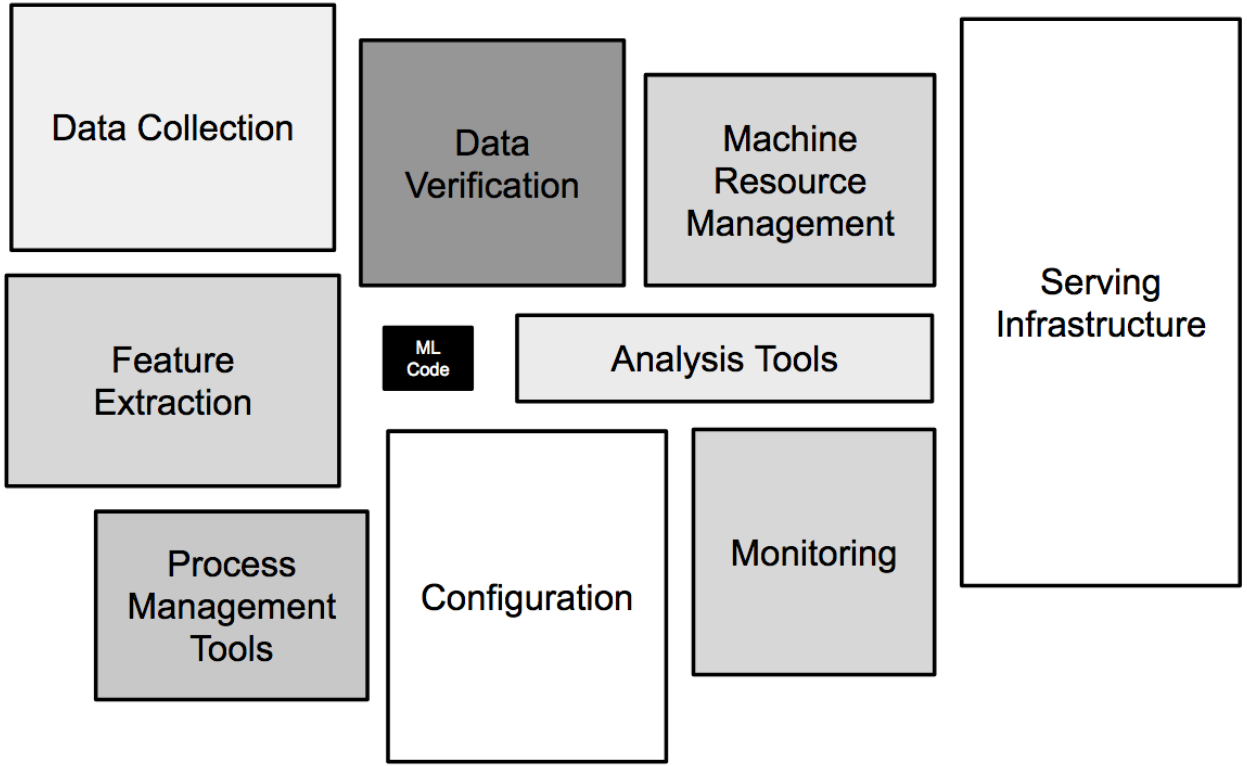
- Caicloud Cloud AI - Clever 负责人
- Kubeflow Maintainer
- Google Developer Expert
- TensorFlow Contributor
- 《深入解析 TensorFlow 》作者
- UCSD Visiting Scholoar



目录

目录

ML is more than a framework



A User's Perspective

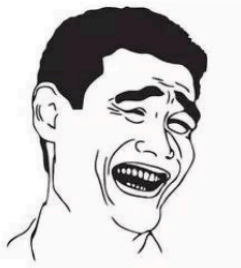
- Write ML code
- Run it
- Get the result

A System Engineer's Perspective

- Machine Learning Framework (TensorFlow, Caffe 1 & 2, PyTorch)
- Hyperparameter Tuning Systems
- Hardware Accelerator (GPU, TPU, FPGA)
- Networking (InfiniBand, NUMA)
- Virtualiazaion (Xen, KVM, Docker)
- Cluster Management System (Mesos, Kubernetes, Swarm, Nomad)
- Continuous Integration and Deployment
- Logging Collection

Why do we need Kubeflow?

- Machine Learning Framework (TensorFlow, Caffe 1 & 2, PyTorch)
- Hyperparameter Tuning Systems
- Hardware Accelerator (GPU, TPU, FPGA)
- Networking (InfiniBand, NUMA)
- Virtualiazaion (Xen, KVM, Docker)
- Cluster Management System (Mesos, Kubernetes, Swarm, Nomad)
- Continuous Integration and Deployment
- Logging Collection



Kubeflow does it for you

Kubeflow

Run  

 dmlc mxnet on 

- The Kubeflow project is dedicated to making **deployments** of machine learning (ML) workflows on Kubernetes simple, portable and scalable.
- Our goal is **not to recreate** other services, but to provide a straightforward way to deploy best-of-breed **open-source systems for ML** to diverse infrastructures.
- Anywhere you are running Kubernetes, you should be able to run Kubeflow.

Kubeflow - Community

::::::::::::

Google	intel
caicloud 才云	redhat
seldon	Microsoft

More Contributors

tf-operator	mxnet-operator	Caffe-operator
Kubeflow/Kubeflow	community	testing
examples		reporting

More Repos

目录

TensorFlow Introduction - Overview

2011 Google DistBelief

- Google
- Youtube
- ...

Large-Scale Deep Learning for Intelligent Computer Systems

Jeff Dean

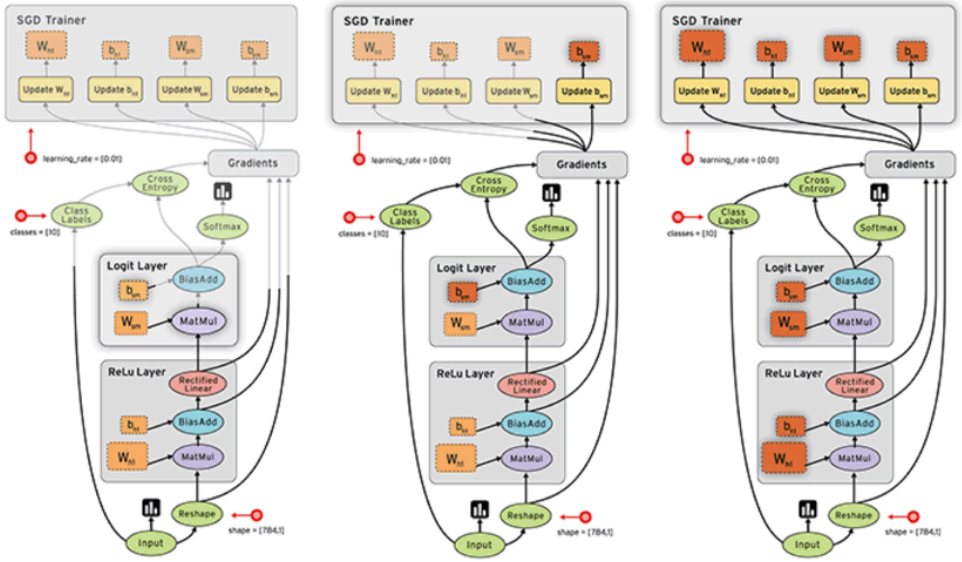
Google Brain team in collaboration with many other teams

2015 Google TensorFlow (Open Source)

- Airbnb
- Google
- Intel
- JD
- Mi
- Uber
- ...

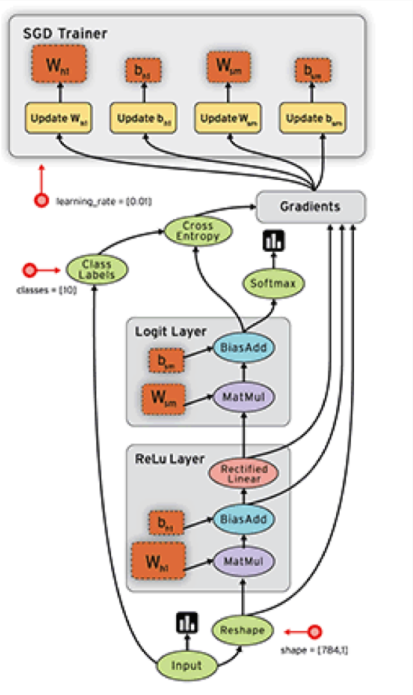
TensorFlow

TensorFlow Introduction – Features



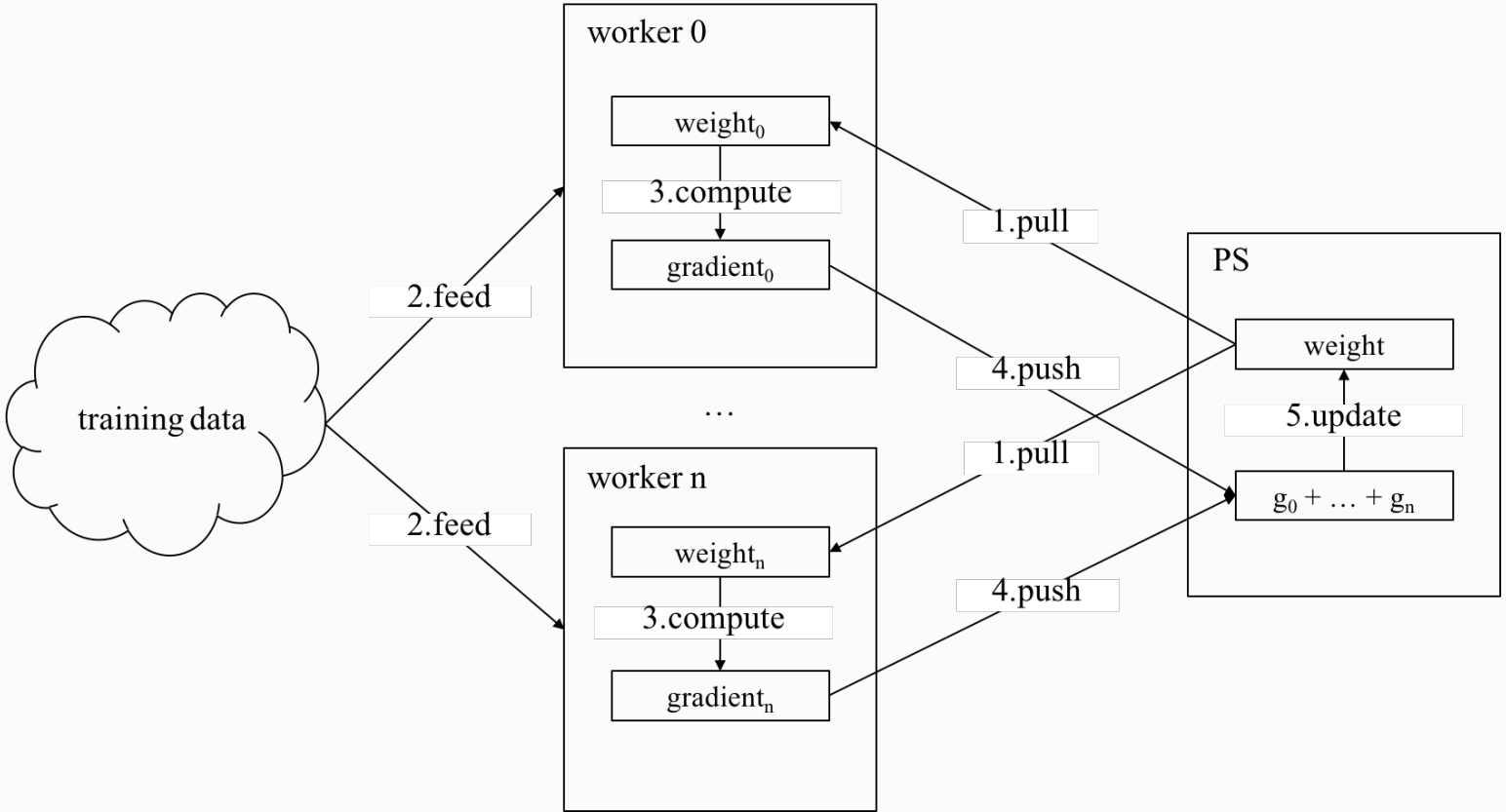
- 运算性能强劲
- 端云计算协同
- 超大集群并行
- 模型设计灵活
- 生产环境部署
- 语言接口丰富
- 教学资源充足

TensorFlow Introduction – Dataflow Graph

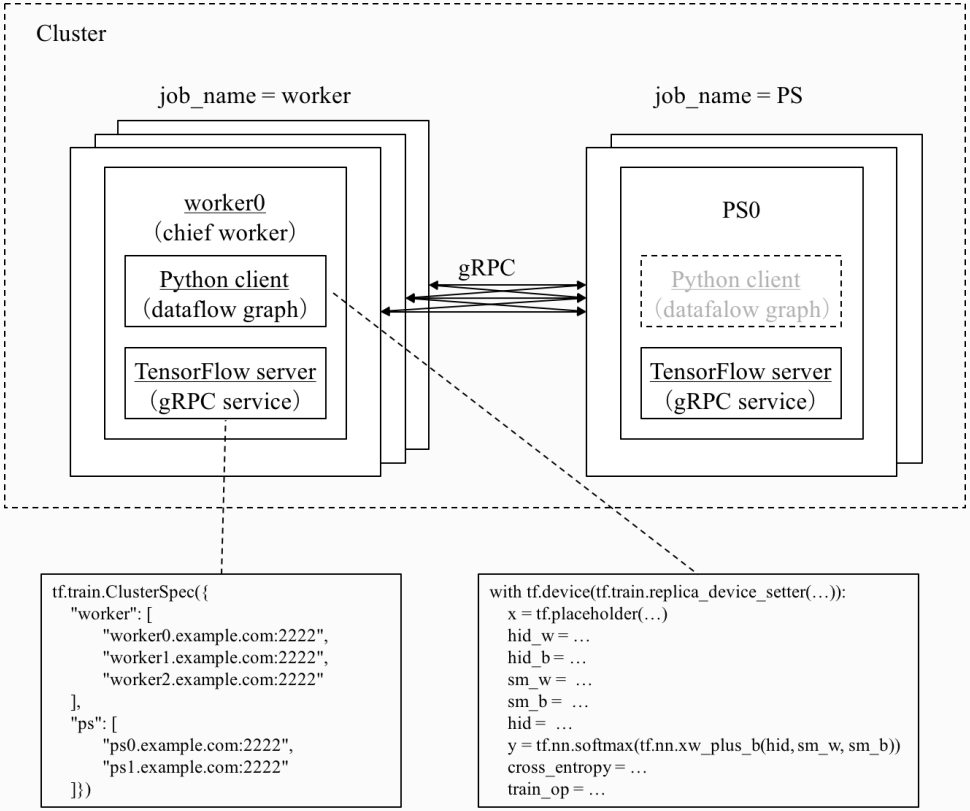


- 数据流图
  - 有向边
  - 节点
    - 张量 — Tensor
    - 稀疏张量 — SparseTensor
    - 计算节点 — Operation
    - 存储节点 — Variable
    - 数据节点 — Placeholder

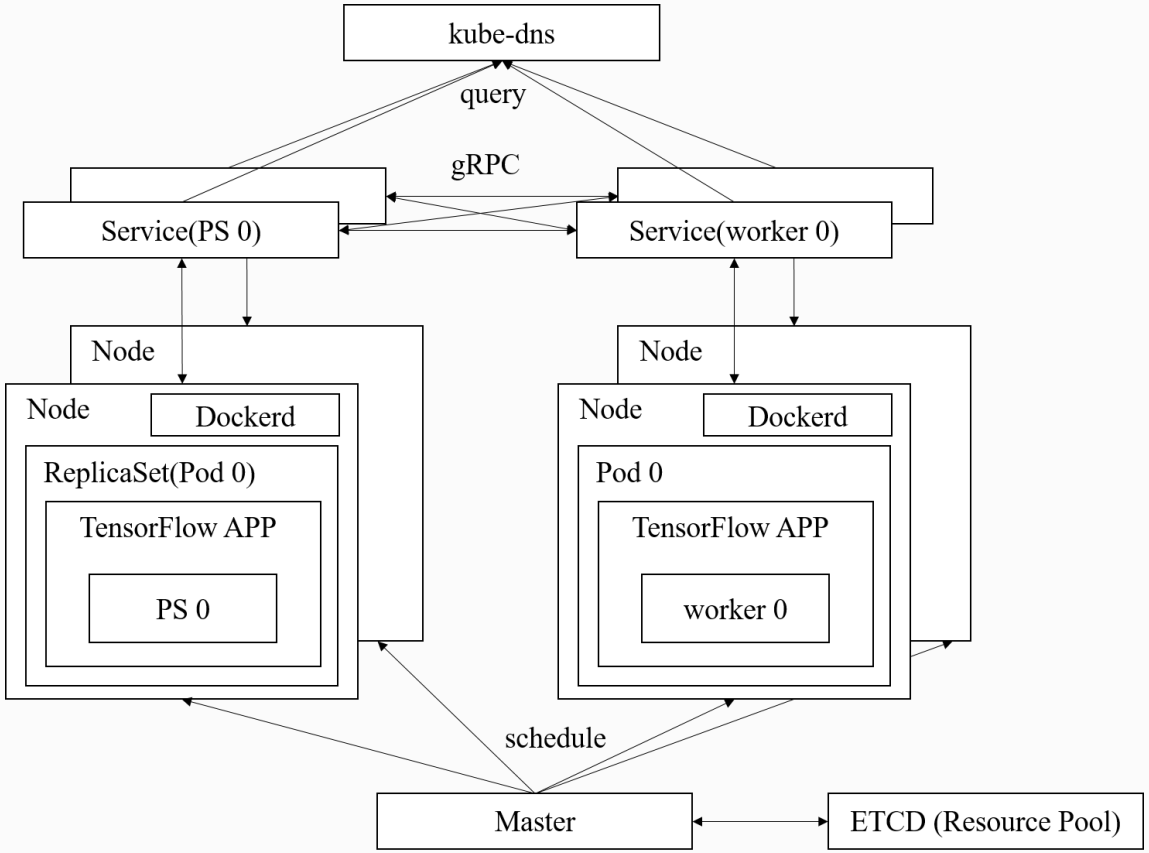
Distributed TensorFlow: PS-worker Framework



Distributed TensorFlow: PS-worker Framework

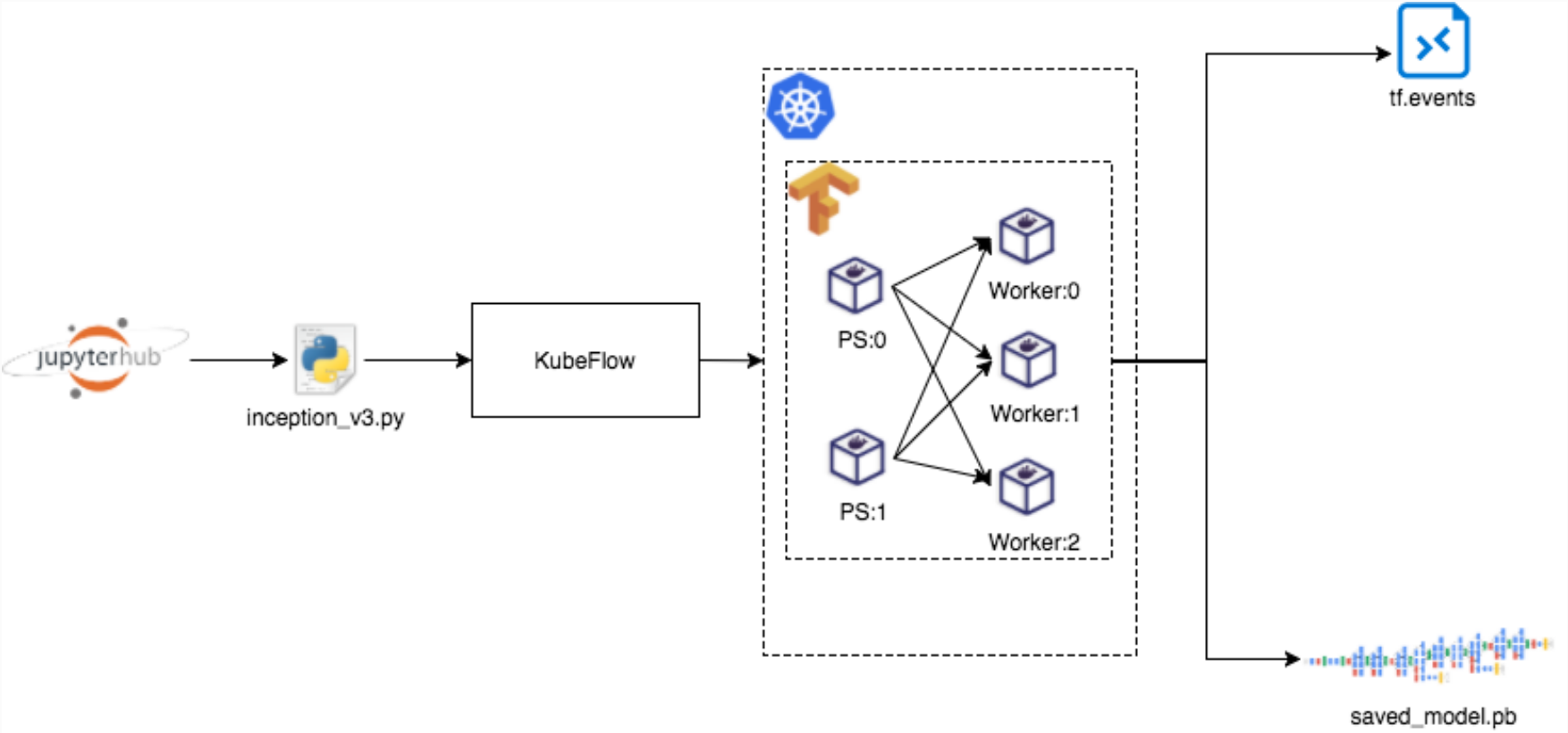


Traditional Distributed TensorFlow on Kubernetes



tf-operator

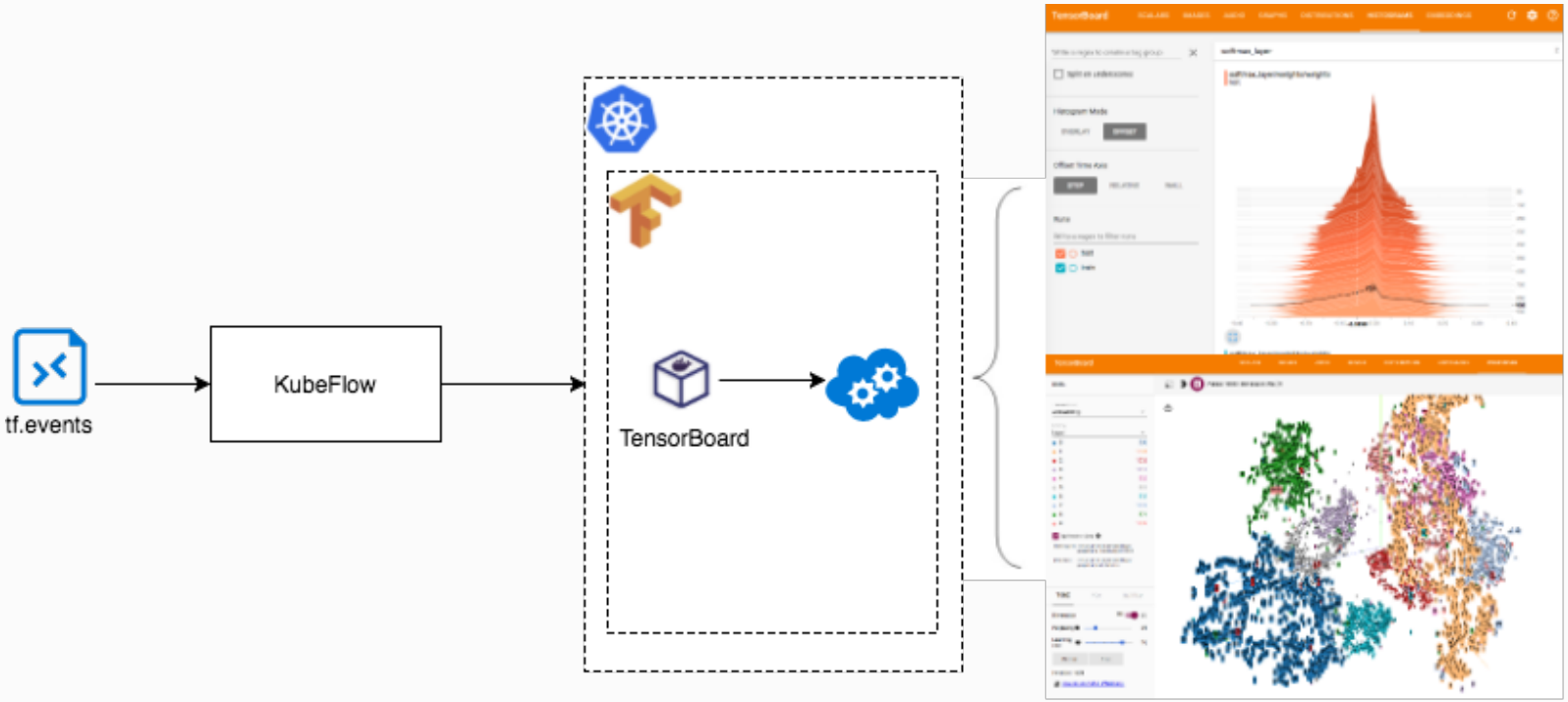
TensorFlow Pipeline with Kubeflow



Deploy Distributed TensorFlow Job with kubeflow/tf-operator

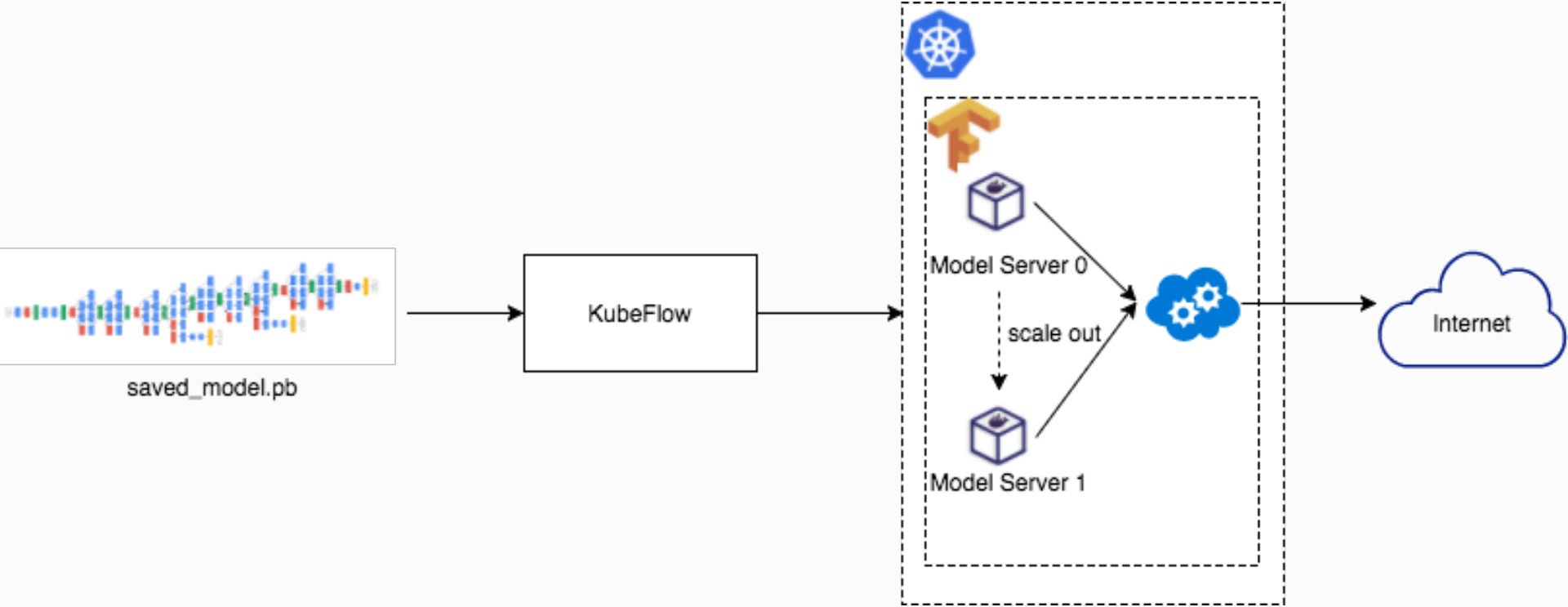
```yaml
apiVersion: "kubeflow.org/v1alpha1"
kind: "TFJob"
metadata:
  name: "example-job"
spec:
  replicaSpecs:
    - replicas: 1
      tfReplicaType: MASTER
      template:
        spec:
          containers:
            - image: tf_sample:dc944ff
              name: tensorflow
          restartPolicy: OnFailure
    - replicas: 1
      tfReplicaType: WORKER
      template:
        spec:
          containers:
            - image: tf_sample:dc944ff
              name: tensorflow
          restartPolicy: OnFailure
    - replicas: 2
      tfReplicaType: PS
      template:
        spec:
          containers:
            - image: tf_sample:dc944ff
              name: tensorflow
  restartPolicy: OnFailure
```

Model Visualization



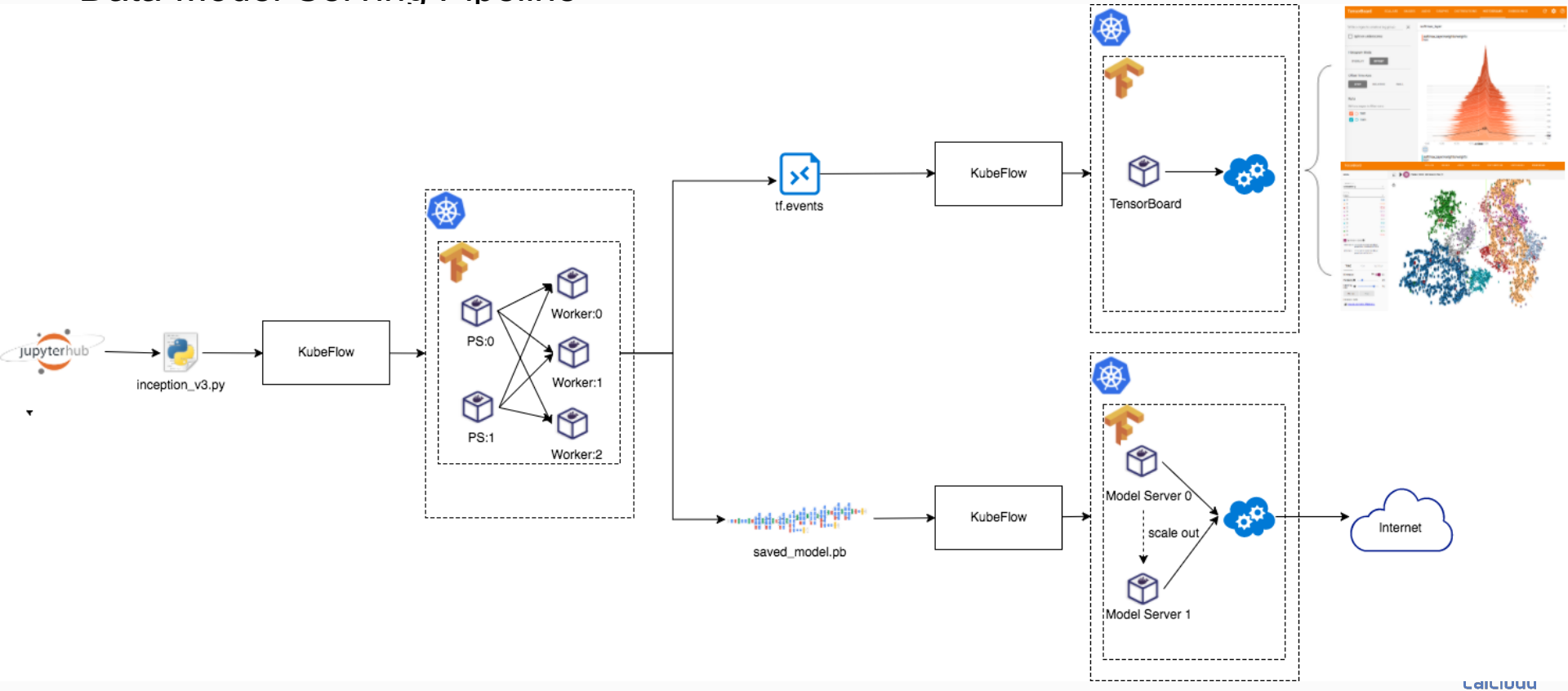
tf-operator

Model Serving



tf-operator

Data-Model-Serving Pipeline



目录

PyTorch Introduction

Tensors and Dynamic neural networks in Python with strong GPU acceleration.

PyTorch is a deep learning framework that puts Python first.

We are in an early-release Beta. Expect some adventures.

Torch vs PyTorch

Torch
(-) Lua
(-) No autograd
(+) More stable
(+) Lots of existing code
(0) Fast

PyTorch
(+) Python
(+) Autograd
(-) Newer, still changing
(-) Less existing code
(0) Fast

Torch vs PyTorch

Torch
(-) Lua
(-) No autograd
(+) More stable
(+) Lots of existing code
(0) Fast

PyTorch
(+) Python
(+) Autograd
(-) Newer, still changing
(-) Less existing code
(0) Fast

Conclusion: Probably use PyTorch for new projects

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray, but runs on GPU

Variable: Node in a computational graph; stores data and gradient

Module: A neural network layer; may store state or learnable weights

PyTorch: Three Levels of Abstraction

	TensorFlow equivalent
Tensor: Imperative ndarray, but runs on GPU	Numpy array
Variable: Node in a computational graph; stores data and gradient	Tensor, Variable, Placeholder
Module: A neural network layer; may store state or learnable weights	tf.layers, or TFSlim, or TFLearn, or Sonnet, or ….

PyTorch: Tensors

PyTorch Tensors are just like numpy arrays, but they can run on GPU.

No built-in notion of computational graph, or gradients, or deep learning.

Here we fit a two-layer net using PyTorch Tensors:

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Create random tensors
for data and weights

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Forward pass: compute predictions and loss

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred
    grad_h_relu = grad_y_pred.mm(w2.t()
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Backward pass: manually compute gradients

PyTorch: Tensors

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Gradient descent step on weights

PyTorch: Tensors

To run on GPU, just cast tensors to a cuda datatype!

```
import torch

dtype = torch.cuda.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Autograd

A PyTorch **Variable** is a node in a computational graph

x.data is a Tensor

x.grad is a Variable of gradients (same shape as x.data)

x.grad.data is a Tensor of gradients

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

PyTorch Tensors and Variables have the same API!

Variables remember how they were created (for backprop)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

We will not want gradients (of loss) with respect to data

Do want gradients with respect to weights

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Forward pass looks exactly the same as the Tensor version, but everything is a variable now

PyTorch: Autograd

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Compute gradient of loss with respect to w1 and w2 (zero out grads first)

PyTorch: Autograd

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Make gradient step on weights

Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static**)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic**)

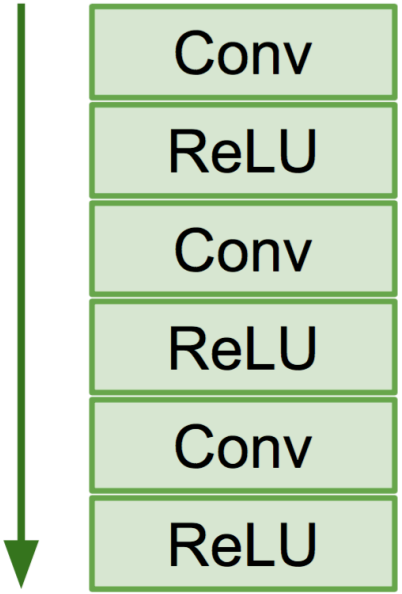
```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

New graph each iteration

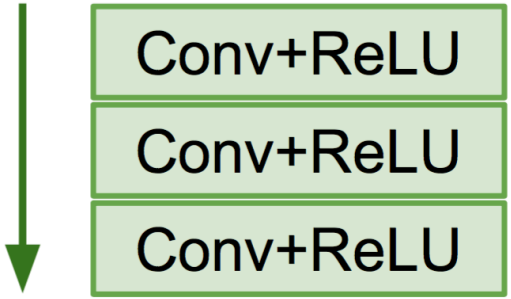
Static vs Dynamic: Optimization

With static graphs, framework can **optimize** the graph for you before it runs!

The graph you wrote



Equivalent graph with **fused operations**



Static vs Dynamic: Serialization

Static

Once graph is built, can **serialize** it and run it without the code that built the graph!

Dynamic

Graph building and execution are intertwined, so always need to keep code around

pytorch-operator

kubeflow / **pytorch-operator**

Code | Issues 7 | Pull requests 0 | Projects 0 | Wiki | Insights

Repository for supporting pytorch. This is experimental.

Experimental repo notice: This repository is experimental and currently only serves as a proof of concept for running distributed training with **PyTorch on Kubernetes**.

pytorch-operator

Using the PyTorch Operator

Run the following to deploy the operator to the namespace of your current context:

```
RBAC=true #set false if you do not have an RBAC cluster
helm install pytorch-operator-chart -n pytorch-operator --set rbac.install=${RBAC} --wait --replace
```

values.yaml

```yaml
# Docker image to use.
image: jgaguirr/pytorch-operator:latest

# Which cloud provider is kubernetes hosted on.
# Supported values are gke or azure.
# Leave blank to use a default, non-cloud specific config.
cloud:

## Wether the dashboard should be installed and the kind of service to use
dashboard:
  install: false
  serviceType: ClusterIP

config:
  configmap:
  file: /etc/config/controller-config-file.yaml

## Install Default RBAC roles and bindings
rbac:
  install: false
  apiVersion: v1beta1
```

Chart.yaml

```yaml
appVersion: 0.1.0
description: K8s Custom Resource and Operator For PyTorch Jobs
home: https://github.com/kubeflow/pytorch-operator
name: pytorch-operator
sources: ['https://github.com/kubeflow/pytorch-operator']
version: 0.1.0
```

Using the PyTorch Operator

For this POC example we will use a configmap that contains our distributed training script.

```
kubectl create -f examples/mnist/configmap.yaml
```

Create a PyTorchJob resource to start training:

```
kubectl create -f examples/mnist/pytorchjob.yaml
```

You should now be able to see the job running based on the specified number of replicas.

```
kubectl get pods -a -l pytorch_job_name=example-job
```

pytorch-operator

pytorchjob.yaml

```yaml
apiVersion: "kubeflow.org/v1alpha1"
kind: "PyTorchJob"
metadata:
  name: "example-job"
spec:
  backend: "tcp"
  masterPort: "23456"
  replicaSpecs:
    - replicas: 1
      replicaType: MASTER
      template:
        spec:
          containers:
          - image: pytorch/pytorch:latest
            imagePullPolicy: IfNotPresent
            name: pytorch
            volumeMounts:
              - name: training-result
                mountPath: /tmp/result
              - name: entrypoint
                mountPath: /tmp/entrypoint
            command: [/tmp/entrypoint/dist_train.py]
          restartPolicy: OnFailure
          volumes:
            - name: training-result
              emptyDir: {}
            - name: entrypoint
              configMap:
                name: dist-train
                defaultMode: 0755
          restartPolicy: OnFailure
    - replicas: 3
      replicaType: WORKER
      template:
        spec:
          containers:
          - image: pytorch/pytorch:latest
            imagePullPolicy: IfNotPresent
            name: pytorch
            volumeMounts:
              - name: training-result
                mountPath: /tmp/result
              - name: entrypoint
                mountPath: /tmp/entrypoint
            command: [/tmp/entrypoint/dist_train.py]
          restartPolicy: OnFailure
          volumes:
            - name: training-result
              emptyDir: {}
            - name: entrypoint
              configMap:
                name: dist-train
                defaultMode: 0755
          restartPolicy: OnFailure
```

Using the PyTorch Operator

Tail the logs for a pod to see its training progress or final status:

```
PODNAME=$(kubectl  get pods -a -l pytorch_job_name=example-job,task_index=0 -o name)
kubectl logs -f ${PODNAME}
```

Example output:

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Processing...
Done!
Rank  0 , epoch  0 :  1.2753884393269066
Rank  0 , epoch  1 :  0.5752273188915842
Rank  0 , epoch  2 :  0.4370715184919616
Rank  0 , epoch  3 :  0.37090928852558136
Rank  0 , epoch  4 :  0.3224359404430715
Rank  0 , epoch  5 :  0.29541213348158385
Rank  0 , epoch  6 :  0.27593734307583967
Rank  0 , epoch  7 :  0.25898529327055536
```

目录

DEMO

目录

Future Work

- Fancy user interface
- Easy to deploy (minikube)
- Standard form for researchers to publish their code with one command
- Operator
  - More frameworks support (PyTorch, Caffe2 WIP)
- Hyperparameter tuning
  - NAS
  - More search algorithms