AN INTRODUCTION TO JAX [BRADBURY ET AL., 2018]

TATJANA CHAVDAROVA

IDIAP RESEARCH INSTITUTE &
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE – EPFL

February 26, 2020

XLA

An optimizing compiler for Machine Learning (ML)

XLA: Accelerated Linear Algebra

An optimizing compiler for ML

- ✓ Eliminates operations (*op*) dispatch overhead
- ✓ Fuses *ops* to avoid round trips to memory:
  - instead of storing the result of *op*1 and loading it for *op*2, fuse *op*1 and *op*2
- ✓ does analysis and choices adjusted to the hardware constraints
- unrolls and vectorizes via known dimensions

JAX: SHORT OVERVIEW

Main Components

JAX

Python + *NumPy* + FUNCTIONAL PROGRAMMING

"JAX is an extensible system for **composable function transformations** of *Python* + *NumPy* code." – as defined by its developers.

- *NumPy* based *Python* library that supports automatic differentiation
- Just–in–time (*jit*) compilation to run efficiently on an accelerator via *XLA*
- math–like programming:
  - Modular functions which map input to output, that do not affect global variables
- uses asynchronous execution by default[1]
- main execution does *not* halt until the result is returned by a processing unit, unless needed (*e.g.* to print it or to use it), instead *JAX* returns *DeviceArray*
- *DeviceArray* is a "result from the future" array whose dimensions we can inspect
- Its developers aim at making it easy to use and focus on reproducibility:
  - minimal & expressive API (based on *NumPy*)
  - same API for CPU/GPU/TPU
  - reproducible results: *JAX*'s seed fixing slightly differs (will come back to this)

[1] Use *block_until_ready* when benchmarking, as without it we would measure the time needed to dispatch the commands to the GPU/TPU, rather then executing them.

JAX

Python + *NumPy* + FUNCTIONAL PROGRAMMING

"JAX is an extensible system for **composable function transformations** of *Python* + *NumPy* code." – as defined by its developers.

- *NumPy* based *Python* library that supports automatic differentiation
- Just–in–time (*jit*) compilation to run efficiently on an accelerator via *XLA*
- math–like programming:
  - Modular functions which map input to output, that do not affect global variables
- uses asynchronous execution by default[1]
- main execution does *not* halt until the result is returned by a processing unit, unless needed (*e.g.* to print it or to use it), instead *JAX* returns *DeviceArray*
- *DeviceArray* is a "result from the future" array whose dimensions we can inspect
- Its developers aim at making it easy to use and focus on reproducibility:
  - minimal & expressive API (based on *NumPy*)
  - same API for CPU/GPU/TPU
  - reproducible results: *JAX*'s seed fixing slightly differs (will come back to this)

[1] Use *block_until_ready* when benchmarking, as without it we would measure the time needed to dispatch the commands to the GPU/TPU, rather then executing them.

JAX

Python + *NumPy* + FUNCTIONAL PROGRAMMING

"JAX is an extensible system for **composable function transformations** of *Python* + *NumPy* code." – as defined by its developers.

- *NumPy* based *Python* library that supports automatic differentiation
- Just–in–time (*jit*) compilation to run efficiently on an accelerator via *XLA*
- math–like programming:
    - Modular functions which map input to output, that do not affect global variables
- uses asynchronous execution by default[1]
- main execution does *not* halt until the result is returned by a processing unit, unless needed (*e.g.* to print it or to use it), instead *JAX* returns *DeviceArray*
- *DeviceArray* is a "result from the future" array whose dimensions we can inspect
- Its developers aim at making it easy to use and focus on reproducibility:
    - minimal & expressive API (based on *NumPy*)
    - same API for CPU/GPU/TPU
    - reproducible results: *JAX*'s seed fixing slightly differs (will come back to this)

[1] Use *block_until_ready* when benchmarking, as without it we would measure the time needed to dispatch the commands to the GPU/TPU, rather then executing them.

A bit more on Device Array (*JAX* abstract types)

Abstract values = sets of possible inputs

- One key idea: Instead of compiling the code for any possible value, compile it for an **abstract** value, *i.e.* set of possible values.
- *E.g.* we can say the input will be of a particular shape (*e.g.* ShapedArray(float32[1,64])), without giving the exact values.
- ***JAX*** produces a “view” of the ***Python*** code valid for many different argument values, by tracing such abstract values.
- ***JAX*** uses multiple different levels of abstraction, and different transformations use different abstraction levels.
- Classes in ***jax/abstrac_arrays.py*** (all inherit from an ***AbstractValue*** class):
  - ***UnshapedArray*** level 2
  - ***ShapedArray*** level 1
  - ***ConcreteArray*** level 0
  - ***AbstractToken***
- Note the <u>trade–off</u>: abstraction *Vs.* being specific in the code, leading to reduced number of re-compilations and “argument specific” functions (contrary to ***Python***’s paradigm where inputs can be anything), respectively.
- Both of best worlds: Avoid re-compilations and still use traceable control flows → use “Structured control flow primitives”.

Imports & Creating JAX device array

- Note (convention): use “jnp” and “np” for ***jax.numpy*** and original ***numpy***, resp.

```
import jax.numpy as jnp  # JAX numpy (will run on GPU if available)
import numpy as np  # original CPU-backed NumPy
```

- Print available devices:

```
import jax
jax.devices()
```

- Make your multiple CPUs visible to ***JAX***[2] (no need to do so for GPU/TPU):

```
import os
os.environ['XLA_FLAGS'] = '--xla_force_host_platform_device_count={}'.format(12)
import jax
jax.devices()
# outputs: [CpuDevice(id=0), CpuDevice(id=1), CpuDevice(id=2), CpuDevice(id=3),
      CpuDevice(id=4), CpuDevice(id=5), CpuDevice(id=6), CpuDevice(id=7), CpuDevice(id=8),
      CpuDevice(id=9), CpuDevice(id=10), CpuDevice(id=11)]
```

- Creating ***JAX*** device array (similar to ***numpy***)

```
x = jnp.asarray([.1, -.1])  # [ 0.1 -0.1], <class 'jax.interpreters.xla.DeviceArray'>
y = jnp.array(x>=0, dtype=jnp.float32)  # [1. 0.], <class 'jax.interpreters.xla.DeviceArray'>
```

[2] Required to test ***pmap*** if you don’t have GPU/TPU on your laptop.

Main JAX components
jit, *grad* & *vmap*

jit – for speeding up your code (sequential functions are compiled together with ***XLA***)

grad – for taking derivatives

vmap – for automatic vectorization or batching.

```
from jax import jit, grad, vmap
```

Quick dive–in example
from NeurIPS '19 tutorial

```
import jax.numpy as jnp
from jax import jit, grad, vmap

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

gradient_fun = jit(grad(loss))
```

jit: XLA compilation

Forces compilation with XLA the first time a function is called, and is cached thereafter

- JAX uses XLA to compile and run the programs on CPUs/GPUs/TPUs.
- By default, ***jit*** traces the code on the ***ShapedArray*** abstraction level (abstraction level 1); also possible on ***UnshapedArray*** (level 2).
- ***jit*** enforces tracing with abstract value, and ***print*** within ***jit***-ed function looks like this:

```
Traced<ShapedArray(int32[]):JaxprTrace(level=-1/1)>
Traced<ShapedArray(int32[]):JaxprTrace(level=-1/1)>
```

- ***jit***–decorated function is typically called only once to trace its behavior and the resulting computation graph is cached for future calls.

  How would this affect:

  - ***print***-s in ***jit***-ed function?
  - branching/conditioning (***if***s), loops (***for***, ***while***) ?
  - (will revisit this)

jit: XLA compilation
Examples

```
def selu(x, alpha=1.67, lmbda=1.05):
  return lmbda * np.where(x > 0, x, alpha * np.exp(x) - alpha)

x = random.normal(key, (1000000,))
%timeit selu(x).block_until_ready()

# to speed-up with jit:
selu_jit = jit(selu)
%timeit selu_jit(x).block_until_ready()
```

You can also use ***@jit*** as a function decorator:

```
@jit
def fn_name(inputs):
   something = inputs ** 2 / jnp.sqrt(5)
   return something

fn_name(jnp.asarray(jnp.arange(4))) # DeviceArray([0. , 0.4472136, 1.7888544, 4.0249224],
      dtype=float32)
```

jit: XLA COMPILATION

jit + BRANCHING REQUIRES SPECIFYING 'STATIC_ARGNUMS'

The tracing of ***jit*** requires precise value, instead of abstract one *i.e.* the set of all possible values for a Boolean, $\mathit{bool} = \{\mathit{true}, \mathit{false}\}$, hence the error:

```
TypeError: Abstract value passed to ‘bool‘, which requires a concrete value.
The function to be transformed can’t be traced at the required level of abstraction.
If using ‘jit‘, try using ‘static_argnums‘ or applying ‘jit‘ to smaller subfunctions instead.
```

Example:

```
def f(x, y):
  return (3. * y ** 2) if x < 3 else (-4 * y)

fjit = jit(f)
fjit_static_argnums = jit(f, static_argnums=(0,))
fjit_static_argnums_wrong_index = jit(f, static_argnums=(1,))
fjit(5, 5)  # This will fail!
fjit_static_argnums(5, 5)  # This won’t!
fjit_static_argnums_wrong_index(5, 5)  # Fails!
```

jit: XLA COMPILATION
jit + BRANCHING REQUIRES SPECIFYING 'STATIC_ARGNUMS'

Example: ***jit*** with ***for*** loop and 'static_argnums' (default is ***None***):

```
def sum_first_n(x, n):
  _sum = 0.
  for i in range(n):
    _sum += x[i]
  return _sum

sum_first_n = jit(sum_first_n, static_argnums=(0,))  # won't work: "TypeError: 'JaxprTracer' object
      cannot be interpreted as an integer"
sum_first_n = jit(sum_first_n, static_argnums=(1,))  # works!



# or: ---
def sum_first_n(x, n):
  return jnp.sum(x[:n])

sum_first_n = jit(sum_first_n)  # won't work: IndexError: Array slice indices must have static
      start/stop/step to be used with Numpy indexing syntax. ...
sum_first_n = jit(sum_first_n, static_argnums=(1,))  # works!



# e.g. test:
sum_first_n(jnp.array([1., 2., 3., 4., 5.]), 2)  # if it works, returns: DeviceArray(3.,
      dtype=float32)
```

grad: JAX's AUTOMATIC DIFFERENTIATION
grad(function_name)

- $grad(f, argnums = 0)$
- By default, the gradient is taken with respect to the first argument of the function f;
- use ***argnums*** to specify w.r.t. which input argument (***int***) of f the gradient should be computed;
- *e.g.* if our loss function has inputs: $loss(params, x, y)$, the gradient w.r.t. ***params*** is $grad(loss)$.

Can also be combined with ***jit***:

```
jit(grad(some_function(input)))
```

c.f. ***jax.vjp*** & ***jax.jvp*** for more advanced autodiff for reverse–mode vector-Jacobian products and forward-mode Jacobian–vector products, resp.

grad: JAX'S AUTOMATIC DIFFERENTIATION
grad(*function_name*)

```
import jax.numpy as jnp
from jax import grad

def test_fun(x, y):
  return x**2 + x*y + y**2

grad_x_test_fun = grad(test_fun, argnums=0)  # default
grad_y_test_fun = grad(test_fun, argnums=1)

# df/dx = 2x + y; f'(0, 1) = 1
print('df/dx (0, 1) = {}'.format(grad_x_test_fun(jnp.float32(0), jnp.float32(1))))
# d^2f/dx^2 = 2; f'(0, 1) = 2
print('d^2 f/dx^2 (0, 1) = {}'.format(grad(grad_x_test_fun)
    (jnp.float32(0), jnp.float32(1))))
print('d^2 f/dx^2 (3, 3) = {}'.format(grad(grad_x_test_fun)
    (jnp.float32(3), jnp.float32(3))))

# df/dy = x + 2y; f'(0, 1) = 2
print('df/dy (0, 1) = {}'.format(grad_y_test_fun(jnp.float32(0), jnp.float32(1))))

def g(x, y):
  """g=3*df/dx = 3*(2x+y)."""
  grad_x = grad_x_test_fun(x, y)
  return 3 * grad_x

print(grad(g)(jnp.float32(0), jnp.float32(1)))  # prints 6, Default is w.r.t. first arg
```

vmap: Auto-vectorization

$vmap(some\_function, in\_axes = (None, 0))$

Vectorizes the operations inside the function:

- ✓ runs faster, as the operations are "vectorized"
- ✓ useful for gradients

Most common use is for handling minibatches:

1. Write your function for **one** sample;
2. Wrap your function f with:

```
batched_f = vmap(f, in_axes=(None, 0), out_axes=0)
```

The argument in_axes:

- specifies over which axes the function's arguments should be parallelized
- it's a tuple of length equal to the number of the function's arguments if there are more than one, otherwise it's one integer
- *e.g.* $(0, 1, None)$: parallelize over 0–*th* and 1–*st* dimension of first and second argument, resp; and don't parallelize over third argument.

The argument out_axes is analogous to in_axes but refers to axes of the function's output to parallelize over. As loss functions map to $\mathbb{R}$ it is often 0.

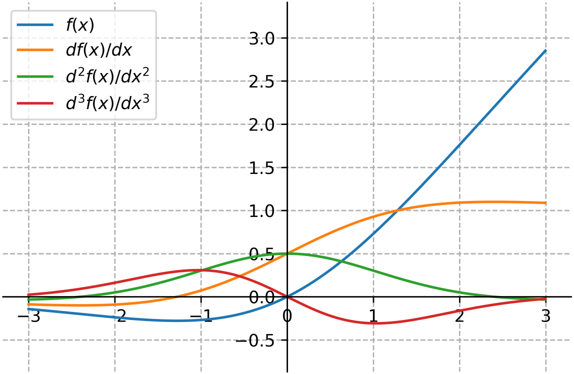
Example for *jit*, *grad* & *vmap*

The Swish Non–linearity [Ramachandran et al., 2017]

```
import jax.numpy as jnp
from jax import grad, jit, vmap

@jit
def sigmoid(x):
  """Computes Sigmoid."""
  return 1 / (1 + jnp.exp(-x))

@jit
def swish(x):
  """Computes the Swish non-linearity: ‘x * sigmoid(x)‘.
  Source: "Searching for Activation Functions" (Ramachandran et al. 2017)
  https://arxiv.org/abs/1710.05941
  """
  return x * sigmoid(x)

batch_swish = vmap(swish, in_axes=0, out_axes=0)
first_derv_swish = vmap(grad(swish), in_axes=0, out_axes=0)
second_derv_swish = vmap(grad(grad(swish)), in_axes=0, out_axes=0)
third_derv_swish = vmap(grad(grad(grad(swish))), in_axes=0, out_axes=0)

x = jnp.arange(-3, 3, .005)  # (1200,)
dx = first_derv_swish(x)
d2x = second_derv_swish(x)
d3x = third_derv_swish(x)
```

EXAMPLE FOR *jit*, *grad* & *vmap*

THE SWISH NON–LINEARITY [RAMACHANDRAN ET AL., 2017]




```
# ...
plt.plot(x, batch_swish(x), label='$f(x)$')
plt.plot(x, dx, label='$df(x)/dx$')
plt.plot(x, d2x, label='$d^2f(x)/dx^2$')
plt.plot(x, d3x, label='$d^3f(x)/dx^3$')
```

JAX: The Gotchas

In-place updates require function calls

Modifying ***DeviceArray*** throws error:

```
import jax.numpy as jnp
import numpy as np


np_arr = np.zeros((2,2), dtype=np.float32)
jnp_arr = jnp.zeros((2,2), dtype=jnp.float32)

np_arr[0, :] = 1  # ok
jnp_arr[0, :] = 1  # TypeError: '<class 'jax.lax.lax._FilledConstant'>' object does not support item
      assignment.
```

Solution: use ***index***, ***index_update*** & ***index_add*** from ***jax.ops***:

```
from jax.ops import index, index_update, index_add

jnp_arr_updated = index_update(jnp_arr, index[0, :], 1)
jnp_arr_added = index_add(jnp_arr, index[0, :], 1)
print(jnp_arr)  # unchanged, all zeros
print(jnp_arr_updated)  # changed, first row ones
print(jnp_arr_added)  # changed, first row ones
```

Pseudo Random Number Genertors (PRNGs)

Splitting state of PRNG for improved reproducibility

Drawbacks of standard PRNGs (*e.g.* ***NumPy***'s Mersenne Twister PRNG):

- shared among processes/threads $\rightarrow$ reduced reproducibility
- large state size $\rightarrow$ memory (~ 2.5 KB)
- can have initialization issues
- slow in general
- *e.g.* ***NumPy***'s Mersenne Twister PRNG fails modern BigCrush tests (*c.f.* TestU01)

JAX uses a modern Three-fry counter-based PRNG that's **splittable**. The random state is described by two unsigned–***int***32s called **key**.

Pseudo Random Number Genertors (PRNGs)

Splitting state of PRNG for improved reproducibility

```
from jax import random
key = random.PRNGKey(0)   # DeviceArray([0, 0], dtype=uint32)

# don't:
n_1 = random.normal(key, shape=(1,))
n_2 = random.normal(key, shape=(1,))   # same as n_1 !!!

# do:
new_key, subkey = random.split(key)
n1 = random.normal(subkey, shape=(1,))
new_key_2, subkey_2 = random.split(new_key)
n2 = random.normal(subkey_2, shape=(1,))  # good: n2 != n1

# or:
key, *subkeys = random.split(key, 5)
for subkey in subkeys:
  print(random.normal(subkey, shape=(1,))[0])
  # or: print(random.randint(subkey, shape=(1,), minval=0, maxval=10)[0])


# or:
key = random.PRNGKey(10)
for _ in range(5):
  key, subkey = random.split(key)
  n = random.randint(subkey, shape=(1,), minval=0, maxval=10)[0]
  print(n)
```

Control Flow

jit when branching requires specifying *static_argnums*, *grad* is OK

It is recommended to use *jit* + *static_argnums* only if the argument that is specified with it rarely changes.

Current structured control flow primitives (allow for *jit*–ing functions that have conditioning and/or loops):

- *lax.cond*, will be differentiable soon;
- *lax.while_loop*, non-differentiable;
- *lax.fori_loop*, non-differentiable;
- *lax.scan*, differentiable.

Print IN *jit*–ED FUNCTION

WE CANNOT PRINT THINGS IN COMPILED CODE (OPERANDS ARE ABSTRACT TYPES)

Example[3]:

```
from jax import jit

@jit
def cube(x):
  print('[cube] input: {}'.format(x))
  y = x**3
  print('[cube] result: {}'.format(y))
  return y

print('2^3={}'.format(cube(2)))
print('3^3={}'.format(cube(3)))
```

Outputs:

```
[cube] input: Traced<ShapedArray(int32[], weak_type=True):JaxprTrace(level=-1/1)>
[cube] result: Traced<ShapedArray(int32[]):JaxprTrace(level=-1/1)>
2^3=8
3^3=27
```

[3] For more information see this explanation.

MNIST Classification using JAX and PyTorch [Paszke et al., 2017]

See full code on GitHub

Based on this jax tutorial

MNIST EXAMPLE
JAX

Using MLP with 2 hidden layers, and 10 output layers (10 digits/-classes):

```
layer_sizes = [784, 512, 512, 10]
step_size = 0.0001
num_epochs = 8
batch_size = 128
n_targets = 10
```

Initialize the parameters of the MLP:

```
def random_layer_params(m, n, key, scale=1e-2, zero_bias=False):
  """Helper function: randomly initialzie weights and biases of
  a dense NN layer. """
  w_key, b_key = random.split(key)
  _w = scale * random.normal(w_key, (n, m))
  _b = np.zeros((n,)) if zero_bias else scale * random.normal(b_key, (n,))
  return _w, _b


def init_network_params(sizes, key, zero_bias=False):
  """Initialize MLP with sizes `sizes`. """
  keys = random.split(key, len(sizes))
  return [random_layer_params(m, n, k, zero_bias=zero_bias)
          for m, n, k in zip(sizes[:-1], sizes[1:], keys)]
```

Other helper functions:

```
@jit
def predict(params, image):
  """ Per-sample predictions. """
  activations = image
  for w, b in params[:-1]:
    outputs = np.dot(w, activations) + b
    activations = relu(outputs)

  final_w, final_b = params[-1]
  logits = np.dot(final_w, activations) + final_b
  return logits - logsumexp(logits)  # for numerical stability

# Make a batched version of the ‘predict‘ function
batched_predict = vmap(predict, in_axes=(None, 0))

@jit
def accuracy(params, images, targets):
  target_class = np.argmax(targets, axis=1)
  predicted_class = np.argmax(batched_predict(params, images), axis=1)
  return np.mean(predicted_class == target_class)
```

Other helper functions:

```
@jit
def relu(x):
  """ ReLU non-linearity. """
  return np.maximum(0, x)

@jit
def loss(params, images, targets):
  preds = batched_predict(params, images)
  return -np.sum(preds * targets)

@jit
def update(params, x, y):
  """ Updates params using loss, inputs x, and targets y."""
  grads = grad(loss)(params, x, y)
  return [(w - step_size * dw, b - step_size * db)
          for (w, b), (dw, db) in zip(params, grads)]
```

Training function:

```
def train_mnist_jax(params, training_generator,
                    tr_images, tr_labels, te_images, te_labels,
                    num_epochs=10, n_targets=10):
  for epoch in range(num_epochs):
    start_time = time.time()
    for x, y in training_generator:  # x: bsize x 784
      y = one_hot(y, n_targets)  # y: bsize x 10
      params = update(params, x, y)
    epoch_time = time.time() - start_time

    train_acc = accuracy(params, tr_images, tr_labels)
    test_acc = accuracy(params, te_images, te_labels)
    print("Epoch {} in {:0.2f} sec".format(epoch, epoch_time))
    print("Training set accuracy {}".format(train_acc))
    print("Test set accuracy {}".format(test_acc))
  return params
```

MNIST EXAMPLE
JAX

Training:

```
params = init_network_params(layer_sizes, random.PRNGKey(0))
params = train_mnist_jax(params, training_generator,
                         train_images, train_labels,
                         test_images, test_labels,
                         num_epochs, n_targets)
```

Output:

```
Epoch 0 in 5.99 sec
Training set accuracy 0.9594333171844482
Test set accuracy 0.9560999870300293
Epoch 1 in 2.97 sec
Training set accuracy 0.97843337059021
Test set accuracy 0.9702000021934509
Epoch 2 in 2.98 sec
Training set accuracy 0.9874666929244995
Test set accuracy 0.976699948310852
Epoch 3 in 2.93 sec
Training set accuracy 0.9911167025566101
Test set accuracy 0.9785999655723572
Epoch 4 in 2.94 sec
Training set accuracy 0.9934166669845581
Test set accuracy 0.9788999557495117
Epoch 5 in 2.97 sec
Training set accuracy 0.9956166744232178
Test set accuracy 0.9799000024795532
Epoch 6 in 3.15 sec
Training set accuracy 0.996399998664856
Test set accuracy 0.9799999594688416
Epoch 7 in 2.99 sec
Training set accuracy 0.9974666833877563
Test set accuracy 0.9810999631881714
```

MNIST EXAMPLE
PYTORCH

Imports & helper functions to convert data:

```
import numpy as onp
import torch
import torch.nn as nn
# jax -> pytorch
torch.from_jax = lambda x: torch.from_numpy(onp.asarray(x))
# pytorch -> jax
np.astensor =  lambda x: np.asarray(x.numpy())
```

Loss function:

```
def criterion(x, y):
  return - torch.sum(x * y)
```

MNIST EXAMPLE
PYTORCH

Class MLP:

```
class MLP(nn.Module):
  def __init__(self, layer_sizes, non_lin=nn.ReLU):
    super().__init__()
    layers = []
    for i in range(1, len(layer_sizes)-1):
      layers.append(nn.Linear(layer_sizes[i-1], layer_sizes[i]))
      layers.append(non_lin())
    layers.append(nn.Linear(layer_sizes[-2], layer_sizes[-1]))
    self.main = nn.Sequential(*layers)
    self.n_param_layers = len(list(self.main.named_parameters())) // 2

  def forward(self, x):
    x = self.main(x)  # bsize x n_labels
    y = torch.logsumexp(x, dim=1)
    return x.sub(y.view(-1, 1))

  def cp_params(self, cp_params):
    if type(cp_params) is not list:
      raise TypeError("Expected list. Got {}".format(type(cp_params)))
    if self.n_param_layers != len(cp_params):
      raise ValueError("Expected equal len. Got {} and {}.".format(
          len(self.main.named_parameters()), len(cp_params)))

    _modules = list(self.main.modules())[0]
    with torch.no_grad():
      for i, (w, b) in enumerate(cp_params):
        _modules[i*2].weight.copy_(torch.from_numpy(onp.asarray(w)).float())
        _modules[i*2].bias.copy_(torch.from_numpy(onp.asarray(b)).float())
```

MNIST EXAMPLE
PYTORCH

Training function:

```
def train_mnist_pytorch(net, training_generator,
                        tr_images, tr_labels, te_images, te_labels,
                        num_epochs=10, n_targets=10):
  optimizer = optim.SGD(net.parameters(), lr=step_size, momentum=0)
  device = torch.device("cuda")
  for epoch in range(num_epochs):
    start_time = time.time()
    for x, y in training_generator:  # x: bsize x 784
      labels = one_hot(y, n_targets)  # y: bsize x 10
      labels = torch.from_jax(labels).long().to(device)
      inputs = torch.from_numpy(x).float().to(device)
      optimizer.zero_grad()
      outputs = net(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()
    epoch_time = time.time() - start_time

    train_acc = accuracy_pytorch(net, tr_images, tr_labels)
    test_acc = accuracy_pytorch(net, te_images, te_labels)
    print("Epoch {} in {:0.2f} sec".format(epoch, epoch_time))
    print("Training set accuracy {}".format(train_acc))
    print("Test set accuracy {}".format(test_acc))
```

MNIST EXAMPLE
PYTORCH

Training:

```
import torch.optim as optim

params = init_network_params(layer_sizes, random.PRNGKey(0)) # JAX
net = MLP(layer_sizes).cuda()
net.cp_params(params)

train_mnist_pytorch(net, training_generator,
                    tr_images=_train_images, tr_labels=train_labels,
                    te_images=test_images, te_labels=test_labels,
                    num_epochs=num_epochs, n_targets=n_targets)
```

MNIST EXAMPLE
PyTorch

Output:

```
Epoch 0 in 3.66 sec
Training set accuracy 0.9591833353042603
Test set accuracy 0.9552000164985657
Epoch 1 in 3.38 sec
Training set accuracy 0.9778666496276855
Test set accuracy 0.9711000323295593
Epoch 2 in 3.29 sec
Training set accuracy 0.9874500036239624
Test set accuracy 0.9777000546455383
Epoch 3 in 3.29 sec
Training set accuracy 0.9908833503723145
Test set accuracy 0.9794000387191772
Epoch 4 in 3.29 sec
Training set accuracy 0.9940833449363708
Test set accuracy 0.9789000749588013
Epoch 5 in 3.56 sec
Training set accuracy 0.9948500394821167
Test set accuracy 0.9787000417709351
Epoch 6 in 3.35 sec
Training set accuracy 0.9974166750907898
Test set accuracy 0.9803000688552856
Epoch 7 in 3.29 sec
Training set accuracy 0.9975500106811523
Test set accuracy 0.9809000492095947
```

MNIST EXAMPLE

VERIFICATION: VALUES WE GET WITH JAX & PYTORCH IMPLEMENTATIONS

JAX:

```
loss jax Traced<ConcreteArray(320.86725)>with<JVPTrace(level=2/0)>
loss jax Traced<ConcreteArray(393.53812)>with<JVPTrace(level=2/0)>
loss jax Traced<ConcreteArray(275.22656)>with<JVPTrace(level=2/0)>
loss jax Traced<ConcreteArray(273.5281)>with<JVPTrace(level=2/0)>
loss jax Traced<ConcreteArray(230.26147)>with<JVPTrace(level=2/0)>
loss jax Traced<ConcreteArray(169.82643)>with<JVPTrace(level=2/0)>
loss jax Traced<ConcreteArray(124.452896)>with<JVPTrace(level=2/0)>
```

PyTorch:

```
loss pytorch  320.8672180175781
loss pytorch  393.5380859375
loss pytorch  275.2265625
loss pytorch  273.528076171875
loss pytorch  230.26145935058594
loss pytorch  169.82643127441406
loss pytorch  124.45288848876953
```

Conclusions

- it's a ***Python*** library for ML that builds on ***NumPy*** that allows for functional programming, with Automatic Differentiation
- main advantage is speed, and can run on CPU/GPU/TPU
- to make use of its advantages, aim to write code with:
    - functions that map inputs $\rightarrow$ outputs, which don't impact global variables
    - compositional functions, with smaller nested functions that can be ***jit***–ed

Some pointers

- Download/Installation & intro
- Read the docs: reference docs & developer docs
- *JAX* tutorial at ***NeurIPS*** '19 @44h26 (video+slides)
- Some cloud colabs
- Colab: Test it out on GPU/TPUs for free: Top menu → ***Runtime*** → ***Change runtime type*** → Choose between ***None***/***GPU***/***TPU***

What's next:

- Libraries for Neural Networks: Stax, Flax, Trax

THANKS.

References I

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. PyTorch. https://github.com/pytorch/pytorch, 2017.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for Activation Functions. arXiv, 2017.