Ray

Opis

Ray je sučelje za raspodijeljeni proračun najpoznatijih knjižnica za strojno učenje kojim se nastoji olakšati razvoj i korištenje ML aplikacija na više procesorskih jedinica i/ili čvorova. U svojoj osnovi, sastoji se od sučelja Ray Core koje omogućava razvoj distribuiranog koda kroz korištenje dekoratora, no najveću funkcionalnost pruža kroz sučelja koja raspodjeljuju dijelove tipičnog machine learning pipelinea:

- Ray Data za rukovanje podacima
- Ray Train za treniranje modela
- Ray Tune za optimizaciju hiperparametara
- Ray RLlib za ojačano učenje

Verzije

verzija	modul	python	Supek	Padobran
2.4.0	scientific/ray/2.4.0-rayproject	3.9	✅	

⚠ **Korištenje aplikacije na Supeku**

Python aplikacije i knjižnice na Supeku su dostavljene u obliku kontejnera i zahtijevaju korištenje wrappera kao što je opisano ispod.

Više informacija o python aplikacijama i kontejnerima na Supeku možete dobiti na sljedećim poveznicama:

- Python, pip i conda
- Apptainer

Dokumentacija

- Službena dokumentacija - https://www.ray.io/
- Priručnik - https://docs.ray.io/en/latest

Primjeri

Ispod se nalaze neki primjeri funckionalnosti koje Ray pruža poput:

1. Sučelja Ray Train u svrhu treniranja modela PyTorch
2. Sučelja Ray Tune u svrhu optimizacije hiperparametara modela TensorFlow
3. Sučelja Ray Air u svrhu pronalaženja najboljeg Scikit-Learn modela za klasifikaciju

Ray Train + PyTorch

train-pytorch.sh

```
#!/bin/bash

#PBS -l select=2:ngpus=2:ncpus=16

# environment
module load scientific/ray/2.4.0-rayproject

# cd
cd ${PBS_O_WORKDIR:-""}

# run
ray-launcher.sh train-pytorch.py
```

train-pytorch.py

```
# source
# - https://docs.ray.io/en/latest/train/examples/pytorch/torch_resnet50_example.html#torch-fashion-mnist-ex
# - https://pytorch.org/vision/stable/generated/torchvision.datasets.FakeData.html#torchvision.datasets.FakeData

import os
import sys
import time
import pprint
import argparse
import datetime

from typing import Dict
from ray.air import session

import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.models import resnet50
from torchvision.transforms import ToTensor

import ray.train as train
from ray.train.torch import TorchTrainer
from ray.air.config import ScalingConfig

training_data = datasets.FakeData(size=2560,
                                  transform=ToTensor())

test_data = datasets.FakeData(size=256,
                              transform=ToTensor())

def train_epoch(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset) // session.get_world_size()
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
```

```
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def validate_epoch(dataloader, model, loss_fn):
    size = len(dataloader.dataset) // session.get_world_size()
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n "
          f"Accuracy: {(100 * correct):>0.1f}%, "
          f"Avg loss: {test_loss:>8f} \n")
    return test_loss

export IMAGE_PATH="${PWD}/ray-2.4.0-rayproject.sif"
export PATH=${PWD}:$PATH
def train_func(config: Dict):
    batch_size = config["batch_size"]
    lr = config["lr"]
    epochs = config["epochs"]

    worker_batch_size = batch_size

    # Create data loaders.
    train_dataloader = DataLoader(training_data, batch_size=worker_batch_size)
    test_dataloader = DataLoader(test_data, batch_size=worker_batch_size)

    train_dataloader = train.torch.prepare_data_loader(train_dataloader)
    test_dataloader = train.torch.prepare_data_loader(test_dataloader)

    # Create model.
    model = resnet50()
    model = train.torch.prepare_model(model)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    loss_results = []

    for _ in range(epochs):
        train_epoch(train_dataloader, model, loss_fn, optimizer)
        loss = validate_epoch(test_dataloader, model, loss_fn)
        loss_results.append(loss)
        session.report(dict(loss=loss))

    # return required for backwards compatibility with the old API
    # TODO(team-ml) clean up and remove return
    return loss_results


def main():

    # node & worker info
    resources = ray.cluster_resources()
    gpus = int(resources['GPU'])
    cpus = int(resources['CPU'])
    resources_per_worker = {'GPU': 1,
```

```
                           'CPU': (cpus-1)//gpus}

    # scaling_config
    scaling_config = ScalingConfig(use_gpu=True,
                                   num_workers=gpus,
                                   resources_per_worker=resources_per_worker)

    # trainer
    trainer = TorchTrainer(train_func,
                           scaling_config=scaling_config,
                           train_loop_config={"lr": 1e-3,
                                              "epochs": 3,
                                              "batch_size": 256})
    result = trainer.fit()
    print(f"Results: {result.metrics}")

if __name__ == "__main__":
    import ray
    ray.init(address='auto',
             _node_ip_address=os.environ['NODE_IP_ADDRESS'])
    main()
```

Ray Tune + TensorFlow

tune-tensorflow.sh

```
#!/bin/bash

#PBS -l select=2:ngpus=2:ncpus=16

# environment
module load scientific/ray/2.4.0-rayproject

# cd
cd ${PBS_O_WORKDIR:-""}

# run
ray-launcher.sh tune-tensorflow.py
```

tune-tensorflow.py

```
import argparse
import os

import ray
from ray import air, tune
from ray.tune.schedulers import AsyncHyperBandScheduler
from ray.tune.integration.keras import TuneReportCallback

import tensorrt
import numpy as np
import tensorflow as tf
from tensorflow import keras

def train_mnist(config):

    # data
    X = np.random.uniform(size=[samples, 224, 224, 3])
    y = np.random.uniform(size=[samples, 1], low=0, high=999).astype("int64")
    dataset = tf.data.Dataset.from_tensor_slices((X, y))

    # model
    model = tf.keras.applications.ResNet50(weights=None)
```

```
    loss = tf.keras.losses.SparseCategoricalCrossentropy()

    optimizer = tf.optimizers.SGD(lr=config["lr"],
                                  momentum=config["momentum"])

    model.compile(optimizer=optimizer,
                  loss=loss,
                  metrics=["accuracy"])

    # fit
    model.fit(X,
              y,
              batch_size = batch_size,
              epochs = epochs,
              verbose = 0,
              callbacks = [TuneReportCallback({"mean_accuracy": "accuracy"})])

def main():

    # vars
    epochs = 10
    batch_size = 256
    samples = batch_size*10
    num_classes = 1000

    # resources
    resources = ray.cluster_resources()
    gpus = int(resources['GPU'])
    cpus = int(resources['CPU'])
    resources_per_worker = {'GPU': 1,
                            'CPU': (cpus-1)//gpus}

    # scheduler
    sched = AsyncHyperBandScheduler(time_attr="training_iteration",
                                    max_t=400,
                                    grace_period=20)

    # tuner
    trainable = tune.with_resources(train_mnist,
                                    resources=resources_per_worker)

    param_space = {"threads": 2,
                   "lr": tune.uniform(0.001, 0.1),
                   "momentum": tune.uniform(0.1, 0.9)}

    tune_config = tune.TuneConfig(metric="mean_accuracy",
                                  mode="max",
                                  scheduler=sched,
                                  num_samples=10)

    run_config = air.RunConfig(name="exp",
                               stop={"mean_accuracy": 1e-4,
                                     "training_iteration": 100})

    tuner = tune.Tuner(trainable=trainable,
                       tune_config=tune_config,
                       run_config=run_config,
                       param_space=param_space)

    results = tuner.fit()

    print("Best hyperparameters found were: ", results.get_best_result().config)

if __name__ == "__main__":
    import ray
    ray.init(address='auto',
             _node_ip_address=os.environ['NODE_IP_ADDRESS'])
    main()
```

Ray AIR + Scikit-Learn

sklearn-automl.sh

```
#!/bin/bash

#PBS -l select=12:ngpus=1:ncpus=4

# environment
# module load scientific/ray/2.4.0-rayproject

# cd
cd ${PBS_O_WORKDIR:-""}

# run

ray-launcher.sh sklearn-automl.py
```

sklearn-automl.py

```
# sources:
# 1) data - https://archive.ics.uci.edu/dataset/15/breast+cancer+wisconsin+original

import os
import glob
import numpy as np
import pandas as pd
import itertools

import ray
from ray import air, tune
from ray.air import Checkpoint, session
from ray.train.sklearn import SklearnTrainer

from sklearn.datasets import make_classification
from sklearn.model_selection import cross_validate
from sklearn.metrics import mean_squared_error, mean_absolute_error

from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier

def get_estimators(estimators):
    for model, options in estimators.items():
        options_cross = itertools.product(*options.values())
        options_keys = options.keys()
        for options_row in options_cross:
            yield model(**{key: value for key, value in zip(options_keys, options_row)})

def cross_validate_config(config, data):

    # X, y
    df = data.to_pandas()
    X = df.drop(columns=['id-number', 'diagnosis'])
    y = df['diagnosis'].map({'M': 1, 'B': 0})

    # cv
    result = cross_validate(estimator=config['estimator'],
                            X=X,
                            y=y,
                            scoring="f1",
                            n_jobs=1)

    # output
    results = { "f1": result['test_score'].mean() }
```

```
    session.report(results)

def main():

    # train, test
    data = ray.data.read_csv('wdbc.data')
    train, test = data.train_test_split(test_size=0.2)

    # estimator space
    estimators = {
        SVC: {
            'kernel': ['linear',
                       'poly',
                       'rbf',
                       'sigmoid'],
            'C': [1,
                  4,
                  16],
        },
        DecisionTreeClassifier: {
            'max_depth': [None,
                          2,
                          8,
                          32],
            'splitter': ['best',
                         'random'],
        },
        KNeighborsClassifier: {
            'algorithm': ['auto',
                          'ball_tree',
                          'kd_tree',
                          'brute'],
            'weights': ['uniform',
                        'distance'],
        },
        MLPClassifier: {
            'hidden_layer_sizes': [10,
                                   40,
                                   160],
            'activation': ['identity',
                           'logistic',
                           'tanh'],
        },
    }
    estimators = list(get_estimators(estimators))

    # grid search
    trainable = tune.with_parameters(cross_validate_config, data=train)

    param_space = {"estimator": tune.grid_search(estimators),
                   "n_splits": 5}

    tune_config = tune.TuneConfig(metric="f1",
                                  mode="max")

    tuner = tune.Tuner(trainable=trainable,
                       param_space=param_space,
                       tune_config=tune_config)

    result_grid = tuner.fit()
    best_result = result_grid.get_best_result()
    print(best_result)
    print(best_result.config)

if __name__ == '__main__':
    import ray
    ray.init(address='auto',
             _node_ip_address=os.environ['NODE_IP_ADDRESS'])
    main()
```

Napomene

Izvođenje aplikacija putem wrappera

Za ispravno korištenje knjižnice Ray, dostavljen je wrapper **ray-launcher.sh** koji osigurava pravilno zauzimanje dodijeljenih resursa stvaranjem Ray klastera i omogućava pokretanje aplikacija kroz ray.init API

U skripti posla PBS, potrebno je pozvati skriptu Python na sljedeći način:

```
# aktiviraj Ray
module load scientific/ray/2.4.0-rayproject

# pokreni aplikaciju
ray-launcher.sh moj_program.py
```

Dok se u skripti Python sučelje Ray incijalizira na sljedeći način:

```
import os
import ray

ray.init(address='auto',
         _node_ip_address=os.environ['NODE_IP_ADDRESS'])

... Python kod ...
```