# Jax

## aka Just After eXecution

@rohan-insitro

# Presentation outline

- Part 1: Jax intro
- Part 1b: Jax transforms
- Part 2: Machine learning

# What is Jax?

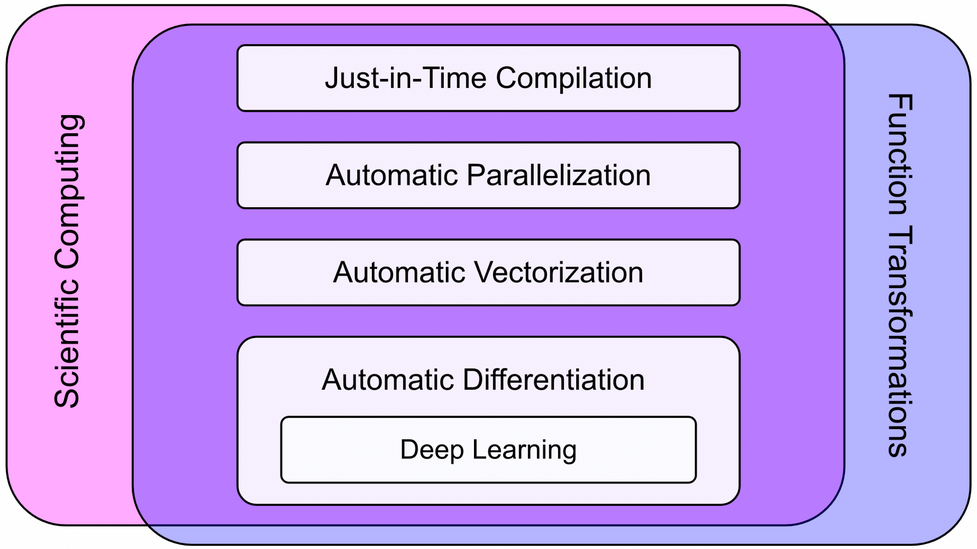
## Numpy on steroids…

- Just an XLA library… that's it!
- Exact same API as Numpy… but with a few extra (useful) features.
- Why so popular in ML?
  - Jax is NOT a machine learning framework!
  - But… the XLA backend enables high-level ML functionalities (+ many other things such as molecular simulation, differential equations, etc.)
  - ML from first principles! Stop abstracting away!

# What's the extra stuff?

## Function transforms!

- Jax provides **four function transforms that can be composed together**. These are the core building blocks on top of the numpy API.
  - jit
  - pmap
  - vmap
  - grad



# Transforms

# vmap

`vmap` is one of the four core functionalities in Jax (next to `pmap`, `grad`, and `jit`).

It allows you to compile a function that was originally intended for scalar inputs and make it accessible for vector inputs. This is useful for things like batch processing in machine learning.

Here is a simple example.

```
import jax
import jax.numpy as jnp

def add_two(input_value):
        return input_value + 2

sample_vector = [1,2,3,4,5,6,7]
add_two_vmap = jax.vmap(add_two)
result = add_two_vmap(jnp.array(sample_vector))
print(result)
```

which prints

```
[3 4 5 6 7 8 9]
```

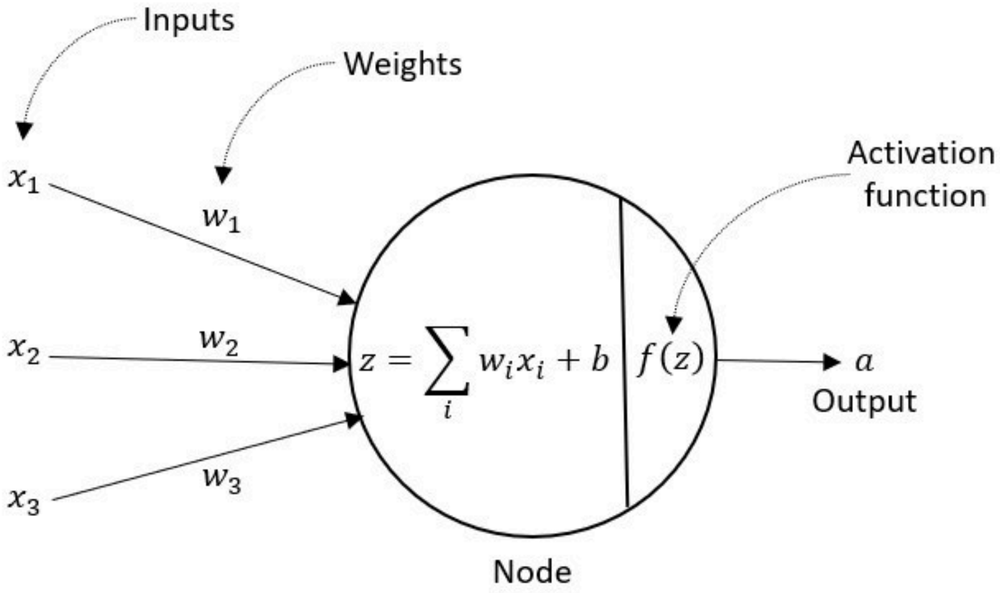
This is a simple example so it would have been easy to create a loop inside the function to handle this case. However, this is inefficient. Jax automatically pre-compiles and vectorizes the function before running it which means we don't need to perform the operation in a loop.

# Machine learning

# Neural networks are just functions!

## OOP vs. functional programming

- Numpy is linear algebra in code. ML is literally just vectors and operations on them. Hence, why we use Numpy for ML. But… we abstract away these primitive concepts via higher-level frameworks such as PyTorch.
- In most modern day deep learning frameworks (PyTorch, TF), we represent neural networks as classes/objects.
- However, neural networks are just **parameterized functions**. $f(x) = \ldots$



## Function

```
def nn(x):
    return forward(x)
```

vs.

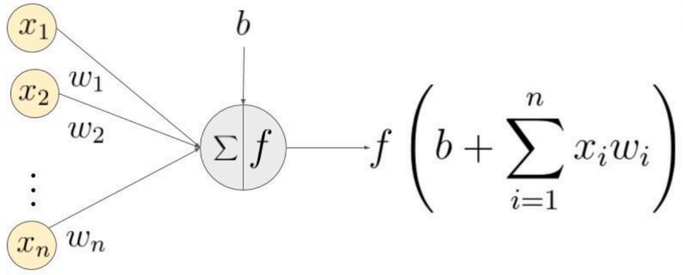
## Function + state (object)

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 10)

    def forward(self, x):
        return F.relu(self.fc1(x))
```

Left code is closer to the mathematical notation!
… if NN's are just functions, why can't we write them As Python functions instead of classes?



# Neural networks + function transforms

- **Jax's unique advantage**: leverage this notion and represent neural networks as functions to be able to apply Jax's main functional transforms:
  **functions + Jax transforms =** 😍.

- But… keeping *params* and *functions* separate becomes messy for modeling.

  - Hence why it is "easier" to program in PyTorch… it abstracts away these concepts! However… less control then!

- Flax vs. Haiku (and why you should use Haiku)

One argument (state kept internally)

```
class MyModule:

  def apply(self, x):
    return self.w * x
```

Two arguments

```
def my_stateless_apply(params, x):
  return params['w'] * x
```