JAX – Now and Future

Nasy

Jul 16, 2023

Contents

1 Introduction

1.1 What is JAX?

1.1.1 Acronym

Originally Just After eXecution

Now JAX is Autograd (automatic obtaining of the gradient function through differentiation of a function) an XLA (accelerated linear algebra).

Fact JAX is JAX

ref: https://github.com/google/jax/discussions/9019

1.1.2 Design

- Follow `numpy` as closely as possible
- Works with various existing frameworks (PyTorch, Tensorflow)
- Immutable and purely functional
- Asynchronous dispatch
- Core: `jit`, `vmap`, `grad`, `pmap`

1.2 Who use JAX?

- Lab
  - Google Brain & Deepmind
  - Google Research
- Models
  - ViT
  - Big Vision
  - AlphaFold
  - MLP-Mixer
  - T5X
  - PaLM
  - ...
- Awesome-jax: https://github.com/n2cholas/awesome-jax

1.3 Basic Usage

1.3.1 JAX and NumPy

1. JAX is accelerated NumPy

   []python import jax import jax.numpy as jnp import numpy as np from rich import print

   print("numpy:", np.asarray([1, 2, 3])) print("jax:", jnp.asarray([1, 2, 3]))

   []python print("jax->numpy:", np.std(jnp.arange(10))) print("numpy->jax:", jnp.std(np.arange(10)))

2. Difference

   JAX is designed to be functional, as in functional programming.

   []python x = np.arange(10) jx = jnp.arange(10) print(f"Original: x=") x[0] = 1 print(f"Inplace replace: x=")

   cannot jx[0] = 1 jx = jx.at[0].set(1)

   def in$_{p}lace_{s}et(x, i, v) : x[i] = vreturnx$

in$_{p}lace_{s}et(x, 2, 10)$

   print(f"Inplace replace 2: x=")

1.3.2 JIT

Using a just-in-time (JIT) compilation decorator, sequences of operations can be optimized together and run at once.

ref: https://jax.readthedocs.io/en/latest/_autosummary/jax.jit.html#jax.jit

ref2, tutorial: https://jax.readthedocs.io/en/latest/jax-101/02-jitting.html

Here is an example to JIT a function.

[]python def func(x: Array) -> Array: ...

jit$_{f}unc = jax.jit(func)$

or

@jax.jit def func(x: Array) -> Array: ...

In most of time, you only need add the jit funciton in the outer most function.

[]python import jax import jax.numpy as jnp import rich import time

def model(params, x): return params[”w”] * x + params[”b”]

def loss$_{f}unction(params, x, y) : return((model(params, x) - y) * * 2).mean()$

def train$_{s}tep(params, x, y, lr) : grads = jax.grad(loss_{f}unction)(params, x, y) return ”w” : params[$

def train(x, y, lr, num$_{s}teps) : params = ”w” : 0.0, ”b” : 0.0 for i in range(num_{s}teps) : params = train_{s}tep(params, x, y, lr) return params$

train$_{s}tep_{j}it = jax.jit(train_{s}tep)$

def train$_{j}it(x, y, lr, num_{s}teps) : params = ”w” : 0.0, ”b” : 0.0 for i in range(num_{s}teps) : params = train_{s}tep_{j}it(params, x, y, lr) return params$

xs = jnp.array([1, 2, 3, 4, 5]) / 2 ys = jnp.array([2, 4, 6, 8, 10]) / 2

t1 = time.time() p1 = train(xs, ys, 0.01, 50) t2 = time.time()

t3 = time.time() p2 = train$_{j}it(xs, ys, 0.01, 50) t4 = time.time()$

rich.print(”no jit:”, p1, t2 - t1) rich.print(”jit:”, p2, t4 - t3)

1.3.3 autograd

JAX use `jax.grad` and `jax.value_and_grad` to get the gradient of a function.

[]python def f(x): return x ** 3

print(jax.grad(f)(10.0)) 3x$^{2} print(jax.value_{a}nd_{g}rad(f)(10.0)) (x^{3}, 3x^{2}) print(jax.grad(jax.grad(f))$
$6x$

You can also do partial differentiation with some structure data in JAX.

[]python def f(xy): return xy[”x”] ** xy[”y”]

x = jnp.array(2.) y = jnp.array(3.)

print(jax.value$_{a}nd_{g}rad(f)(”x” : x, ”y” : y)) Higher order derivatives print(jax.hessian(f)(”x” : x, ”$

1.3.4 vmap and pmap

JAX for single-program, multiple-data (SPMD).

`jax.vmap(f)(x)`, where the shape of `x` is `batch_size, ...`

Here is an example for `vmap`

[]python we want to calculate the gradient for x and y, however, our x and y is batched. @jax.grad def f2(xy): x, y = xy return x ** y

xs = jnp.array([2., 2.]) ys = jnp.array([3., 3.])

for-loop grads = [] for x, y in zip(xs, ys): grads.append(f2((x, y))) print("For-loop:", grads)

vmap vmap$_g rads = jax.vmap(f2) print("vmap : ", vmap_g rads((xs, ys)))$

How about `pmap`?

`pmap` is like `vmap`, but parallel evaluate the function on different devices.

`jax.pmap(f)(x)`, where the shape of `x` is `devices, ...`

And you can use both `pmap` and `vmap`, like `jax.pmap(jax.vmap(f))` the shape will be `devices, batch_size, ...`

1.3.5 Performance

Here is a comparison between numpy, jax and pytorch.

[]python import time import torch np.random.seed(42)

arr = np.random.random(1000000).reshape(-1, 1000).astype("float32") * 10 jrr = jnp.array(arr)

def $_f unc(x) : return x@x@x@x@x@x@x@x@x@x$

def func(x): return $(_f unc(x) *_f unc(x) +_f unc(x) *_f unc(x))/_f unc(x)$

import time

numpy time t1 = time.time() func(arr) t2 = time.time()

jax time t3 = time.time() func(jrr).block$_u ntil_r eady() t4 = time.time()$

jax jit compile jit$_f unc = jax.jit(func) jit_f unc(jrr).block_u ntil_r eady()$

jax jit time t5 = time.time() jit$_f unc(jrr).block_u ntil_r eady() t6 = time.time()$

trr = torch.from$_n umpy(arr)$

torch time func(trr) t7 = time.time() func(trr) t8 = time.time()

print("Numpy time: ", t2 - t1) print("Jax time: ", t4 - t3) print("Jax jit time: ", t6 - t5) print("Torch time: ", t8 - t7)

2 JAX vs PyTorch in a Pipeline

2.1 Keras

Those who have previously used TensorFlow should be quite familiar with Keras. Currently, Keras has reached version 3.0 and includes the 'keras-core' library, which allows for very easy switching between JAX, TensorFlow, and PyTorch.

[]sh run with jax KERAS$_{B}ACKEND = jaxpythontrain.py$

run with torch KERAS$_{B}ACKEND = torchpythontrain.py$

run with tensorflow KERAS$_{B}ACKEND = tensorflowpythontrain.py$

2.2 Install and run

Install

[]sh jax pip install –upgrade ”jaxlib[cuda12$_{p}ip]”” jax[cuda12_{p}ip]” - fhttps : //storage.googleapis.com/jax-releases/jax_{c}uda_{r}eleases.htmljaxnnlibrarypipinstallflax$

torch pip install torch

Run. As far as I know, both `JAX` and `PyTorch` now ship with nvidia-cuda-toolkits, so you do not need to setup `LD_LIBRARY_PATH` anymore.

[]sh jax python train.py

torch python train.py

2.3 Load data

JAX does not have built-in data loading utilities, so we can use both `tensorflow` or `torch` dataloader to load the dataset.

Here is a example for `torch` dataloader.

[]python import jax.numpy as jnp from jax.tree$_{u}tilimporttree_{m}apfromtorch.utilsimportdata$

def collate$_{f}n(x) : returntree_{m}ap(jnp.asarray, data.default_{c}ollate(batch))$

data$_{g}enerator = Dataloader(dataset, collate_{f}n = collate_{f}n, batch_{s}ize = 128, shuffle = False, num_{w}orkers = 2,)$

And you can find others in the `jax` document. https://jax.readthedocs.io/en/latest/advanced_guide.html

2.4 Define and initialize model

2.4.1 PyTorch

[]python import torch from torch import nn

class TM(nn.Module): ”””Torch Model.”””

def $_{i}nit_{(self,in=100,h1=300,h2=200,h3=100):super().}{_{i}nit}_{()self.l1=nn.Linear(in_{,}h1)self.bn1=nn.BatchNorm1d(h1)self.dp1=nn.Dropout}$

def forward(self, x): x = torch.relu(self.l1(x)) x = self.bn1(x) x = self.dp1(x) x = torch.relu(self.l2(x)) x = torch.relu(self.l3(x)) return self.out(x)

tm = TM() print(tm) rich.print(jax.tree$_{m}ap(lambdax : x.shape, dict(tm.state_{d}ict())))$

2.4.2 JAX w/ flax

[]python import jax import jax.numpy as jnp from flax import linen

class JM(linen.Module): """JAX and flax model."""

h1 = 300 h2 = 200 h3 = 100

@linen.compact def $_{call_{(self,x,training=True):x=linen.Dense(self.h1)(x)x=linen.BatchNorm()(x,use_running_average=nottraini}}$

jm = JM() variables = jm.init(jax.random.key(42), random key jnp.ones((1, 100)), input (Batch, Features) training=False, training mode)

rich.print(jax.tree$_map(lambdax : x.shape, variables))print(jm.tabulate(jax.random.key(42), jnp.c$ $False, compute_flops = True, compute_vjp_flops = True)$)

2.5 TrainState and Train loop

Usually, we need to store the state of the model, like the parameters, the optimizer, the learning rate scheduler, etc.

2.5.1 PyTorch

[]python import torch import torch.nn as nn from typing import NamedTuple

def train$_loop(conf : dict, dataloader) : """Thepytorchtrainingloop."""model = TM(conf)$

loss$_fn = nn.CrossEntropyLoss()$

optim = torch.optim.Adam(model.parameters(), lr=conf["lr"])

state = TrainState(loss=0, state=model.state$_dict(), optim_state = optim.state_dict(), step = 0, metric = 0,$)

model.to("cuda")

def train$_step(batch, ys) : batch, ys = batch.to("cuda"), ys.to("cuda")$

forward loss = loss$_fn(model(batch), ys)gradloss.backward()updateparamsoptim.step()optim.zero_g$

return loss

for e in range(conf["epochs"]): model.train() for i, (batch, ys) in dataloader: loss = train$_step(batch, ys)$

eval model.eval() metric = ... compute metric

state = state.$_replace(loss = loss.item(), state = model.state_dict(), optim_state = optim.state_dict(), step = state.step + 1, metric = metric,$)

save torch.save(state, "model.pt")

2.5.2 JAX

[]python import jax import jax.numpy as jnp import optax from flax.core.scope import Collection from flax.training import train$_stateimportpickle$

class TrainState(train$_{s}tate.TrainState)$: """$Trainingstates.$"""

default apply$_{f}nthemodelforwardfunctionparamstxoptimsteptrainingstep$

our custom batch$_{s}tats$: $Collectionourmetricsloss$: $jax.Arraymetric$: $jax.Array$

def create$_{t}rain_{s}tate(conf : dict)$: """$Createinitialtrainingstate.$"""$model = JM(conf)$

@jax.jit def init() -> Collection: return model.init(jax.random.key(42), jnp.ones((1, 100)), training=False)

variables = init()

return TrainState.create(apply$_{f}n = model.apply, params = variables[\text{"}params\text{"}], tx = optax.adamw(conf[\text{"}lr\text{"}]), model_{s}tate = Collection(), loss = jnp.inf, metric =$ $0.0, batch_{s}tats = variables[\text{"}batch_{s}tats\text{"}], step = 0,)$

@jax.jit def train$_{s}tep(state, rng, batch, ys)$:

@jax.jit optional since we already have jit outside train$_{s}tep@jax.value_{a}nd_{g}raddeflossfn(params)$: $returnoptax.cross_{e}ntropy_{l}oss(state.apply_{f}n(\text{"}params\text{"} : state.params, \text{"}batch_{s}tats\text{"} : state.batch_{s}tats,$ $True, rngs = \text{"}dropout\text{"} : rng, mutable = [\text{"}batch_{s}tats\text{"}],), ys,).mean()$

loss, grad = lossfn(state.params) the step, params, tx$_{s}tateswillautomaticallybeupdatedreturnstate$ $grad, loss = loss)$

@jax.jit def eval$_{s}tep(state, batch, ys)$:

value = state.apply$_{f}n(\text{"}params\text{"} : state.params, \text{"}batch_{s}tats\text{"} : state.batch_{s}tats, batch, training =$ $False,)metric = ...returnmetric$

def train$_{l}oop(conf : dict, dataloader)$: """$Trainingloop.$"""$state = create_{t}rain_{s}tate(conf)$

for epoch in range(conf["epochs"]): for batch, ys in dataloader: state = train$_{s}tep(state, batch)$

eval metric = eval$_{s}tep(state, batch, ys)state = state.replace(metric = metric)$

save with open("model.pkl", "wb") as f: pickle.dump("params": state.params, "tx": state.tx, "batch$_{s}tats\text{"} : state.batch_{s}tats, \text{"}metric\text{"} : state.metric, f)$

3 Parallel and Distributed Computing

3.1 Resource

- Flax tutorial: https://flax.readthedocs.io/en/latest/guides/parallel_training/index.html
- JAX tutorial: https://jax.readthedocs.io/en/latest/notebooks/Distributed_arrays_and_automatic_parallelization.html

3.2 Transfer data between devices

[]python import jax import jax.numpy as jnp import os

os.environ["XLA$_F LAGS$"] = " $--xla_f orce_h ost_p latform_d evice_c ount =$ 8"

x = jnp.ones((8, 8))

1. check devices print("global devices:", jax.devices()) print("local devices:", jax.local$_d evices())youcanspecifythedevicetypeprint("cpudevices :$ ", $jax.devices("cpu"))$

2. check the device of x print("x devices:", x.devices())

Put x to a specific device y = jax.device$_p ut(x, jax.devices("cpu")[1])Getybacktohost(numpyarray)z$ $jax.device_g et(y)print("xdevices : ", x.devices())print("ydevices : ", y.devices())print("ztype :$ ", $type(z))$

3.3 Use `pmap` to train with data parallel.

[]python import jax from functools import partial

@partial(jax.pmap, static$_b roadcasted_a rgnums = (0,))defcreate_t rain_s tate(conf :$ $dict) : """Createinitialtrainingstate.""" ...$

@jax.pmap def train$_s tep(state, rng, batch, ys) : ...$

@jax.jit def eval$_s tep(state, batch, ys) :$

preds = state.apply$_f n((batch, 1)"params" : state.params, "batch_s tats" : state.batch_s tats, batch, tra$ $False,)logits = jax.nn.sigmoid(preds)acc = jnp.mean((logits > 0.5) ==$ $ys)returnacc$

@partial(jax.pmap, axis$_n ame = "batch")defeval_s tep_p map(state, batch, ys) :$

preds = state.apply$_f n((devices, batch, 1)"params" : state.params, "batch_s tats" : state.batch_s tats, b$ $False,)logits = jax.nn.sigmoid(preds)pacc = jax.lax.pmean((logits >$ $0.5) == ys, axis_n ame = "batch")acc = jnp.mean(pacc)returnacc$

3.4 Use jit with sharding.

sharding, split a large array into smaller pieces, and each piece is stored on a different device.

3.4.1 Basic

[]python import jax import jax.numpy as jnp from jax.experimental import mesh$_u tilsfromjax.shardingimportPositionalSharding, NamedShardingimportnumpyasnp$

import os

os.environ["XLA$_F LAGS$"] = " $--xla_f orce_h ost_p latform_d evice_c ount =$ 8" dc = jax.device$_{c}ount()print(f"wehavedcdevices.")$

If you want to track gpu usage install go pip install jax-smi from jax$_{s}miimportinitialise_{t}rackinginitialise_{t}racking()$

batch feature (batch, feature), here batch = Nxdevice$_{c}ountxs = jax.random.normal(jax.random.k$ $dc * 64, 8 * dc * 64))$

dmesh dmesh = mesh$_{u}tils.create_{d}evice_{m}esh((dc,))ordmesh = np.asarray(jax.devices()).reshape($

sharding = PositionalSharding(dmesh)

since xs shappe is (batch, feature) sharding = sharding.reshape((-1, 1)) put jax across devices print(xs.devices()) xs = jax.device$_{p}ut(xs, sharding)print(xs.devices())$

jax.debug.visualize$_{a}rray_{s}harding(xs)$

3.4.2 Different shardings

[]python xs = jax.device$_{p}ut(xs, sharding.reshape(1, -1))$

jax.debug.visualize$_{a}rray_{s}harding(xs)$

[]python xs = jax.device$_{p}ut(xs, sharding.reshape(dc//2, -1))jax.debug.visualize_{a}rray_{s}harding(xs$

3.4.3 Performance

[]python xs = jax.device$_{p}ut(xs, sharding.reshape(dc//2, -1))xsc0 = jax.device_{p}ut(xs, jax.devices()[0]$

cos = jax.jit(jnp.cos) cos(xsc0).block$_{u}ntil_{r}eady()cos(xs).block_{u}ntil_{r}eady()$

t1 = time.time() for $_{i}nrange(20) : r = cos(xsc0).block_{u}ntil_{r}eady()t2 = time.time()$

t3 = time.time() for $_{i}nrange(20) : rr = cos(xs).block_{u}ntil_{r}eady()t4 = time.time()$

cos = jax.pmap(jnp.cos)

pxs = xsc0.reshape(dc, -1, xsc0.shape[-1])

t5 = time.time() for $_{i}nrange(20) : rrr = cos(pxs).block_{u}ntil_{r}eady()t6 = time.time()$

print("In single device: ", t2 - t1) print("In multi device: ", t4 - t3) print("In multi device pmap: ", t6 - t5)

3.4.4 Use sharding with `jit`

[]python xs = jax.device$_{p}ut(xs, sharding.reshape(dc, -1))$

def f(x): return jnp.sin(x)

in with None, out with (dc, 1)

print("before:") jax.debug.visualize$_{a}rray_{s}harding(xs)$

out = jax.jit(f, in$_{s}hardings = sharding.reshape(dc, -1), out_{s}hardings = sharding.reshape(-1, dc))(xs)print("after : ")jax.debug.visualize_{a}rray_{s}harding(out)$

4 JAX echosystem

- JAX
- NN library
  - Flax (Google)
  - Haiku (Deepmind)
  - Trax (Google brain)
  - HuggingFace (Flax)
  - keras
  - jraph (GNN)
  - RLax (Deepmind, RL)
  - Coax (Microsoft, RL)
- Optimizer
  - jaxopt
  - optax
- Others
  - orbax-checkpoint
  - jax-md (molecular dynamics)
  - mpi4jax