# Python Frameworks for Machine Learning

*Machine Learning Fundamentals for Economists*


## Jesse Perla

*jesse.perla@ubc.ca*

*University of British Columbia*

# Table of contents


- Python
- Python Ecosystem
- JAX Ecosystem

# Python

# Why Python?

- For “modern” ML: **all** the well-supported frameworks are in Python
- In particular, auto-differentiation is central to many ML algorithms
- Why should you avoid Julia/Matlab/R in these cases?
  - → Poor AD, especially for reverse-mode
  - → Network effects. Very few higher level packages for ML pipeline
  - → But Julia dominates for many ML topics (e.g. ODEs) and R is outstanding for classic ML
- Should you use Python for more things?
  - → Maybe, but it is limited and can be slow unless you jump through hoops
  - → Personally, if I have algorithms but no need for AD or particular packages, Julia is a much better language and less frustrating

# There is No Such Thing as “Python”!

- Many incompatible wrappers around C++ for numerical methods
- Numpy/Scipy is the baseline (a common API)
- Pytorch
- JAX
- Ones to avoid
  - → Tensorflow, common in industry but old
  - → Numba (for me, reasonable people disagree)

# Pytorch

- In recent years, the most flexible and popular ML framework for researchers
- Key features:
  - → Most of the code is for auto-differentiation/GPUs
  - → JIT/etc. for GPU and fast kernels for deep learning
  - → Neural Network libraries and utilities
  - → A good subset of numpy
  - → Utilities for ML pipelines optimization/etc.

# Pytorch Key Downsides

- Not really for general purpose programming
  - → Intended for making auto-differentiation of neural networks easy, and updating gradients for solvers
  - → May be very slow for simple things or ones which don't involve high-order AD
- Won't always have packages you need for general code, and compatibility is ugly

# JAX

- Compiler that enables layered program transformations
    1. `jit` compiler to **XLA**, including accelerators (e.g. GPUs)
    2. `grad` Auto-differentiation
    3. `vmap` vectorization
    4. Flexibility to add more transformations
- **JAX PyTrees** provide a nested tree structure for compiler passes
- Closer to being a full JIT for general code than pytorch
- For ML, not full-featured like pytorch. Need to shop for other libraries

# JAX Key Downsides

- JAX is now stable and central to Google DeepMind's infrastructure
  - → Mature enough for production use, though API changes still occur
- Windows support has improved but Linux/macOS remain better supported
- Subset of python. Can't really use loops, etc. Functional-style programming
  - → Much more restrictive than it seems, and far more restrictive than pytorch

# Python Ecosystem

# Environments

- See **Python Environment Setup** for installation instructions and discussion of reproducibility
- `uv` is great as a `pip` replacement, but conda sometimes has better binary support

# Baseline, Safe Packages to Use

- Numpy and Scipy
- Pandas for dataframes
- Matplotlib for general plotting
- Seaborn for plotting data
- Statsmodels for classic econometrics
- Scikit-learn for classic ML

# General Tools for ML Pipelines

- Logging/visualization: **Weights and Biases**
  - → Sign up for an account! Built in hyperparameter optimization tools
- CLI useful for many pipelines and HPO. See **here**
- For more end-to-end frameworks for deep-learning
  - → **Keras** is a higher-level framework for deep learning. Traditionally tensorflow, but now many.
  - → **Pytorch Lightning** is easy and flexible, eliminating a lot of boilerplate for CLI, optimizers, GPUs, etc.
  - → Also **FastAI**
- **HuggingFace** is a great resource for NLP and transformers
- **Optuna** is a great hyperparameter optimization framework, etc.

# JAX Ecosystem

# Examples of Core Transformations

## From JAX quickstart

## Builtin composable transformations: `jit`, `grad` and `vmap`

```
1 import jax
2 import jax.numpy as jnp
3 import numpy as np
4 from jax import grad, jit, vmap, random
```

# Compiling with `jit`

```
def selu(x, alpha=1.67, lmbda=1.05):
  return lmbda * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
key = random.key(0)
x = random.normal(key, (1000000,))
%timeit selu(x).block_until_ready()
selu_jit = jit(selu)
%timeit selu_jit(x).block_until_ready()
```

```
1.23 ms ± 7.03 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
670 µs ± 1.3 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

# Convenience Decorators for `jit`

- Convenience python decorator `@jit`

```
@jit
def selu(x, alpha=1.67, lmbda=1.05):
  return lmbda * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
%timeit selu(x).block_until_ready()
```

```
670 µs ± 1.04 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

# Differentiation with `grad`

```
def sum_logistic(x):
  return jnp.sum(1.0 / (1.0 + jnp.exp(-x)))

derivative_fn = grad(sum_logistic)
x_small = jnp.array([1.0, 2.0, 3.0])
print(derivative_fn(x_small))
```

```
[0.19661197 0.10499357 0.04517666]
```

# Manual “Batching”/Vectorization

## Common to run the same function along one dimension of an array

```
mat = random.normal(key, (150, 100))
batched_x = random.normal(key, (10, 100))

def f(v):
  return jnp.dot(mat, v)
def naively_batched_f(v_batched):
  return jnp.stack([f(v) for v in v_batched])
%timeit naively_batched_f(batched_x).block_until_ready()
```

```
638 µs ± 1.54 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

# Using vmap

## The vmap applies across a dimension

```
@jit
def vmap_batched_f(v_batched):
  return vmap(f)(v_batched)

print('Auto-vectorized with vmap')
%timeit vmap_batched_f(batched_x).block_until_ready()
```

```
Auto-vectorized with vmap
33.1 µs ± 231 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

# More `vmap`

## Can fix dimensions with `in_axes`

```
1 def f(a, x, y):
2   return a * x + y
3 a = 2.0
4 x = jnp.arange(5.)
5 y = jnp.arange(5.)
6 vmap(f, in_axes=(None, 0, 0))(a, x, y)
```

```
Array([ 0.,  3.,  6.,  9., 12.], dtype=float32)
```

# Save `vmap` functions

## Can fix dimensions with `in_axes`

```
@jax.jit
def f(a, x, y):
  return a * x + y
a = 2.0
x = jnp.arange(5.)
y = jnp.arange(5.)
f_batched = vmap(f, in_axes=(None, 0, 0))
f_batched(a, x, y)
```

```
Array([ 0.,  3.,  6.,  9., 12.], dtype=float32)
```

# Key JAX Neural Network Libraries/Frameworks

- Neural Network Libraries
  - → **Flax NNX**
    - ↳ NNX is the new Flax API, Linen the older one
    - ↳ Has momentum, supported by google (for now)
  - → **Equinox**
    - ↳ General, not just neural networks. Similar to NNX
  - → **Keras** supports JAX (as well as PyTorch, TF, etc.)

# Other ML-oriented Packages

- Tough to keep up, see Awesome JAX
- Optax for ML-style optimization
- Checkpointing and serialization: Orbax

# More Scientific Computing in JAX

- jax.scipy which is a subset of scipy
- Nonlinear Systems/Least Squares: Optimistix
- Linear Systems of Equations: Lineax
- Matrix-free operators for iterative solvers: COLA
- Differential Equations: diffrax
- More general optimization and solvers: JAXopt
- Interpolation: interpax

# JAX Challenges

- Basically only pure functional programming
  - → No “mutation” of vectors
  - → Loops/conditionals are tough
  - → Rules for what is `jit`able are tricky
- See JAX - The Sharp Bits
- May not be faster on CPUs or for “normal” things
- Debugging

# PyTrees

- JAX uses a generic **tree structure**. Powerful but takes time to understand: Examples from: **here** and **here**

```
f = lambda x, y: jnp.vdot(x, y)
X = jnp.array([[1.0, 2.0],
               [3.0, 4.0]])
y = jnp.array([3.0, 4.0])
print(f(X[0], y))
print(f(X[1], y))

mv = vmap(f, in_axes = (
  0, # broadcast over 1st index of first argument
  None # don't broadcast over anything of second argument
  ), out_axes=0)
print(mv(X, y))
```

```
11.0
25.0
[11. 25.]
```

# PyTree Example 1

## The `in_axes` can match more complicated structures

```
dct = {'a': 0., 'b': jnp.arange(5.)}
def foo(dct, x):
 return dct['a'] + dct['b'] + x
# axes must match shape of the PyTree
x = 1.
out = vmap(foo, in_axes=(
  {'a': None, 'b': 0}, #broadcast over the 'b'
  None # no broadcasting over the "x"
  ))(dct, x)
# example now: {'a': 0, 'b': 0} etc.
print(out)
```

```
[1. 2. 3. 4. 5.]
```

# PyTree Example 2

```
dct = {'a': jnp.array([3.0, 5.0]), 'b': jnp.array([2.0, 4.0])}
def foo2(dct, x):
 return dct['a'] + dct['b'] + x
# axes must match shape of the PyTree
x = 1.
out = vmap(foo2, in_axes=(
  {'a': 0, 'b': 0}, #broadcast over the 'a' and 'b'
  None # no broadcasting over the "x"
  ))(dct, x)
# example now: {'a': 3.0, 'b': 2.0} etc.
print(out)
```

```
[ 6. 10.]
```

# PyTree Example 3

```
dct = {'a': jnp.array([3.0, 5.0]), 'b': jnp.arange(5.)}
def foo3(dct, x):
 return dct['a'][0] * dct['a'][1] + dct['b'] + x
# axes must match shape of the PyTree
out = vmap(foo3, in_axes=(
  {'a': None, 'b': 0}, #broadcast over the 'b'
  None # no broadcasting over the "x"
  ))(dct, x)
# example now: {'a': [3.0, 5.0], 'b': 0} etc.
print(out)
```

```
[16. 17. 18. 19. 20.]
```