

제발 RL하면 JAX 합시다

I. JAX 란?

II. 이것이 JAX다!!

III. JAX를 내가 해야 하나?

IV. RL에서 JAX가 더욱 강한 이유

V. 저희 속도 말고 다른 생각을 해봅시다

면책조항

이 PPT 의 필자는 딱히 JAX 와 관련해서 Google 과 접점이 있거나, 행사에 참여하거나,
집필에 참여하거나, 회사에서 이걸로 일을 하거나 **하지 않습니다.**

또한 RL 만으로 박사를 받거나, 그걸로 학회를 가보거나, 공신력 있는 논문을 퍼블리시
했거나, 체계적인 교육을 받거나 지도를 받은적도 **없는 사람입니다.**

이 모든 내용은 딱히 공신력도, 증거도, 연구도 없는 **개인적 견해 이니 훌러들어도 됩니다.**

저는 RL을 사랑하는데, 개도 절 사랑했으면 RL을 '짝사랑' 하는 사람이라고 소개하진
않았겠죠. 지금 한국에 있지도 않았을테고.

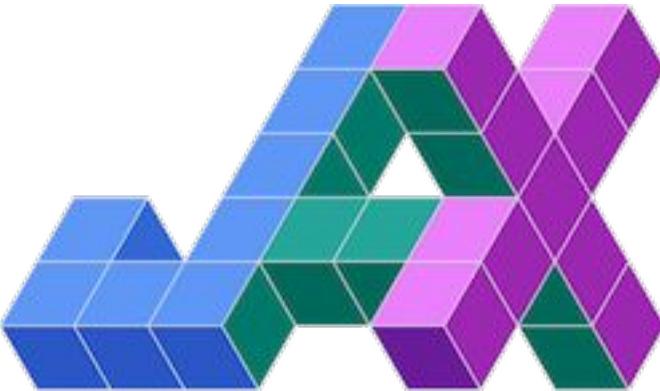
고려대 번역조합

이 PPT 의 필자는 JAX로 4년 넘게 개인프로젝트를 해왔으며, 석사 졸업논문을 JAX로 완료하였고, 어떠한 공동작업이나 논문의 공식구현, LLM 봄에 탑승하지 않고 **순수 JAX 구현**이라는 **표관수련에 가까운 레포들**으로 **총합 116 스타** 를 수집했습니다.

동일하게 RL을 '공부' 한지 7년 넘게 일편단심 이었으며, 교수님과 싸우고 1학기 휴학 하면서도 석사 졸업논문을 RL으로 하였고, 어떠한 공동작업이나 논문의 공식구현, LLM 봄에 탑승하지 않고 **순수 RL 구현**이라는 **표관수련에 가까운 레포들**으로 **총합 116 스타** 를 수집했습니다.

소위 말하는 스타 개발자나 연구자도 아닌데 JAX + RL 이라는 주제로 개인 레포에서 스타를 이만큼 모은거면 인정해 줘야 한다고 생각합니다.

JAX 란?



- **JAX = Numpy + AutoGrad + XLA**
 - Numpy : 수치계산 라이브러리
 - AutoGrad : 모든 함수에 대한 자동 도함수 변환
 - XLA : 가속화된 대수계산 컴파일러(CPU/GPU/TPU)

JAX 란? : Numpy



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

size = int(5e4)

a = np.arange(size).reshape(1, size)
b = np.arange(size).reshape(size, 1)

start = time.time()
c = a * b
end = time.time()
print(f"numpy Time taken: {end - start} seconds")
print(c.shape)

a = jnp.arange(size).reshape(1, size)
b = jnp.arange(size).reshape(size, 1)

start = time.time()
c = a * b
end = time.time()
print(f"jax Time taken: {end - start} seconds")
print(c.shape)
```

```
numpy Time taken: 3.0079097747802734 seconds
(50000, 50000)
jax Time taken: 0.0767209529876709 seconds
(50000, 50000)
```

- **Numpy**
 - 원래의 대부분 numpy 함수들이 그대로 이식되어 있으며, 이를 통해 대부분의 연산을 해 낼수 있고 gpu 같은 하드웨어 가속이 자연스럽게 적용되므로, 강력한 성능을 쉽게 사용할 수 있음
 - JAX 자체에서는 대부분 numpy 연산만이 구현되어 있으며, 뉴럴 네트워크의 구현은 numpy 연산으로 우회적으로 구현 가능

JAX 란? : Numpy

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

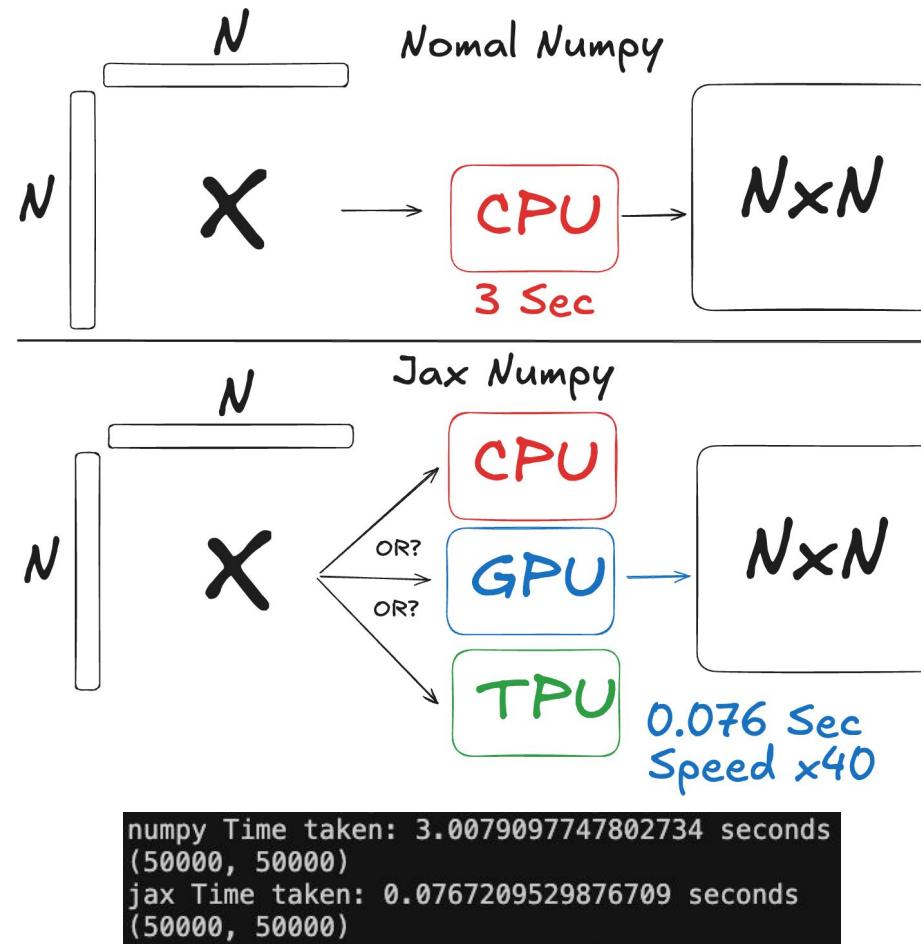
size = int(5e4)

a = np.arange(size).reshape(1, size)
b = np.arange(size).reshape(size, 1)

start = time.time()
c = a * b
end = time.time()
print(f"numpy Time taken: {end - start} seconds")
print(c.shape)

a = jnp.arange(size).reshape(1, size)
b = jnp.arange(size).reshape(size, 1)

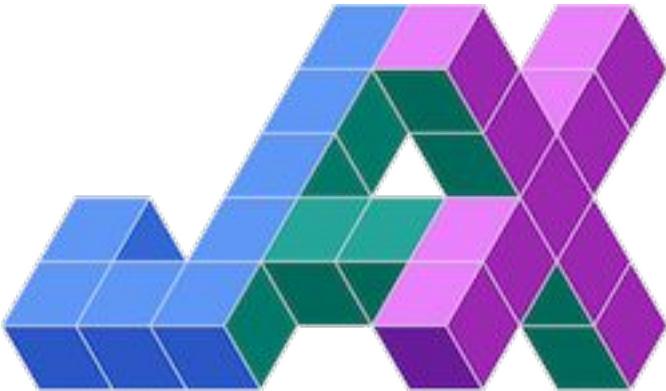
start = time.time()
c = a * b
end = time.time()
print(f"jax Time taken: {end - start} seconds")
print(c.shape)
```



- **Numpy**

- 원래의 대부분 numpy 함수들이 그대로 이식되어 있으며, 이를 통해 대부분의 연산을 해 낼수 있고 gpu 같은 하드웨어 가속이 자연스럽게 적용되므로, 강력한 성능을 쉽게 사용할 수 있음
- JAX 자체에서는 대부분 numpy 연산만이 구현되어 있으며, 뉴럴 네트워크의 구현은 numpy 연산으로 우회적으로 구현 가능

JAX 란? : AutoGrad



```
import jax
import jax.numpy as jnp

def f(x):
    return 5 * x**4 + 3 * x**3 + 2 * x**2 + x + 1

x = jnp.array(1.0)

print(f"f(x): {f(x)}")

print(f"grad(f)(x): 20 * x^3 + 9 * x^2 + 4 * x + 1 = {jax.grad(f)(x)}")

print(f"grad(grad(f))(x): 60 * x^2 + 18 * x + 4 = {jax.grad(jax.grad(f))(x)}")

print(f"grad(grad(grad(f)))(x): 120 * x + 18 = {jax.grad(jax.grad(jax.grad(f)))(x)}")

f(x): 12.0
grad(f)(x): 20 * x^3 + 9 * x^2 + 4 * x + 1 = 34.0
grad(grad(f))(x): 60 * x^2 + 18 * x + 4 = 82.0
grad(grad(grad(f)))(x): 120 * x + 18 = 138.0
```

- **AutoGrad**

- `jax.grad` : 입력된 '함수'의 인수에 대하여 출력의 그레디언트를 반환하는 함수를 출력
- 인수라면 뉴럴넷의 파라미터 또한 가능하므로, 이를 통해 backprop 된 그레디언트 값을 구할수 있고, 이를 통해 뉴럴넷을 학습 가능함
- 그 외에 2차 3차 도함수도 아주 쉽게 변환이 가능하므로, 이러한 도함수들을 이용한 복잡한 구조의 설계 가능

JAX 란? : AutoGrad

```
import jax
import jax.numpy as jnp

def f(x):
    return 5 * x**4 + 3 * x**3 + 2 * x**2 + x + 1

x = jnp.array(1.0)

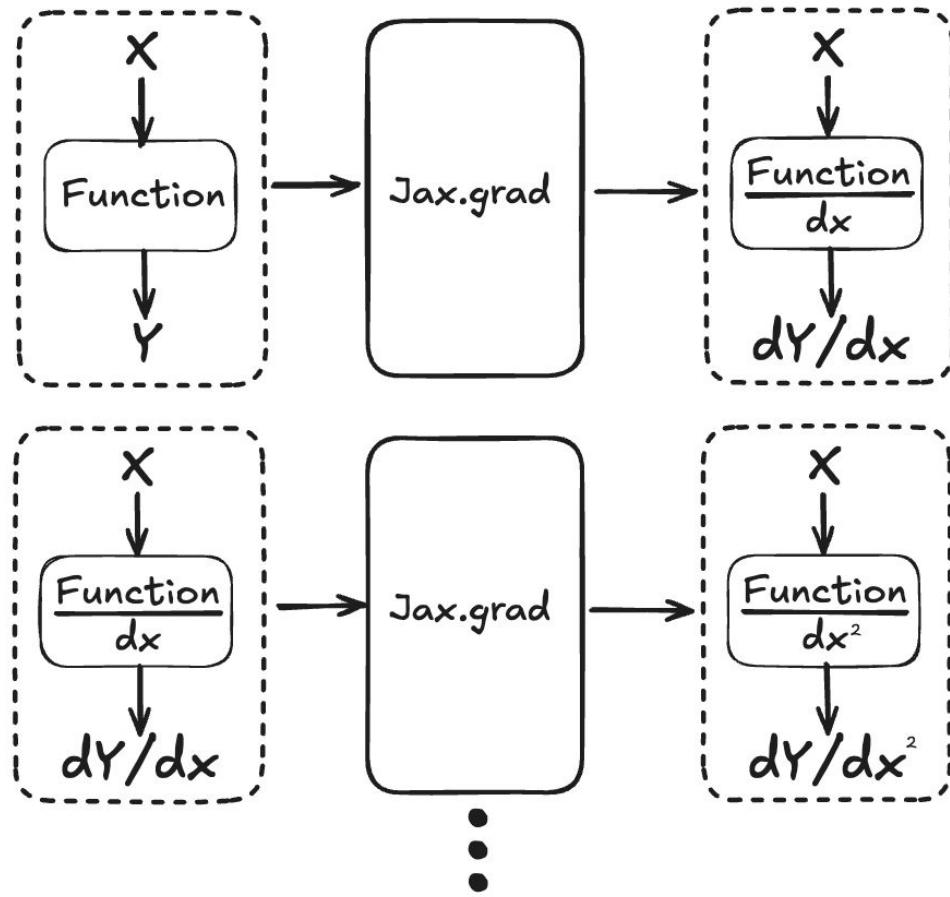
print(f"f(x): {f(x)}")

print(f"grad(f)(x): 20 * x^3 + 9 * x^2 + 4 * x + 1 = {jax.grad(f)(x)}")

print(f"grad(grad(f))(x): 60 * x^2 + 18 * x + 4 = {jax.grad(jax.grad(f))(x)}")

print(f"grad(grad(grad(f)))(x): 120 * x + 18 = {jax.grad(jax.grad(jax.grad(f)))(x)}")

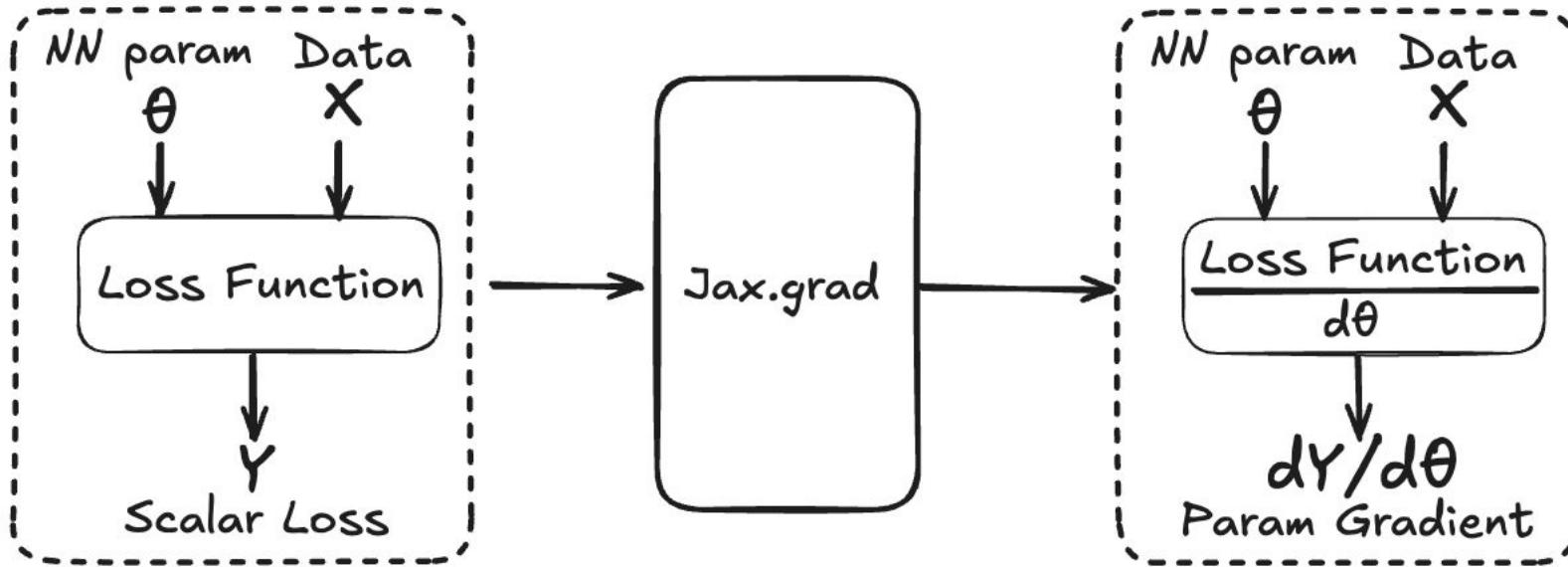
f(x): 12.0
grad(f)(x): 20 * x^3 + 9 * x^2 + 4 * x + 1 = 34.0
grad(grad(f))(x): 60 * x^2 + 18 * x + 4 = 82.0
grad(grad(grad(f)))(x): 120 * x + 18 = 138.0
```



- **AutoGrad**

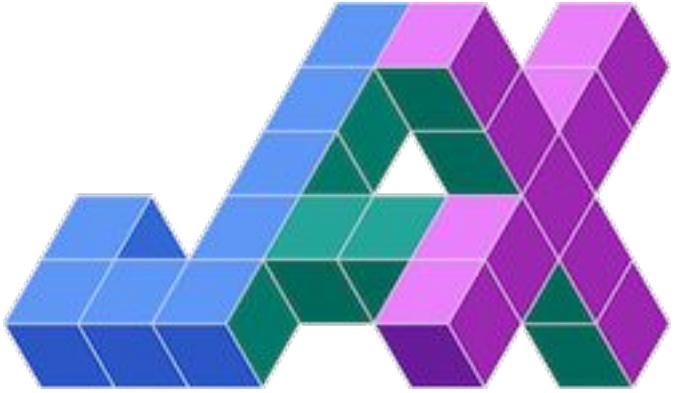
- **jax.grad** : 입력된 '함수'의 인수에 대하여 출력의 그라디언트를 반환하는 함수를 출력
- 인수라면 뉴럴넷의 파라미터 또한 가능하므로, 이를 통해 backprop 된 그라디언트 값을 구할수 있고, 이를 통해 뉴럴넷을 학습 가능함
- 그 외에 2차 3차 도함수도 아주 쉽게 변환이 가능하므로, 이러한 도함수들을 이용한 복잡한 구조의 설계 가능

JAX 란? : AutoGrad



- AutoGrad
 - `jax.grad` : 입력된 '함수'의 인수에 대하여 출력의 그라디언트를 반환하는 함수를 출력
 - 인수라면 뉴럴넷의 파라미터 또한 가능하므로. 이를 통해 backprop 된 그라디언트 값을 구할수 있고. 이를 통해 **뉴럴넷을 학습 가능함**
 - 그 외에 2차 3차 도함수도 아주 쉽게 변환이 가능하므로. 이러한 도함수들을 이용한 복잡한 구조의 설계 가능

JAX 란? : XLA



- **XLA**

- XLA 적용 조건 : 입력과 출력의 형태, 길이, 타입 전부가 변경되지 않는 '함수'로 작성되어야 함
- Just In Time(JIT) Compilation : '함수'를 지정된 입력 형식에 맞춰 컴파일해 실행
- vmap : 동일한 하드웨어 내에서 함수의 병렬화를 진행
- pmap : 동시에 다른 하드웨어들 내에서 함수의 병렬화를 진행 (Multi GPU Support)

```
import time
import jax
import jax.numpy as jnp

def compute_intensive(x):
    # Simulating a complex computation
    for i in range(100):
        x = jnp.sin(x) * jnp.cos(x)
    return x

x = jnp.ones(1000)
_ = jax.jit(compute_intensive)(x[0]) # warmup

start = time.time()
original_result = compute_intensive(x[0])
original_time = time.time() - start

start = time.time()
jit_result = jax.jit(compute_intensive)(x[0])
jit_time = time.time() - start

print(f"Original: {original_result:.5f} in {original_time:.6f}s")
print(f"JIT: {jit_result:.5f} in {jit_time:.6f}s")
print(f"Results match: {jnp.allclose(original_result, jit_result)}")
print(f"Speedup: {original_time/jit_time:.2f}x")

def calculate_for_single_value(x):
    return jnp.sin(x) * jnp.exp(-x**2)

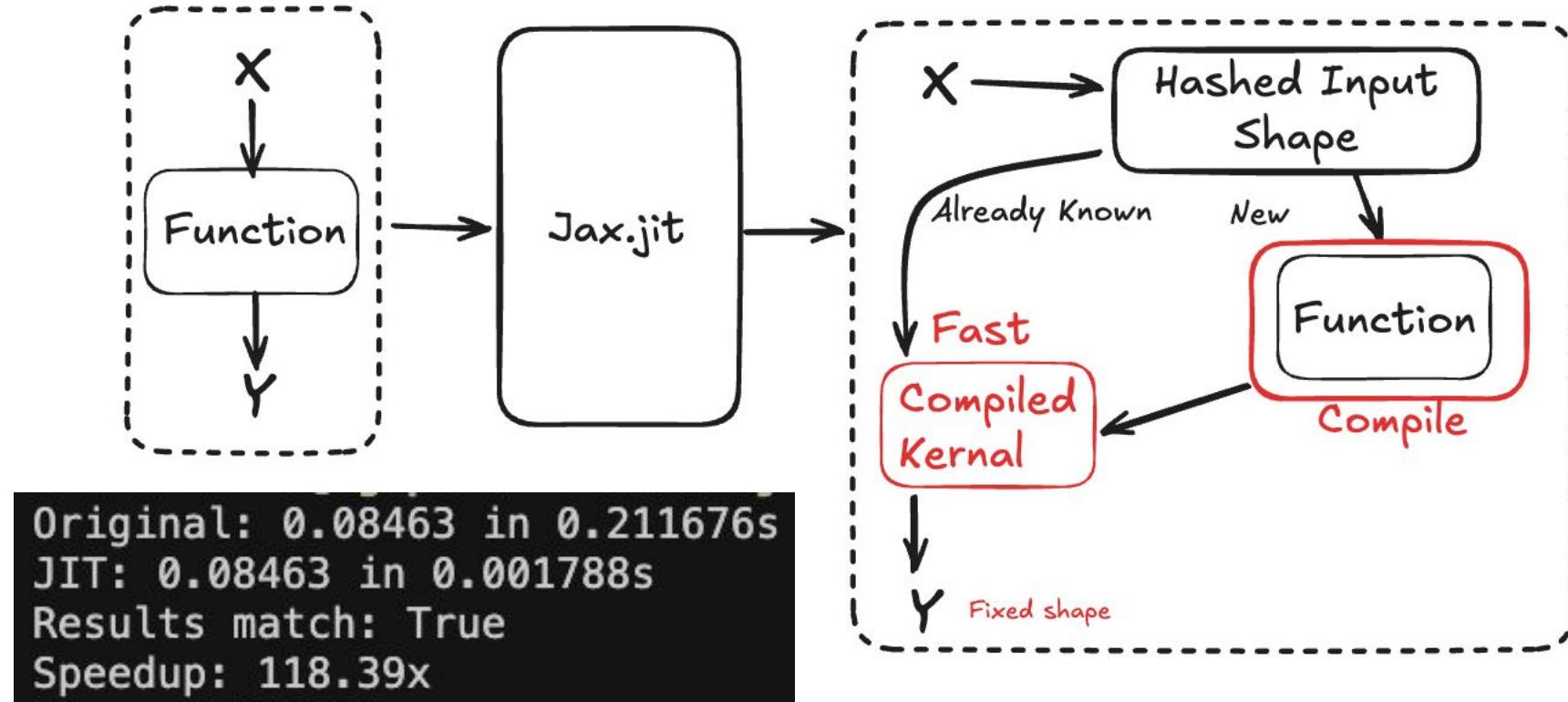
inputs = jnp.linspace(0, 5, 1000)
start = time.time()
result1 = jnp.array([calculate_for_single_value(x_i) for x_i in inputs]) # Slow way
original_time = time.time() - start

_ = jax.vmap(calculate_for_single_value)(inputs) # warmup
start = time.time()
result2 = jax.vmap(calculate_for_single_value)(inputs) # Fast way: vmap
vmap_time = time.time() - start

print(f"Loop: {original_time:.6f}s")
print(f"vmap: {vmap_time:.6f}s")
print(f"Results match: {jnp.allclose(result1, result2)}")
print(f"Speedup: {original_time/vmap_time:.2f}x")
```

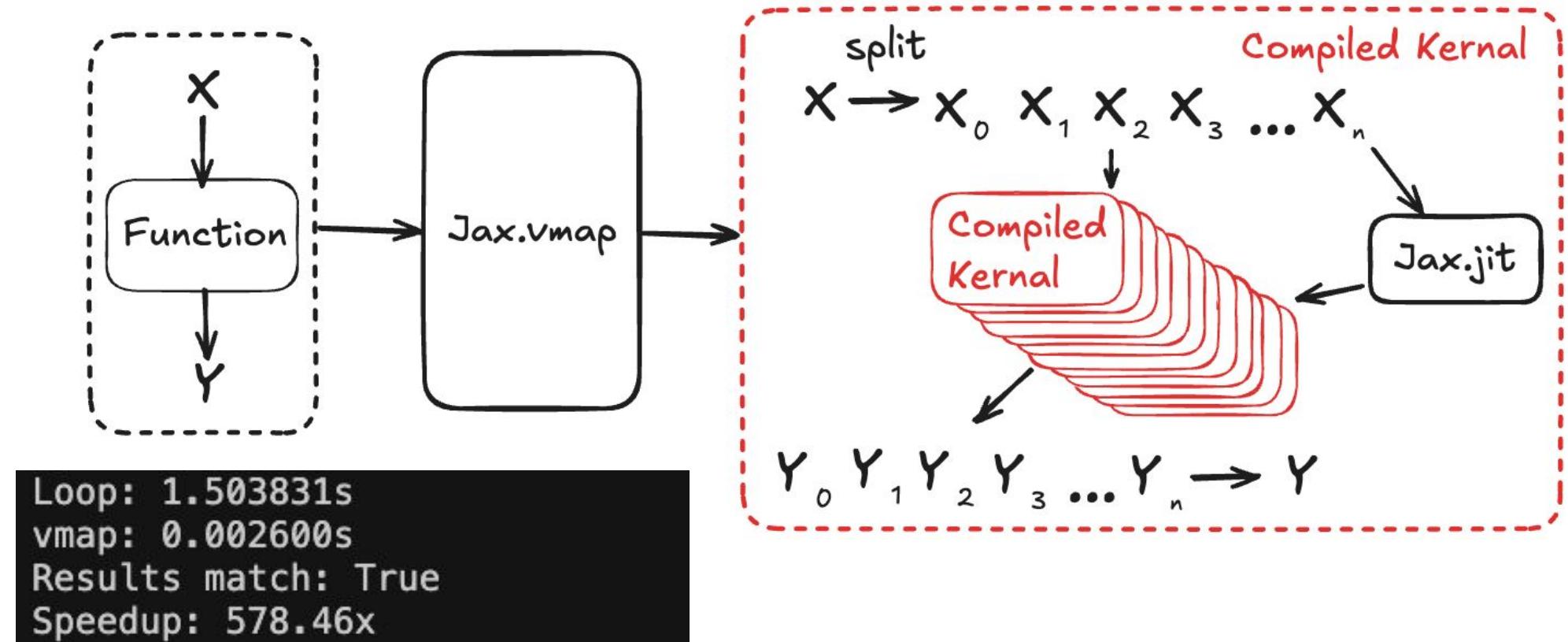
```
Original: 0.08463 in 0.211676s
JIT: 0.08463 in 0.001788s
Results match: True
Speedup: 118.39x
Loop: 1.503831s
vmap: 0.002600s
Results match: True
Speedup: 578.46x
```

JAX 란? : XLA



- XLA
 - XLA 적용 조건 : 입력과 출력의 형태, 길이, 타입 전부가 변경되지 않는 '함수'로 작성되어야 함
 - **Just In Time(JIT) Compilation** : '함수'를 **지정된 입력 형식에 맞춰 컴파일해 실행**
 - vmap : 동일한 하드웨어 내에서 함수의 병렬화를 진행
 - pmap : 동시에 다른 하드웨어들 내에서 함수의 병렬화를 진행 (Multi GPU Support)

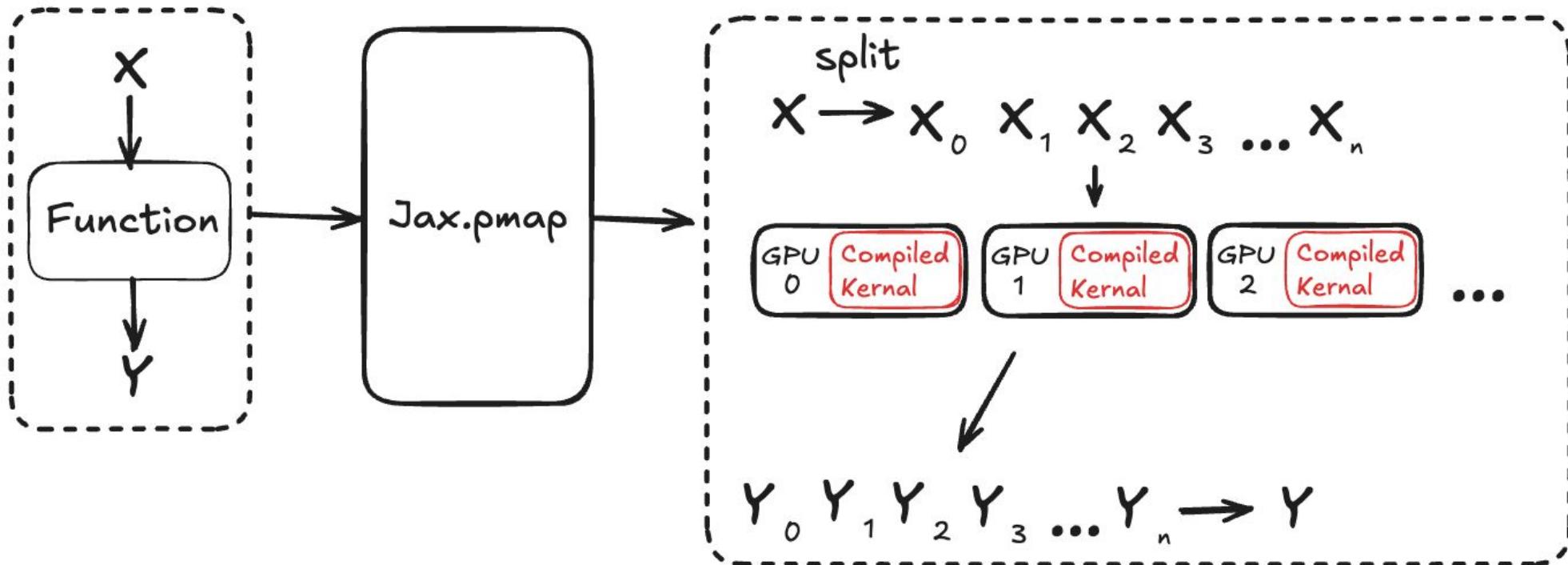
JAX 란? : XLA



- XLA

- XLA 적용 조건 : 입력과 출력의 형태, 길이, 타입 전부가 변경되지 않는 '함수'로 작성되어야 함
- Just In Time(JIT) Compilation : '함수'를 지정된 입력 형식에 맞춰 컴파일해 실행
- **vmap** : 동일한 하드웨어 내에서 함수의 병렬화를 진행
- **pmap** : 동시에 다른 하드웨어들 내에서 함수의 병렬화를 진행 (Multi GPU Support)

JAX 란? : XLA



- XLA

- XLA 적용 조건 : 입력과 출력의 형태, 길이, 타입 전부가 변경되지 않는 '함수'로 작성되어야 함
- Just In Time(JIT) Compilation : '함수'를 지정된 입력 형식에 맞춰 컴파일해 실행
- `vmap` : 동일한 하드웨어 내에서 함수의 병렬화를 진행
- `pmap` : 동시에 다른 하드웨어들 내에서 함수의 병렬화를 진행 (Multi GPU Support)

JAX 런? : Flax, Haiku, Equinox, Optax

```
import jax
import jax.numpy as jnp
import flax.linen as nn

class MyModel(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Dense(5)(x)
        x = nn.Dense(5)(x)
        x = nn.Dense(1)(x)
        return x

model = MyModel()
params = model.init(jax.random.PRNGKey(0), jnp.ones((1, 5)))
output = model.apply(params, jnp.ones((1, 5)))

print(params)
print(output)
```



Fla
x

- 순수 JAX는 위에서 설명한 Numpy, AutoGrad, XLA 외 3가지만 구현되어 있으므로, ML 연구(뉴럴넷)을 위해서는 아래와 같은 라이브러리를 사용하여 뉴럴넷을 구현하고 학습해야함
 - Flax : 구글 브레인의 메인스트림 뉴럴넷 구현 라이브러리
 - Haiku : Deepmind의 메인스트림 뉴럴넷 구현 라이브러리(였던 것). 딥마인드가 구글 브레인과 합쳐지며 버려짐...
 - Equinox : 어느 개인이 관리중인 뉴럴넷 구현 라이브러리. 하지만 Flax의 대안으로 떠오르고 있음. 인기 많음
 - Optax : JAX의 유니버설한 Optimizer 구현. Flax는 Haiku는 Equinox는 Optax를 통해 학습됨

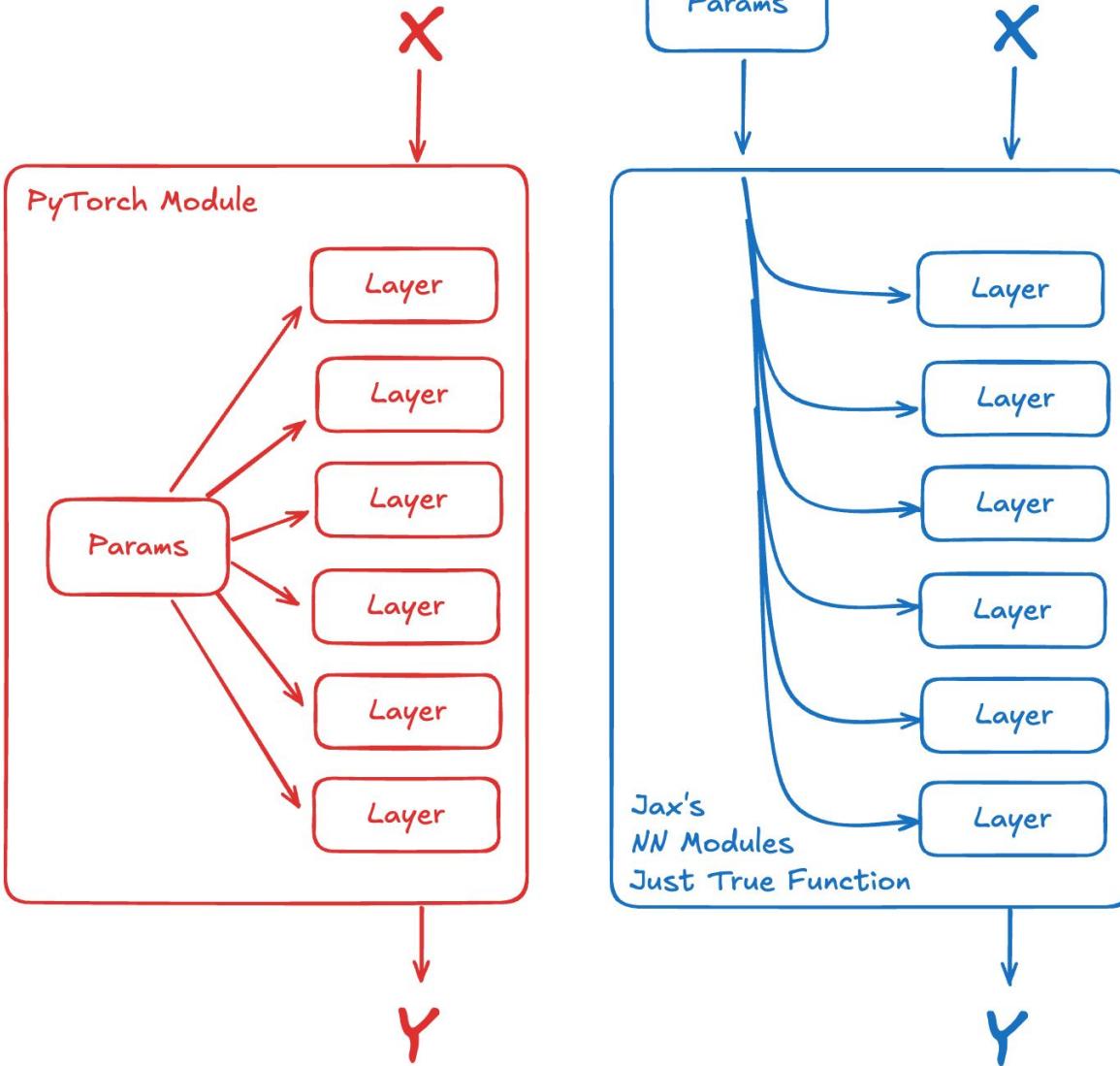
JAX 랑? : Flax, Haiku, Equinox, Optax

```
import jax
import jax.numpy as jnp
import flax.linen as nn

class MyModel(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Dense(5)(x)
        x = nn.Dense(5)(x)
        x = nn.Dense(1)(x)
        return x

model = MyModel()
params = model.init(jax.random.PRNGKey(0), jnp.ones((1, 5)))
output = model.apply(params, jnp.ones((1, 5)))

print(params)
print(output)
```



- JAX로 작성되는 코드는 모두 순수 "함수" 형태를 띠어야 하므로, NN module은 Function의 역할을 함
- NN의 Param들은 그 함수의 인수로써 입력되는 형태로 작성됨
- 그 외엔 생각 외로 PyTorch랑 크게 다르진 않다

이것이 JAX 대!!: 희망편

- 장점
 - XLA 기능을 기반으로 한 압도적인 최적화와 속도
 - 여러 하드웨어 종류(CPU, GPU, TPU)에서 단순하고 유니버설하게 동작하고 최적화된 코드
 - 함수형 프로그래밍을 기반으로 한 유연한 모듈 변경
 - 짜다 보면 역으로 OOP 보다 훨씬 직관적 일수도?

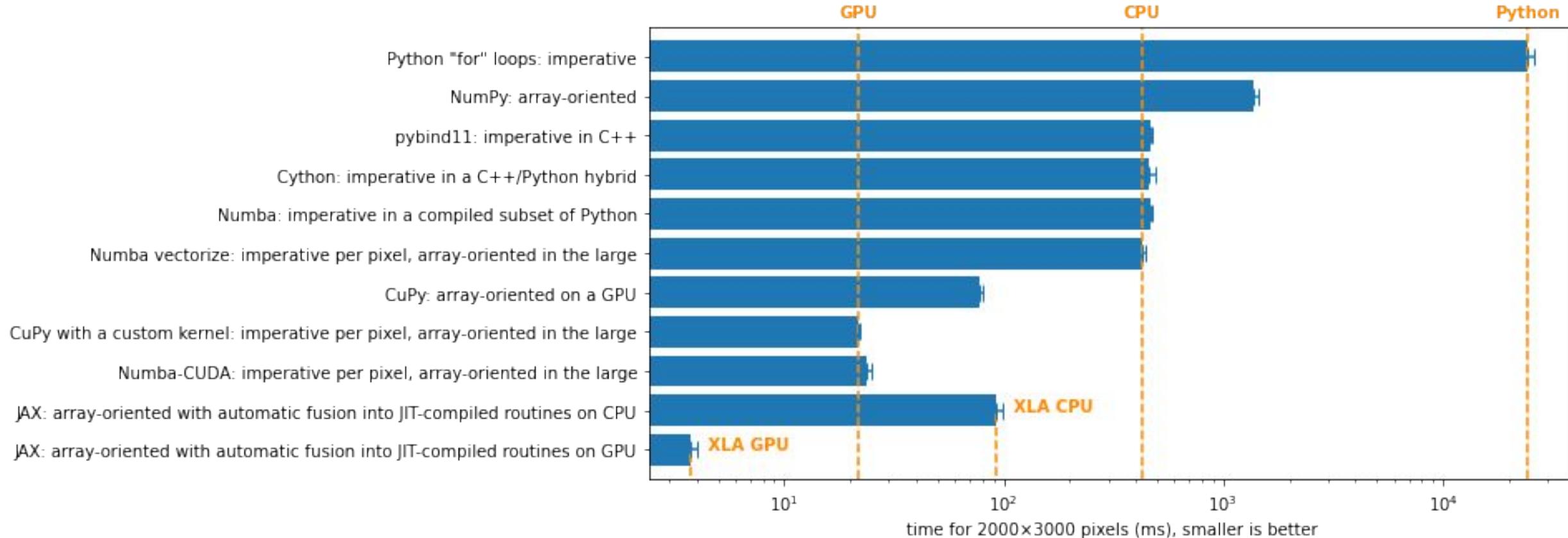


이것이 JAX 대!!: 파멸편

- 단점
 - 함수의 입출력 종류 제한이 엄격해 복잡한 구조의 프로그램의 경우 JIT 이 되는 구조를 만들기 어려움
 - Python 을 주로 개발해온 ML 개발자 들에게 함수형 프로그램은 많이 낫 설 수 있음
 - List, Set, Dictionary 등의 동적 형태 데이터구조 사용 X – 당신에겐 고정 크기 np.array 와 tuple 들이 있을 뿐
 - 디버그가 어려움



이것이 JAX 대! 성능편



- 그럼에도 불구하고 말 그대로 강력하다
 - XLA 는 다른 수치연산 '컴파일러' 를 과도 비교해도 상당히 강력하고
 - 충분히 뉴럴 네트워크를 구현할 수 있지만, 그것만 구현할 수 있는 것도 아니다
 - Numpy + AutoGrad + XLA 오직 그것 만으로도 정말 많은 것이 가능하다

이것이 JAX 대! 성능편

Benchmark times

We report the runtimes for all models on an NVIDIA RTX3090 GPU with 24-core CPU. The times are averaged over two runs, which commonly have a standard deviation of <5%.

- 다양한 네트워크와 예시에서 학습 시간 측정 비교
- 일반적인 모델 학습에서도 기본 2배정도의 학습 속도 향상
- Pytorch 는 이미 PyTorch Lightning 을 기반으로 최적화 된 상태
- Vision Transformer 에서는 1.1배정도의 적은 속도 상승(아마 Torch Vision 지원으로 인한 최적화가 원인 인듯)

Tutorial 5: Inception, ResNet and DenseNet

Models	PyTorch	JAX
GoogleNet	53min 50sec	16min 10sec
ResNet	20min 47sec	7min 51sec
Pre-Activation ResNet	20min 57sec	8min 25sec
DenseNet	49min 23sec	20min 1sec

평균 : 3배 학습 속도

Tutorial 6: Transformers and Multi-Head Attention

Models	PyTorch	JAX
Reverse Sequence	0min 26sec	0min 7sec
Anomaly Detection	16min 34sec	3min 45sec

평균 : 4배 학습 속도

Tutorial 9: Deep Autoencoders

Models	PyTorch	JAX
AE - 64 latents	13min 10sec	7min 10sec
AE - 128 latents	13min 11sec	7min 10sec
AE - 256 latents	13min 11sec	7min 11sec
AE - 384 latents	13min 12sec	7min 14sec

평균 : 2배 학습 속도

Tutorial 11: Normalizing Flows

Models	PyTorch	JAX
MNIST Flow - Simple	2hrs 37min 29sec	1hrs 17min 59sec
MNIST Flow - VarDeq	3hrs 25min 10sec	1hrs 36min 56sec
MNIST Flow - Multiscale	2hrs 17min 10sec	57min 57sec

평균 : 2배 학습 속도

Tutorial 15: Vision Transformer

Models	PyTorch	JAX
Vision Transformer	28min 40sec	27min 10sec

극적인 학습 속도 상승 X

이것이 JAX 대!: 성능편

Whisper JAX

This repository contains optimised JAX code for OpenAI's [Whisper Model](#), largely built on the 😊 Hugging Face Transformers Whisper implementation. Compared to OpenAI's PyTorch code, Whisper JAX runs over **70x** faster, making it the fastest Whisper implementation available.

- OpenAI 의 Whisper 모델을 JAX 로 최적화한

**파이프라인으로 인퍼런스 했을때, 기존의 Pytorch 구현과
비교해 최대 70배가 넘는 속도로 실행할 수 있음**

- Python 위에서 동작하는 프로그램. 동시에 학습용으로
작성된 프로그램을 C++ 나 여타 최적화된 Cuda 프로그램.
또는 TensorRT 에 가깝게 또는 그 이상으로 최적화 할 수
있다는 것은 매우 매력적인 일

Benchmarks

We compare Whisper JAX to the official [OpenAI implementation](#) and the 😊 [Transformers implementation](#). We benchmark the models on audio samples of increasing length and report the average inference time in seconds over 10 repeat runs. For all three systems, we pass a pre-loaded audio file to the model and measure the time for the forward pass. Leaving the task of loading the audio file to the systems adds an equal offset to all the benchmark times, so the actual time for loading and transcribing an audio file will be higher than the reported numbers.

OpenAI and Transformers both run in PyTorch on GPU. Whisper JAX runs in JAX on GPU and TPU. OpenAI transcribes the audio sequentially in the order it is spoken. Both Transformers and Whisper JAX use a batching algorithm, where chunks of audio are batched together and transcribed in parallel (see section [Batching](#)).

Table 1: Average inference time in seconds for audio files of increasing length. GPU device is a single A100 40GB GPU. TPU device is a single TPU v4-8.

	OpenAI	Transformers	Whisper JAX	Whisper JAX
Framework	PyTorch	PyTorch	JAX	JAX
Backend	GPU	GPU	GPU	TPU
1 min	13.8	4.54	1.72	0.45
10 min	108.3	20.2	9.38	2.01
1 hour	1001.0	126.1	75.3	13.8

이것이 JAX API: 회사별

First
Party

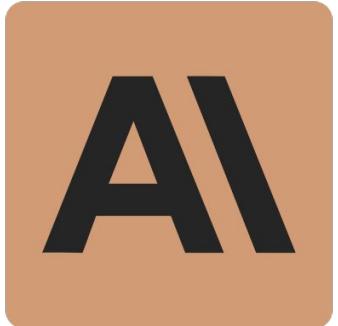


Google

DeepMind

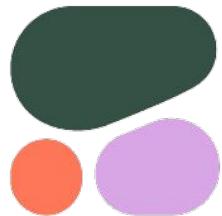
- Gemini ...
- Alpha Fold ...
- All Researches ...

Second
Party



Anthropi

All Claude
models



Cohere



Apple
Intelligence
Engine



xAI
All Grok
models



HuggingFace
JAX
SNAPshots
JAX

JAX ■ 내가 해야 하나? - RL 이 아니면 : 하면 좋다

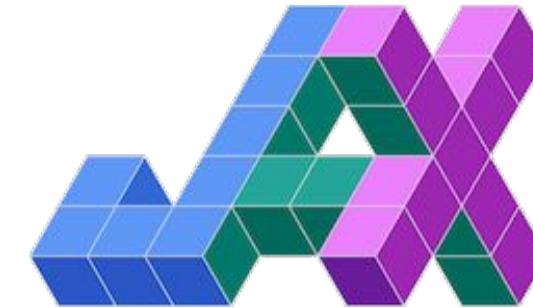


- 그냥 지도학습이나 비지도 학습을 하는거라면 '엄청나게' 큰 매리트 까지는 없음
- PyTorch 는 많은 자료와 추가적인 최적화 라이브러리(Flash Transformer 등등...)도 많으며, 학습 파이프라인의 이미 많이 최적화 되어있음
- 그럼에도 불구하고, JAX 의 기본적인 속도 향상은 상당히 매력적이고 강력함

JAX ■ 내가 해야 하나? - RL 이라면 : 무조건 좋다

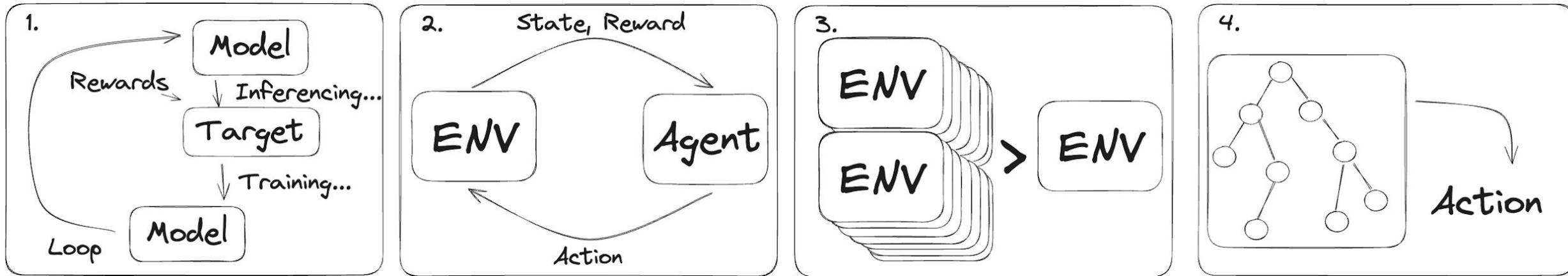
PyTorch

vs



- 하지만 RL 은 주로 구성되는 연산들의 특수성으로 인해 JAX 는 RL 를 매우 강력하게 최적화 할 수 있음!!
- 현재 딥마인드에서 발표되는 대부분의 RL 논문은 JAX 를 사용하고 있으며. 그 외의 RL 연구 커뮤니티에서도 많이 이용되고 있는 추세

RL에서 JAX 가 더욱 강한 이유: RL의 특수성?



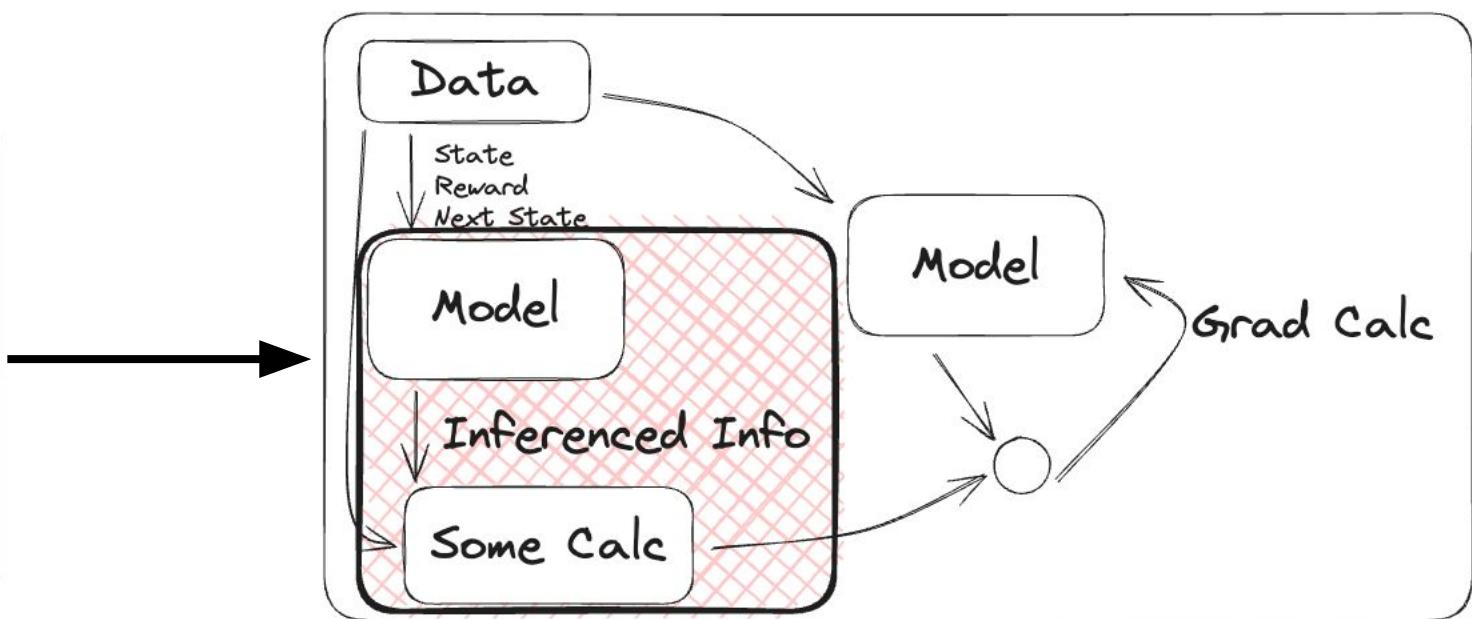
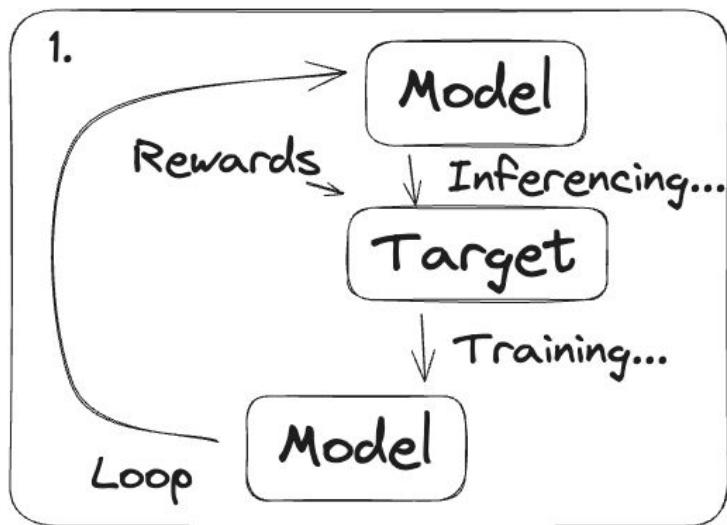
1. 복잡한 학습 파이프라인

2. 환경 탐색 루프

3. 병렬화에 따른 이득

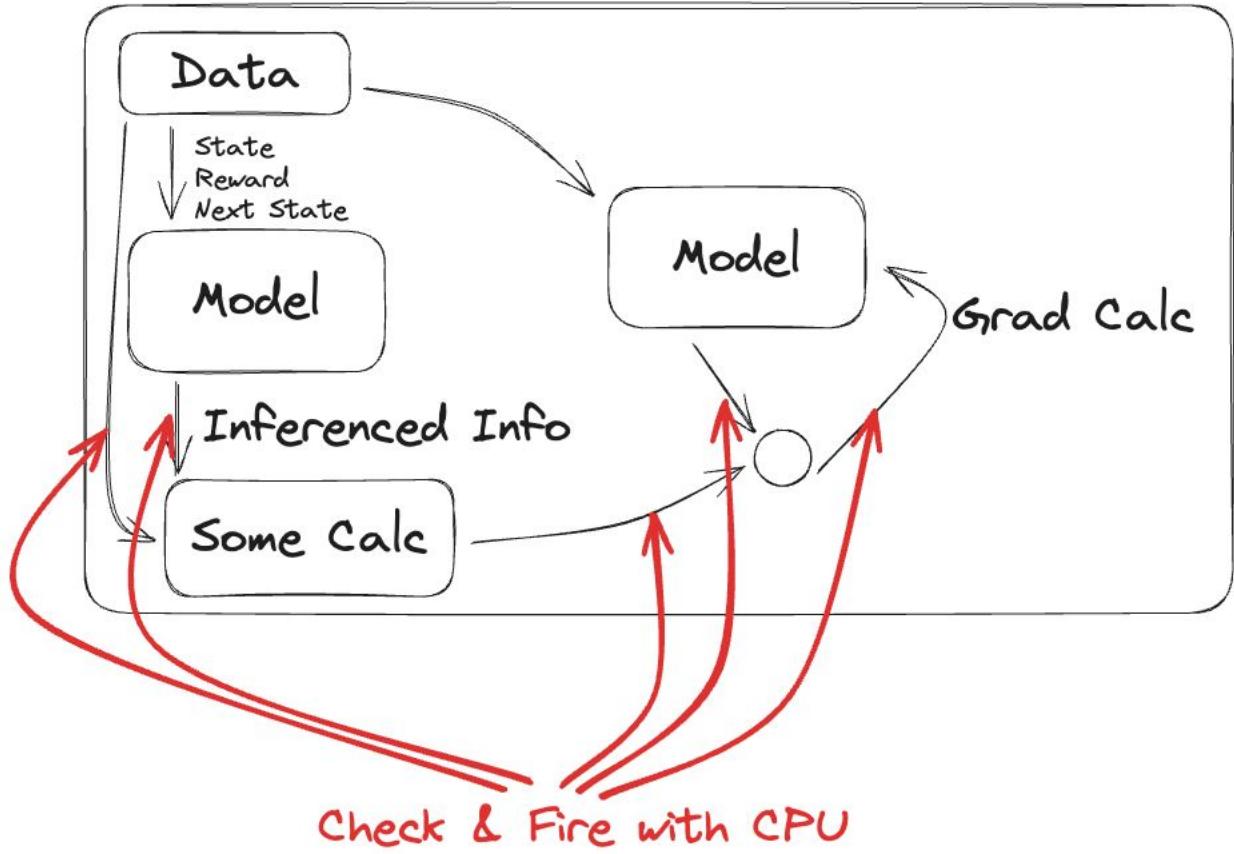
4. 인퍼런스의 복잡성

RL 의 특수성: 복잡한 파이프라인



- RL 의 '학습'에서 가장 중요하고, 오래 걸리는 연산은 사실 grad 를 계산하고 optimize 하는 것이 아님
- 근사함수인 Model 을 인퍼런스 하고, 데이터를 다시 재가공해서 모델이 예측해야 할 값을 재생산 하는 연산이 가장 오래 걸리고 중요
- 이러한 복잡한 파이프라인을 최적화 하는것이 RL 의 '알고리즘' 성능 최적화에서 가장 핵심적

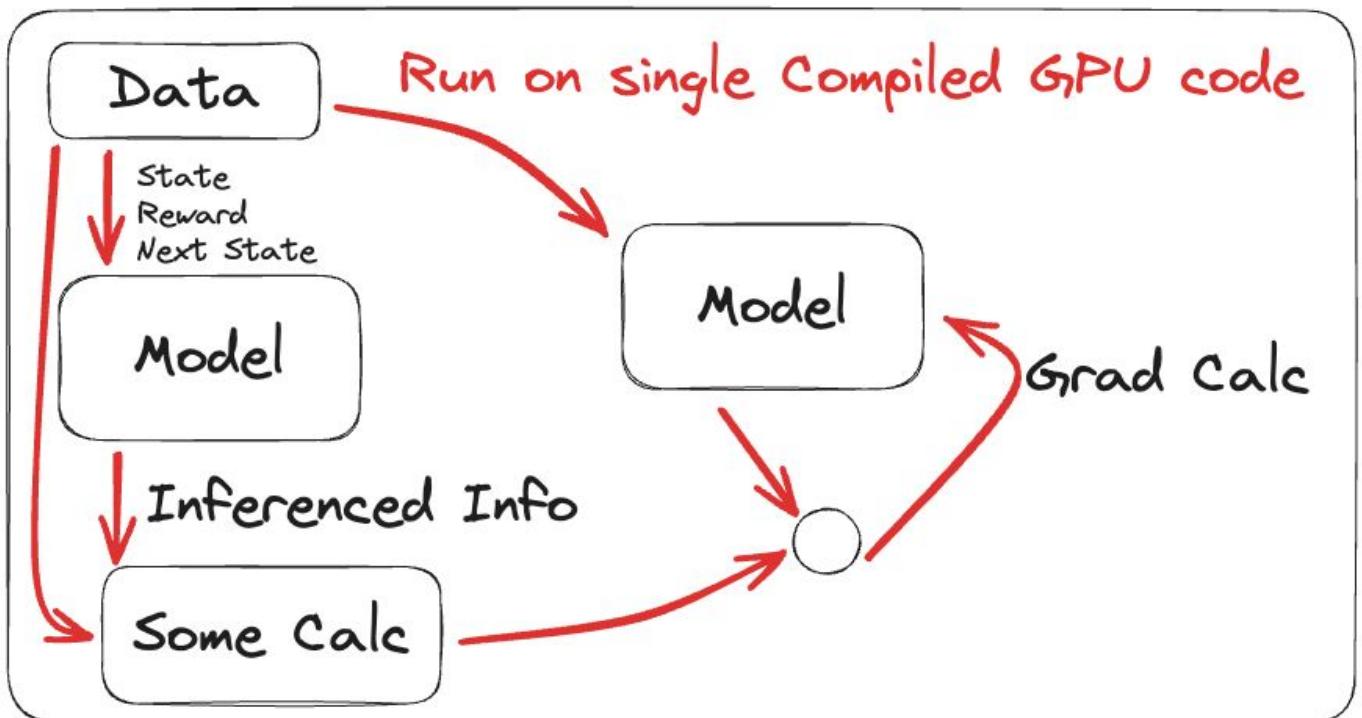
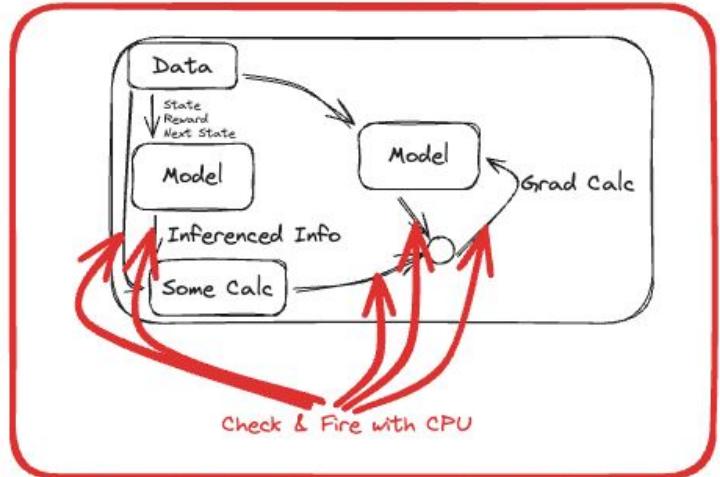
RL 의 특수성: 복잡한 파이프라인



- 물론 Pytorch 에서 저 연산들을 CPU 에서 수행하는 것은 아니지만, 저렇게 많은 파이프라인의 중간중간 연결을 CPU 에서 관리하고 연결하고 실행시키면서 많은 GPU 사용률이 낭비됨
- 매우 무거운 모델을 실행하는 것뿐만 아닌 간단하게 곱하고 더하고 나누는 과정이 전부 python 으로 된 CPU 코드에서 관리되는것.

RL 의 특수성: 복잡한 파이프라인

@JAX.jit



- 하지만 JAX 의 JIT(Just In Time) compilation 을 사용하면 위와 같이 하나의 GPU code 로 변환이 가능하며, 중간 중간 CPU로 관리해가며 생기던 속도 저하를 거의 완전히 배제할 수 있음
- 모델을 실행하고, 라벨과 그래디언트에 따라 최적화만 하면 되는 지도학습이나 비지도학습은 이렇게 최적화 되는 단계가 적지만, RL은 이러한 중간 과정이 매우 많기 때문에, 같은 환경에서 DQN의 torch 구현과 jax 구현의 동작 속도는 매우 큰 차이가 날

RL 의 특수성: 복잡한 파이프라인

Stable Baselines Jax (SBX)

Stable Baselines Jax (SBX) is a proof of concept version of Stable-Baselines3 in Jax.

It provides a minimal number of features compared to SB3 but can be much faster (up to 20x times!): <https://twitter.com/araffin2/status/1590714558628253698>

Stable Baselines Jax 에서는 SB3(pytorch) 와 비교하여, 최대 20배의 학습 속도가 나온다고 '주장'

Jax-Baseline

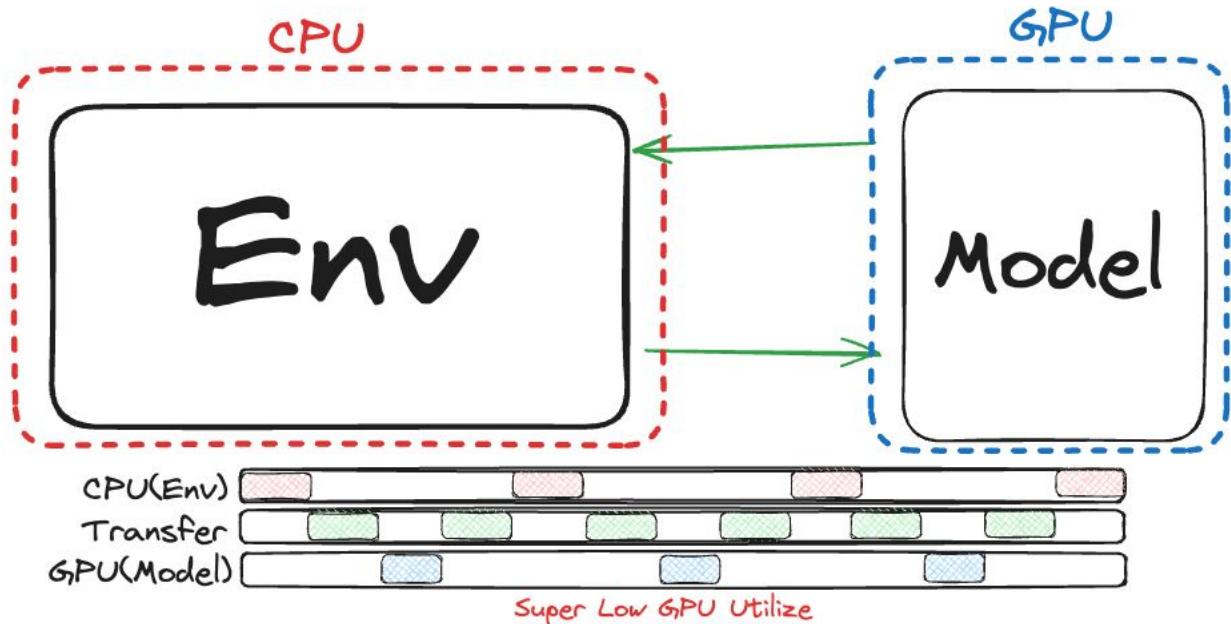
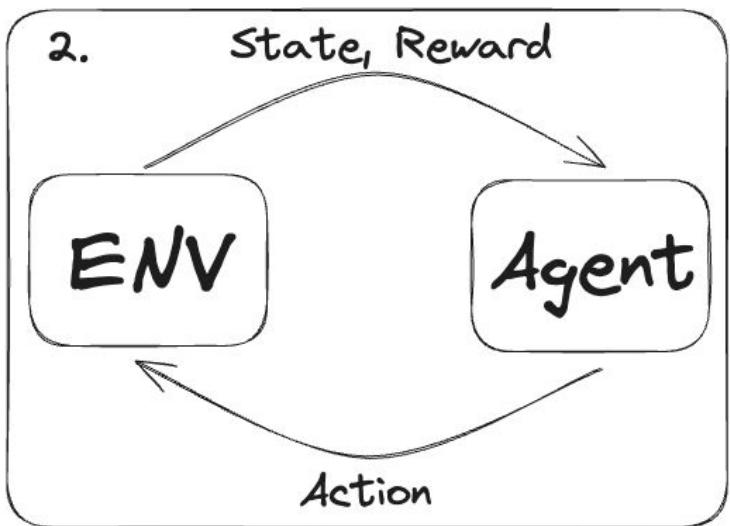
Jax-Baseline is a Reinforcement Learning implementation using JAX and Flax/Haiku libraries, mirroring the functionality of Stable-Baselines.

Features

- 2-3 times faster than previous Torch and Tensorflow implementations
- Optimized using JAX's Just-In-Time (JIT) compilation

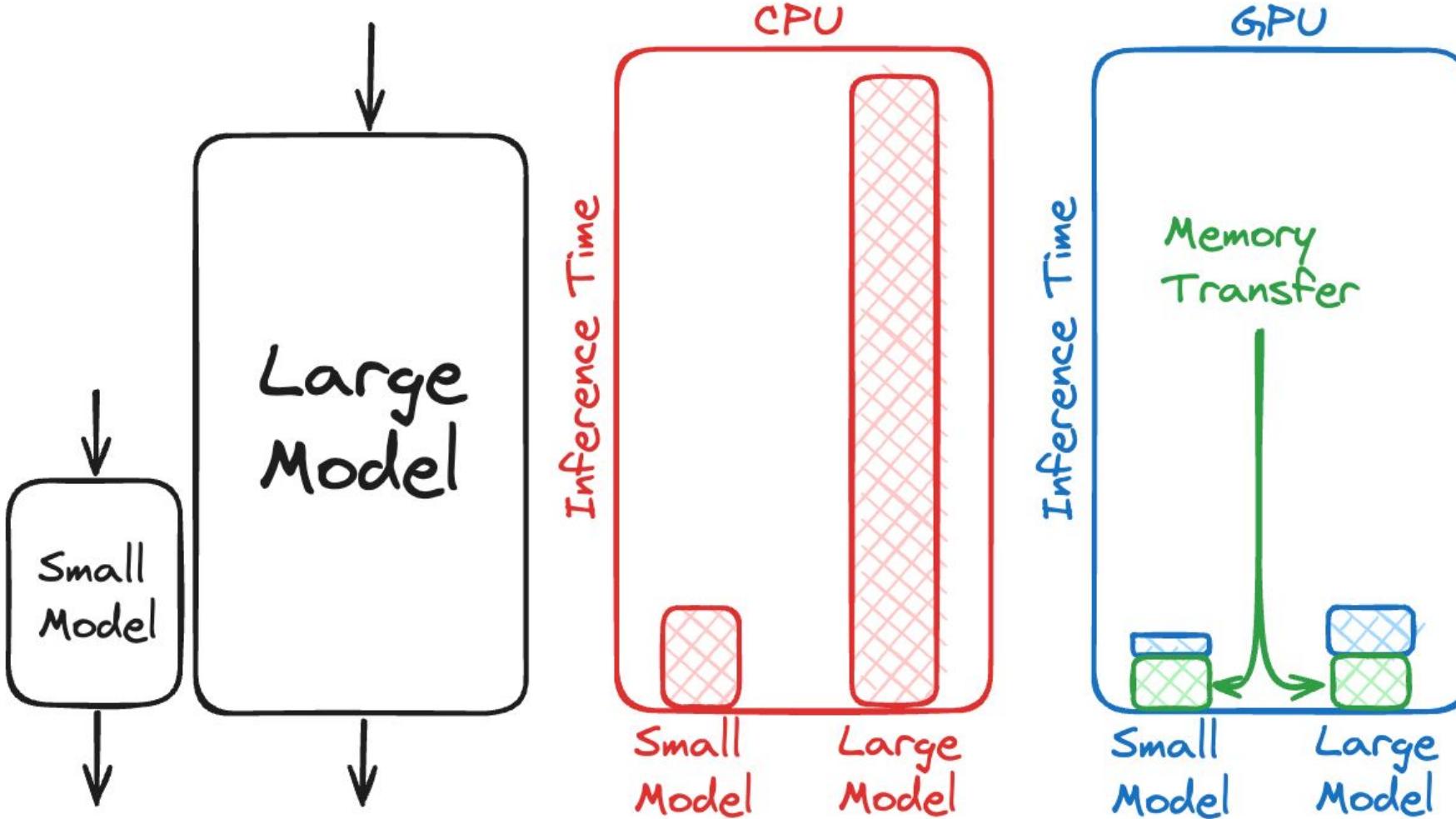
개인적인 구현에서는 같은 CPU 환경을 해결하는 DQN 에서 2 ~ 3 배의 프레임을 달성하였음

RL 의 특수성: 환경 탐색 루프



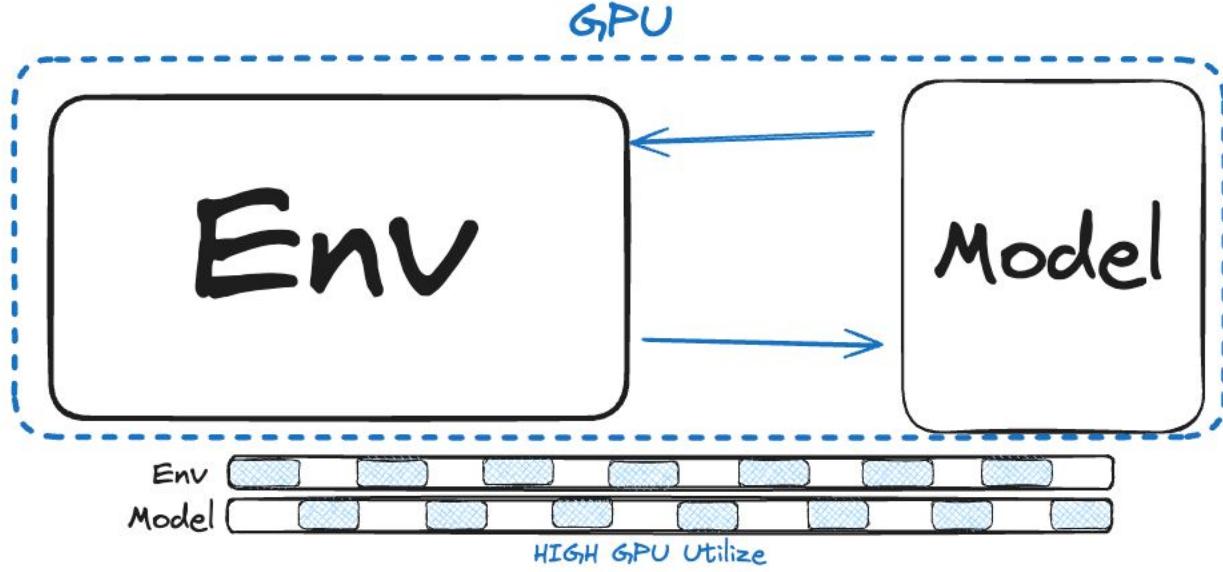
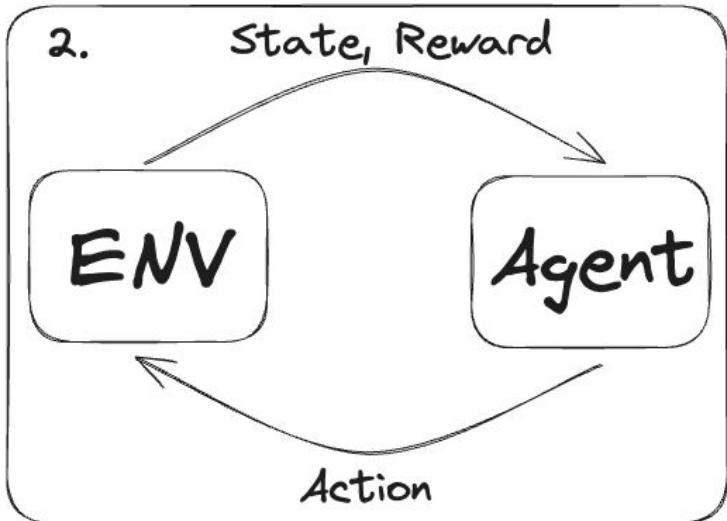
- RL 은 모델을 학습 하는것 뿐만 아닌 '환경' 과 계속해서 상호작용 하며, 변화된 정책에 따른 결과를 수집해 가며 학습이 필요함
- 하지만, Env 는 CPU 에서 작성되었고, 결정을 내리는 Model 은 GPU 에 존재하므로 필연적으로 계속해서 CPU와 GPU 메모리간 통신이 이루어져야 함
- 하지만 state 하나를 model 이 평가하는 데에는 오래 걸리지 않고, 환경이 행동 하나를 진행하는데에도 오래 걸리지 않는다면... 학습 과정중 대부분을 차지 하는건 통신

RL 의 특수성: 환경 탐색 루프



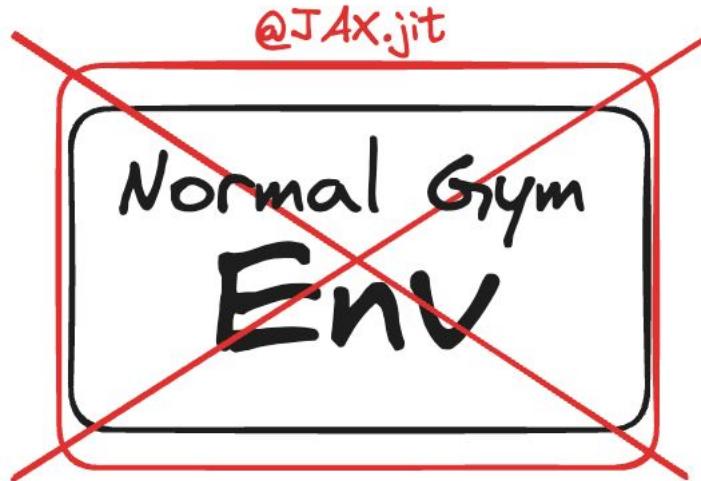
- GPU에서 데이터를 처리하는 것은 CPU보다 훨씬 빠를테지만, 처리 시간의 감소는 완전히 정비례하지 않음
- 위 그림처럼, 데이터를 전송하는데 걸리는 시간이 명백히 존재

RL 의 특수성: 환경 탐색 루프



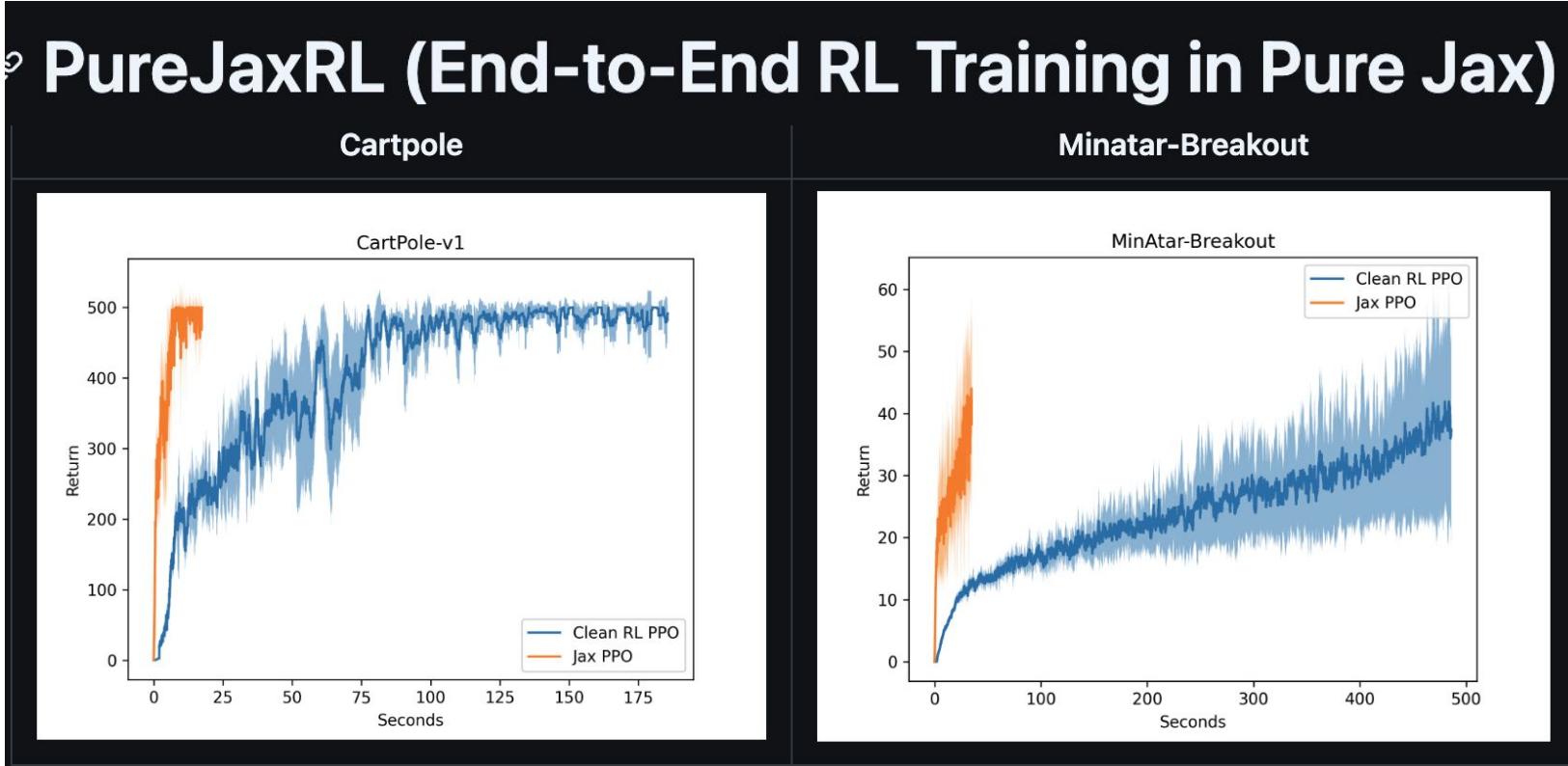
- 그렇다면 환경을 GPU에서 실행한다면?
- GPU 와 CPU 사이에서 발생하는 통신이 필요 없어 매우 높은 GPU 사용률을 달성 가능함
- 또한, 이 환경과의 '통신'과 환경의 동작도 '파이프라인'이므로... jax.jit 으로 모든 학습 루프를 하나의 GPU 코드로써 실행 가능
- 이로서 이전보다 매우 잘 최적화된 학습 루프를 구성할 수 있음

RL 의 특수성: 환경 탐색 루프



- 하지만 매우 치명적인 문제가 존재
- 기존의 Python 이나 그 외. CPU에서 동작하는 환경을 GPU에 들어가도록 만들 수는 없음
- Jax로 순수하게 작성된 환경 코드가 필요하며, JAX의 특수한 문법을 완벽히 맞추면서 매우 복잡한 환경을 작성하는 것은 거의 **고문에 가까움**

RL 의 특수성: 환경 탐색 루프



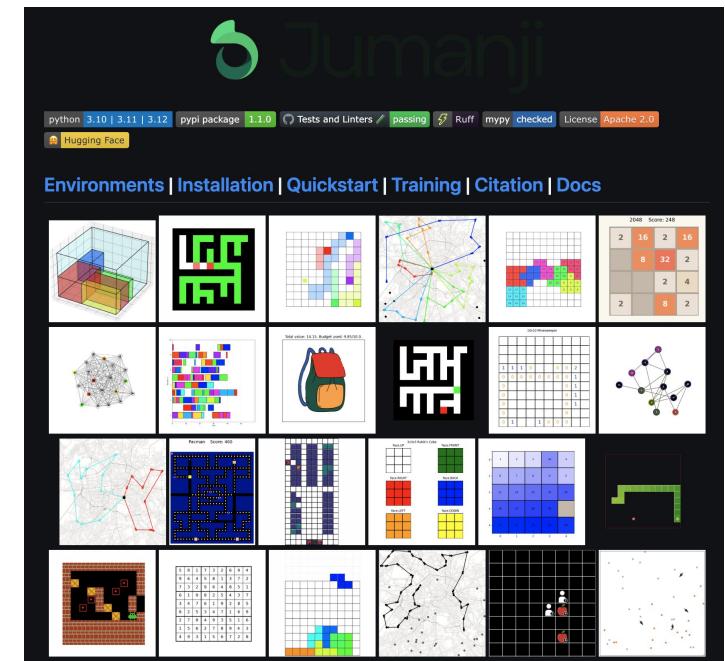
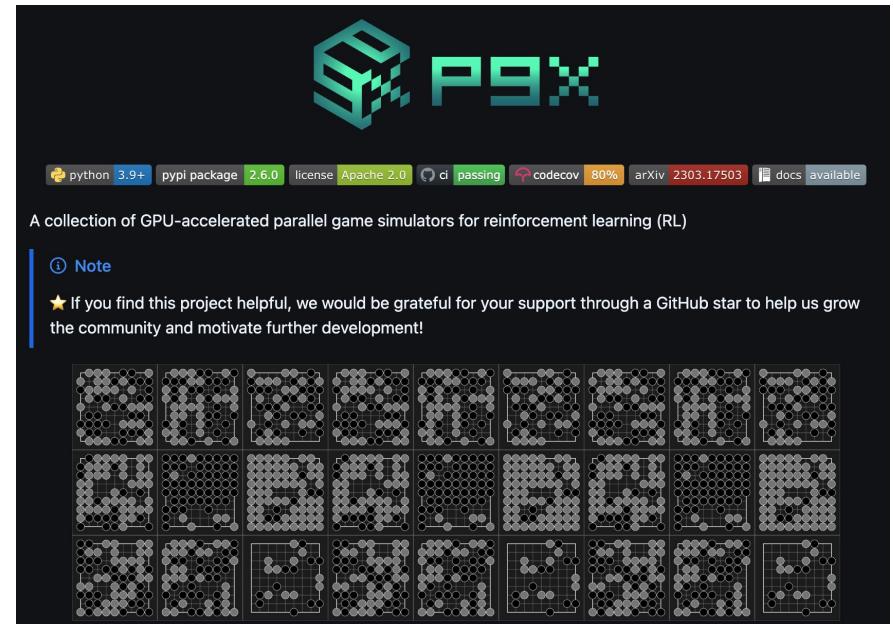
- 그럼에도 불구하고, 인내의 과정은 달다
- GPU에서 모든 환경 탐색 과정을 포함한 학습 루프를 JIT 해 동작시키는 End-to-End RL Training 은 Pytorch로 작성된 Clean RL 과 비교하여 10배 이상의 속도를 보장

RL 의 특수성: 환경 탐색 투표

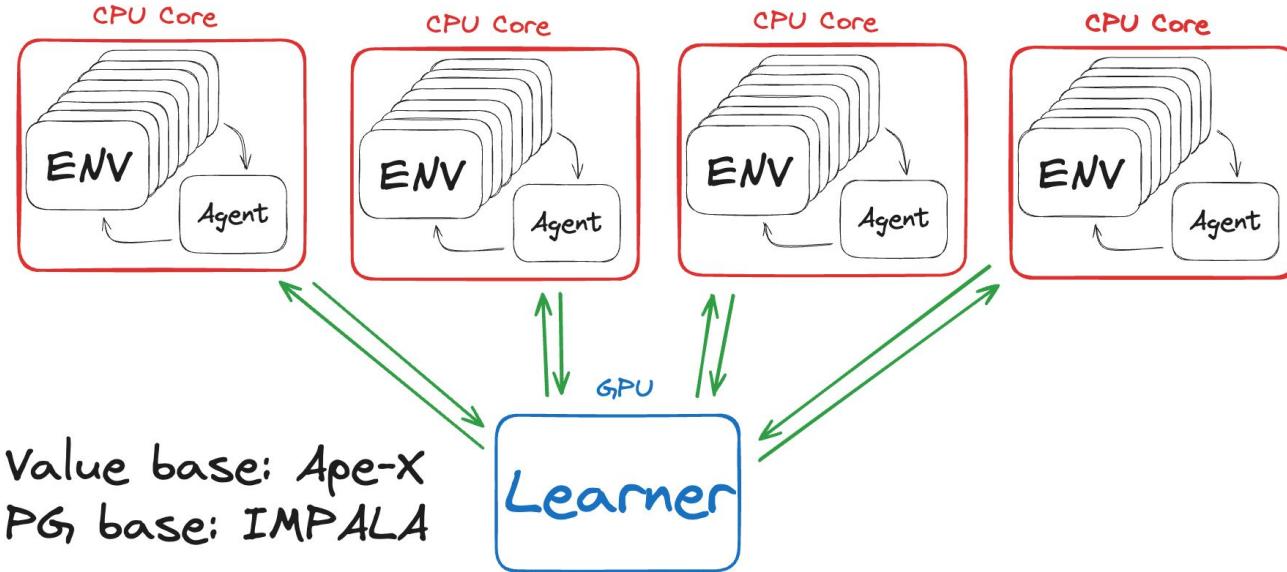
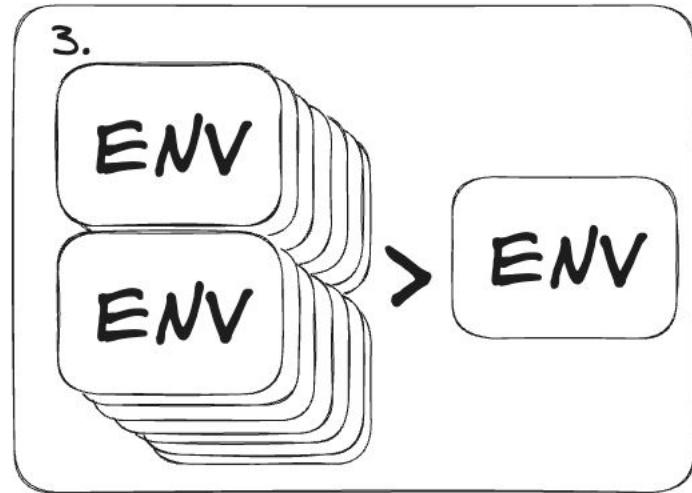


Update: Craftax was accepted at ICML 2024 as a spotlight!

- 이러한 명확한 이점에 힘입어, 많은 환경들이 새롭게 JAX 네이티브로 작성되고 있음.

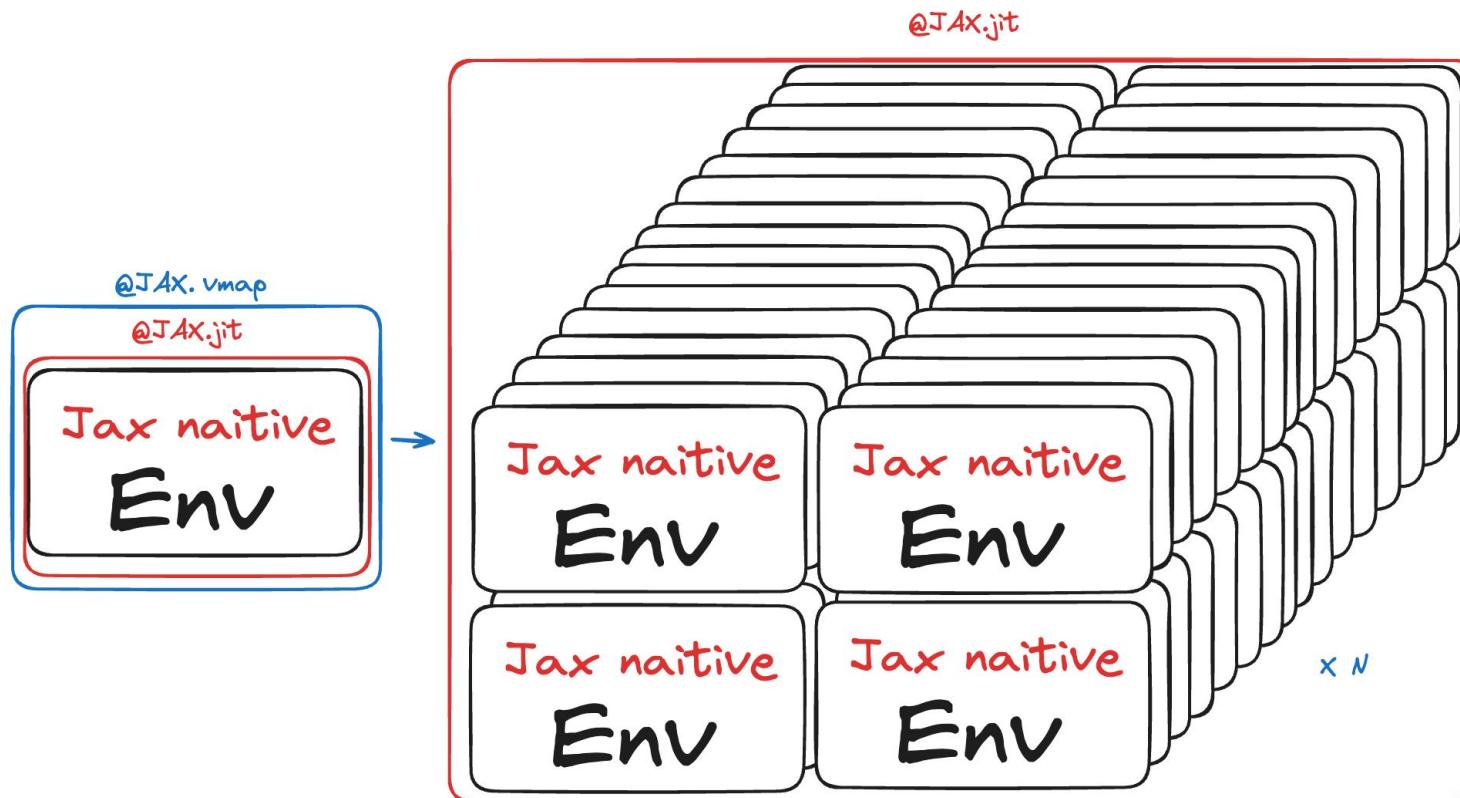


RL 의 특수성: 병렬화에 대한 이득



- RL 의 알고리즘에 따라 다르지만, 대부분 한 epoch 동안 수집되는 데이터의 수가 많으면 많을수록 성능과 학습 시간에서 매우 큰 이득이 있음
- 이 때문에 Offpolicy 에서는 Ape-X 라는 방법론, Policy Optimize 에서는 기본적으로 대부분 병렬 환경을 사용하며, 이어서는 싱글 프로세스에서는 부족한 규모의 병렬화를 위해 Impala, DDPO 같은 방법론들까지 등장
- 하지만, CPU에서의 빠른 병렬화된 환경을 위해서는 프로세스들의 분리가 필요하지만... GPU 는?

RL 의 특수성: 병렬화에 대한 이득



- JAX 로 잘 작성되고, 조건을 만족하는 함수는 jax.vmap 을 사용하여 해당 동작을 병렬적으로 동작하는 함수로 바로 변환 가능
- 이를 통해 CPU 환경들을 사용해야하는 알고리즘들과 달리 학습 구조가 복잡해질 필요가 없고 선형적인 샘플 수집, 성능 향상 가능

RL 의 특수성: 병렬화에 대한 이득

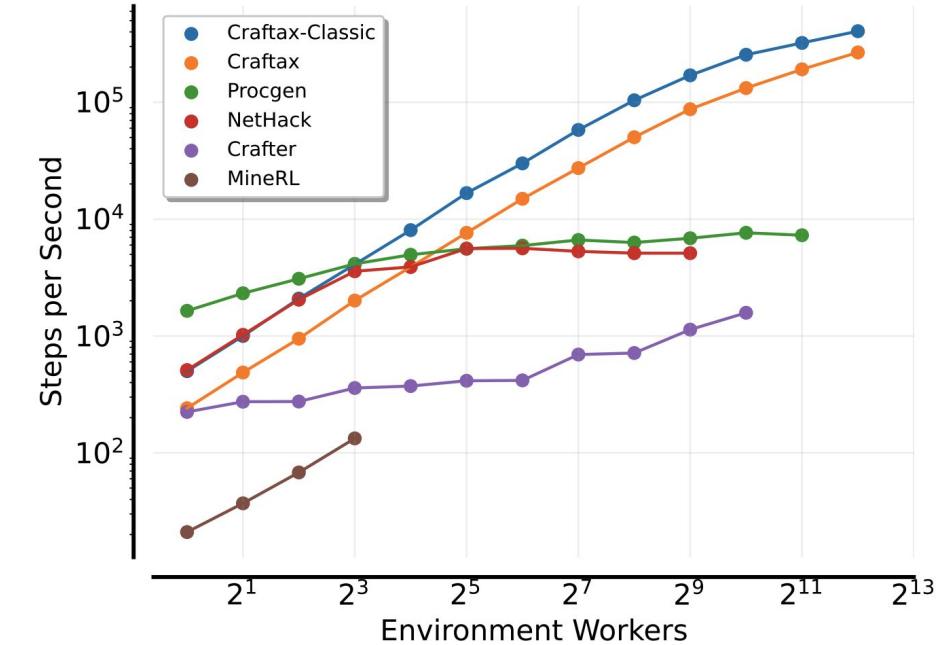
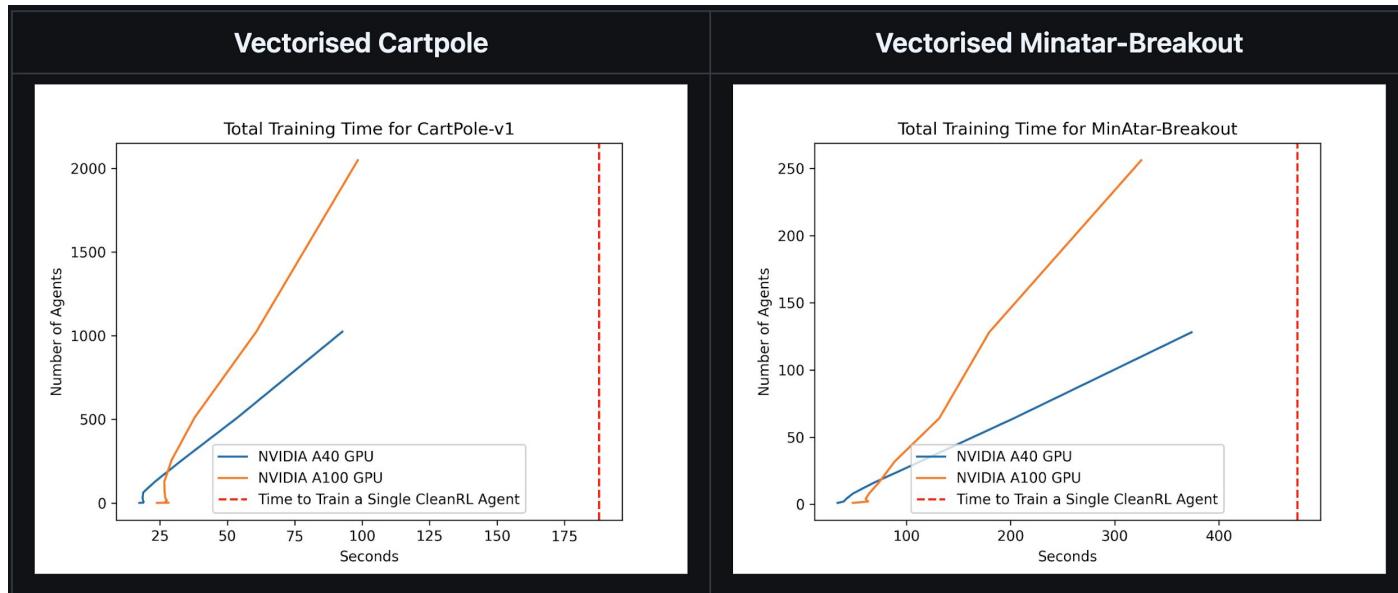
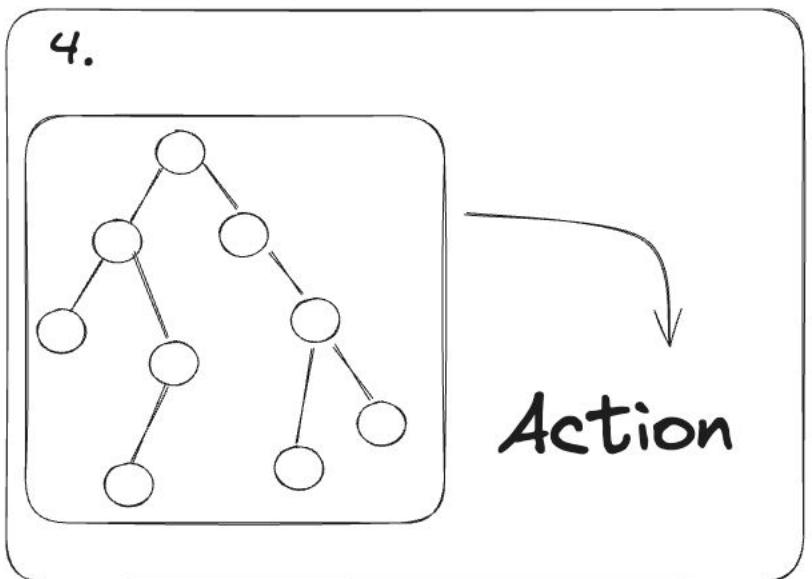


Figure 2. Speed comparison with popular benchmarks for open-ended learning. Craftax-Classic and Craftax are 257x and 169x faster than Crafter respectively. Details of the speed test are in Appendix B and best case results are in Table 1.

- 이러한 특성으로 인해, JAX로 작성된 환경은 거의 선형적으로 샘플 수집 속도를 늘릴 수 있고
- JAX로 End to End로 작성된 알고리즘은 거의 선형적으로 학습 규모를 늘릴 수 있음.

RL 의 특수성: 인퍼런스의 복잡성



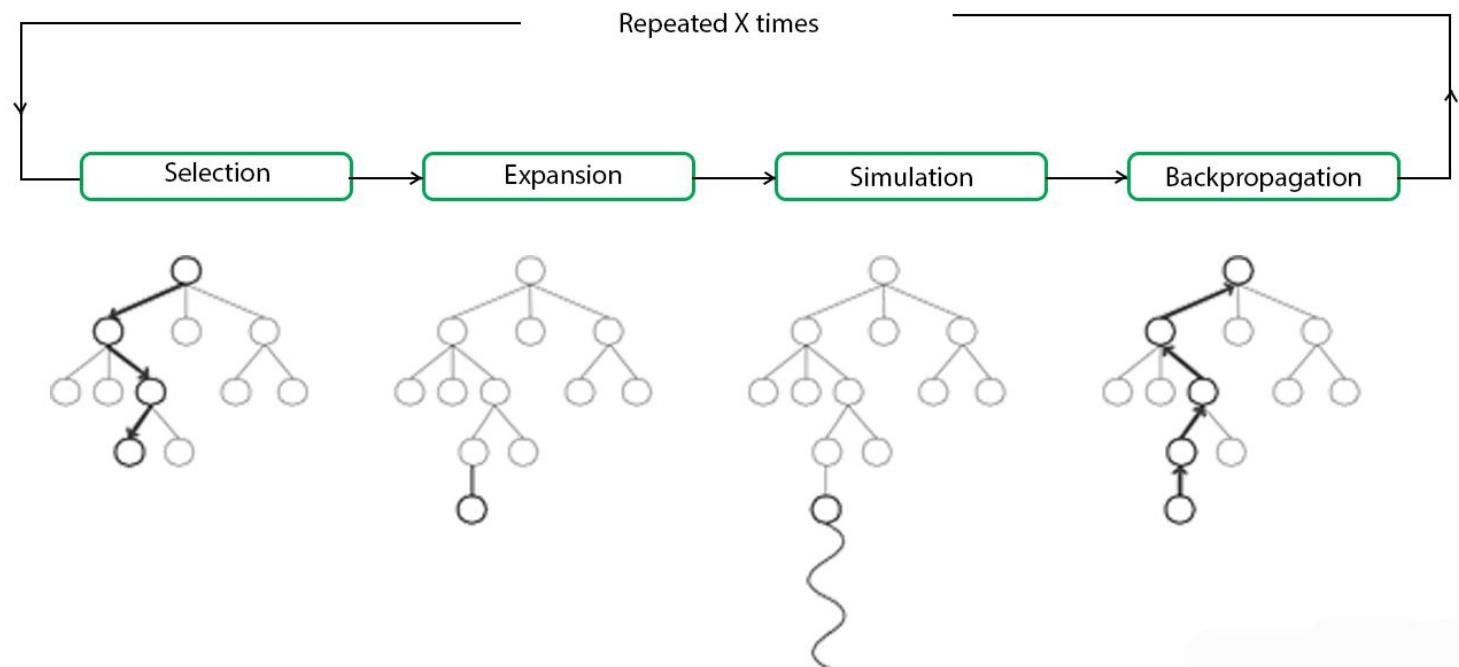
- 모든 강화학습이 그런 건 아니지만, 일부 매우 복잡한 문제를 풀어야 하는 경우 액션을 도출하는 인퍼런스 자체가 매우 복잡한 경우들이 있음
- 이 경우 복잡한 로직에 따라 모델의 계산을 여러번 거치는 파이프라인이 필요하며, 이런 로직은 단순 코드 최적화 뿐만 아니라 하드웨어간 데이터 전송이 빈번 하기 때문에 연산 가속이 어려움
- 가장 대표적인 경우가 MCTS 를 사용하는 AlphaZero, MuZero 류의 알고리즘

RL 의 특수성: 인퍼런스의 복잡성

- Deepmind 가 작성한 MCTS 의 JAX 구현
- 서치 루프 전체를 JAX 로 구현하여 C++ 구현과 비교해도 경쟁력이 있다고 주장
- AlphaZero 의 경우 환경이 JAX 로 작성되어야 하고, MuZero 는 환경이 월드 모델이므로 기본적으로 JAX 로 구현됨

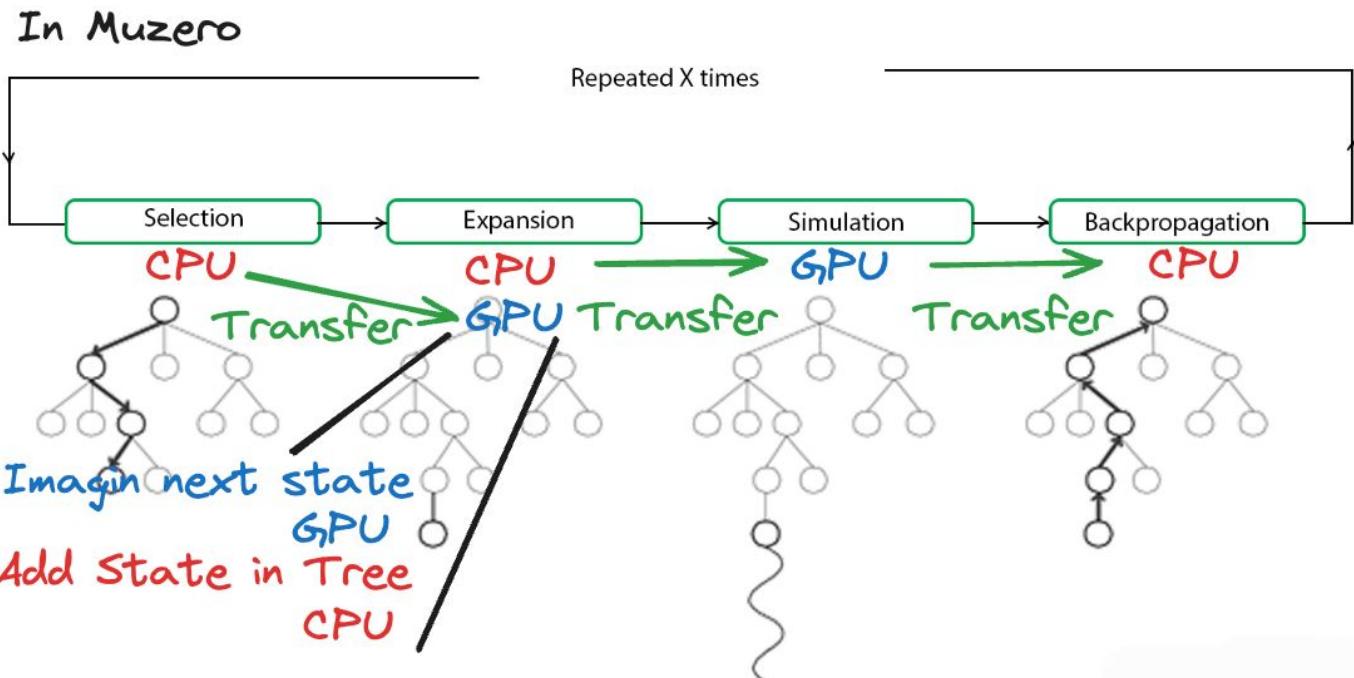
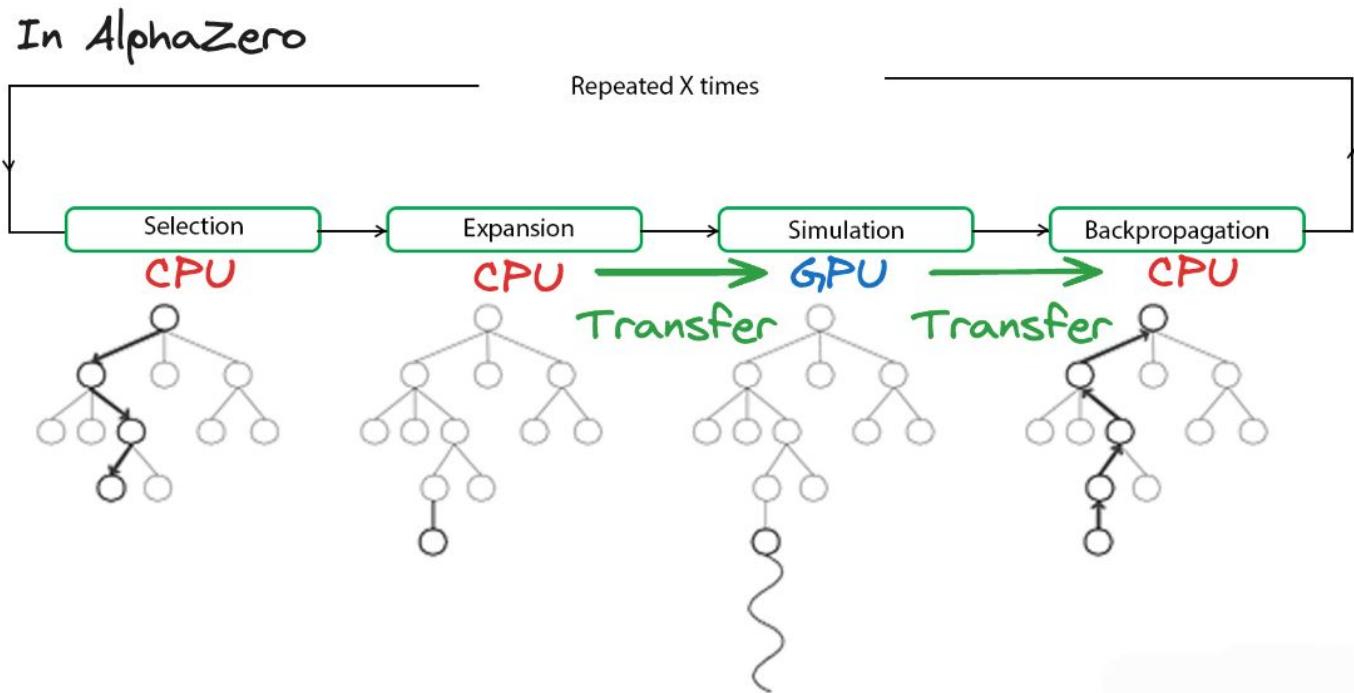
Mctx: MCTS-in-JAX

Mctx is a library with a [JAX](#)-native implementation of Monte Carlo tree search (MCTS) algorithms such as [AlphaZero](#), [MuZero](#), and [Gumbel MuZero](#). For computation speed up, the implementation fully supports JIT-compilation. Search algorithms in Mctx are defined for and operate on batches of inputs, in parallel. This allows to make the most of the accelerators and enables the algorithms to work with large learned environment models parameterized by deep neural networks.



RL 의 특수성: 인퍼런스의 복잡성

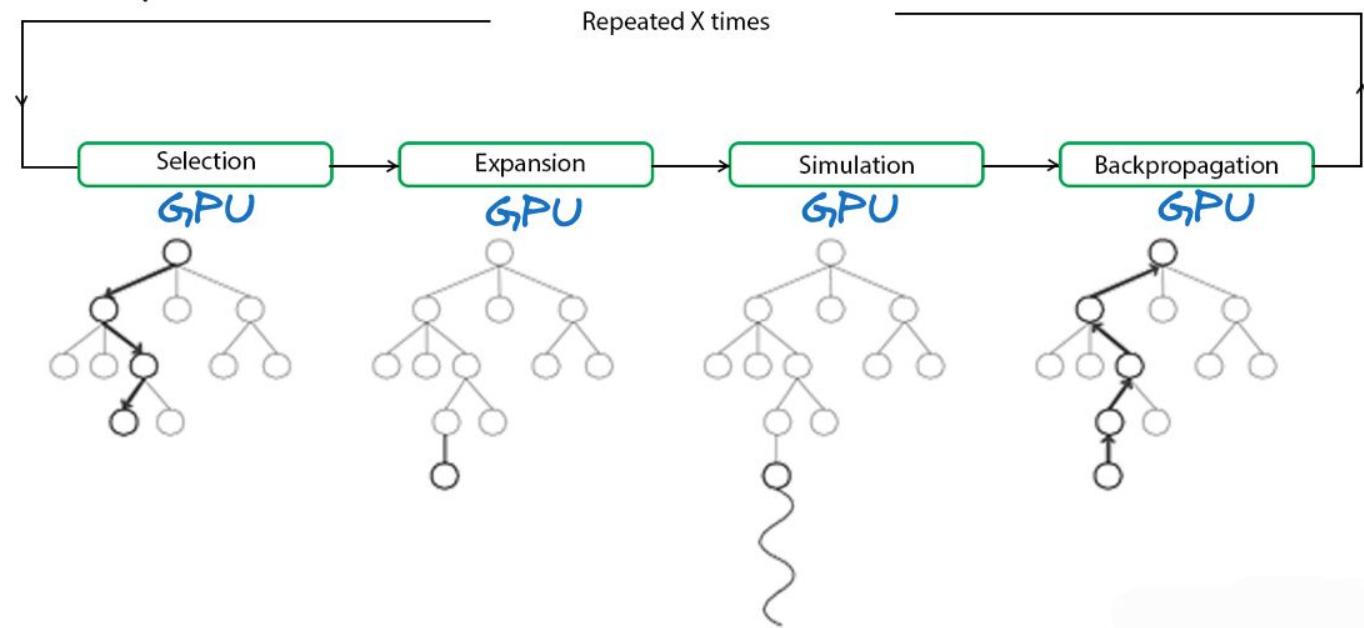
- 기존의 CPU 와 GPU 로 함께 작성될 경우
AlphaZero 와 Muzero 또한 이전에 설명한 데이다
전송 문제와 직면
- CPU 에서 작성된 코드가 C++ 로 잘 최적화
되어있어도 이런 문제로 GPU 의 실제 사용량이
크게 낮을 수 있음



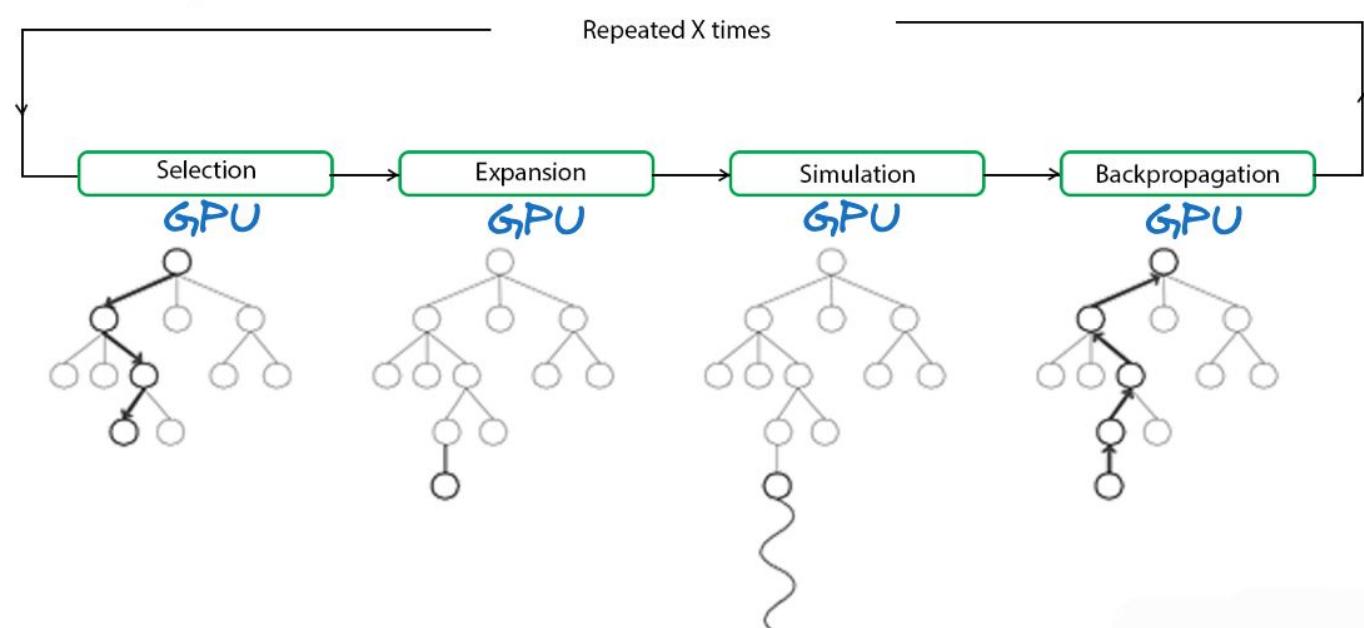
RL 의 특수성: 인퍼런스의 복잡성

- GPU에서 전체 루프가 동작할 경우 이런 문제가 배제됨
- 하지만 GPU의 싱글코어 속도는 CPU에 비하여 매우 느리므로, 아직까지는 압도적인 성능차이를 얻을 수는 없음
- 이를 위해선 MCTS의 구현이 배치 단위로 재구성이 이루어져야 함

In AlphaZero



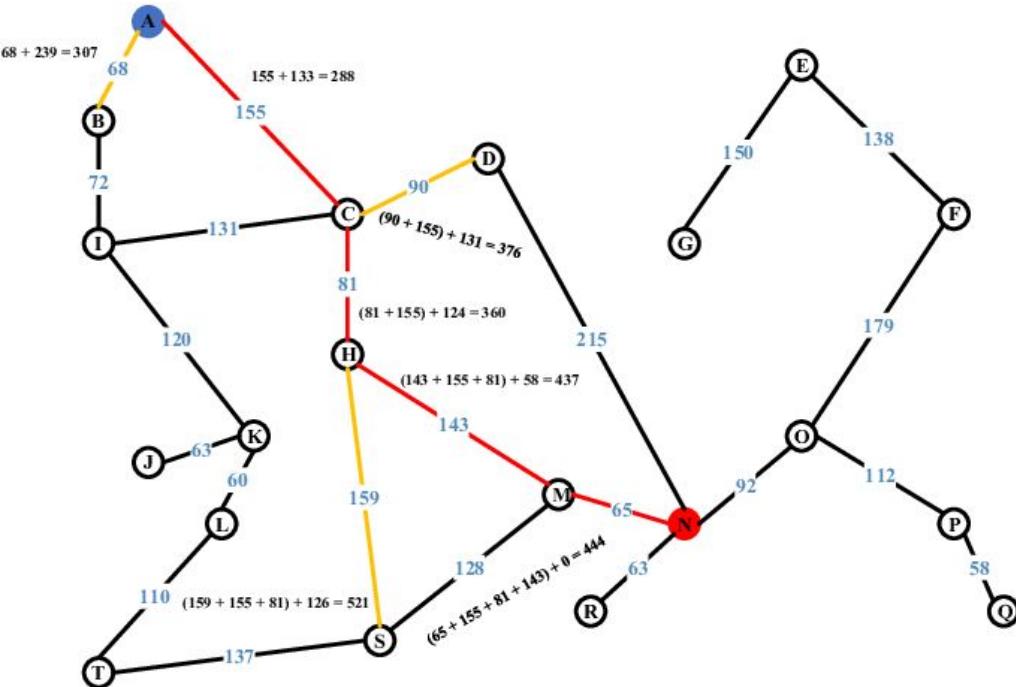
In MuZero



RL 의 특수성: 인퍼런스의 복잡성

- 개인 프로젝트로 진행한 A* 의 JAX 구현
- 서치 루프 전체를 JAX 로 구현
- 이전 구현(Python & JAX) 대비 1500 배
C++ 구현(C++ & torch(batch)) 대비 30 배

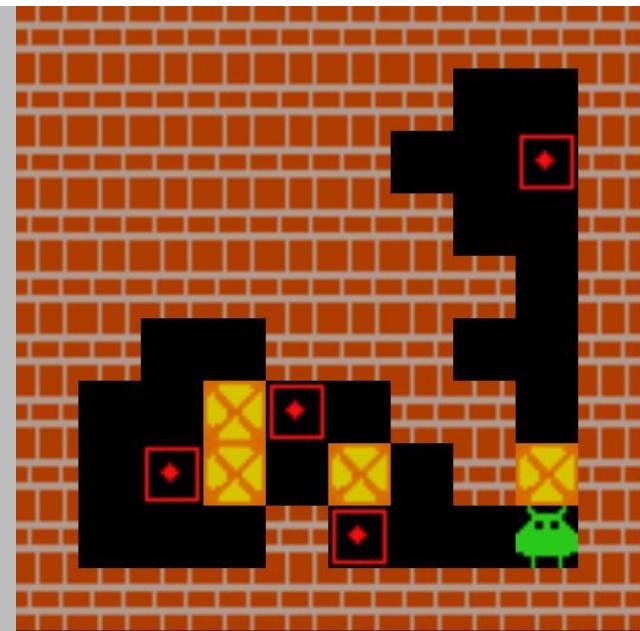
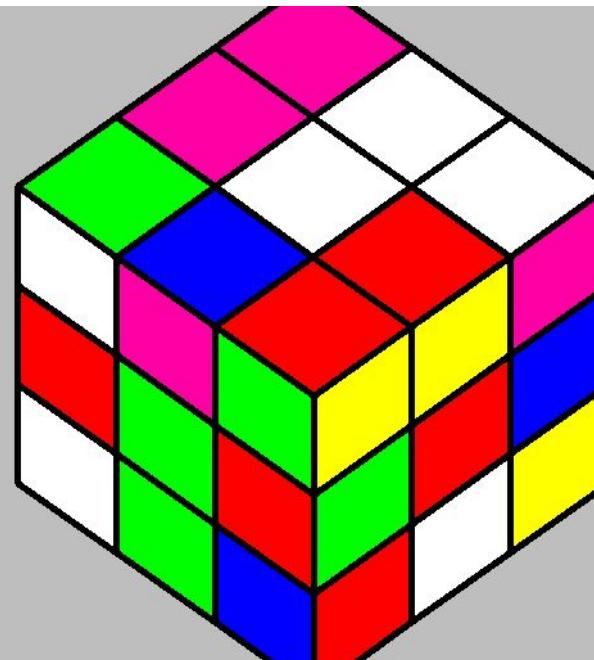
속도



JA^{xtar}

JA^{xtar}: GPU-accelerated Batched parallel A* & Q* solver in pure JAX!

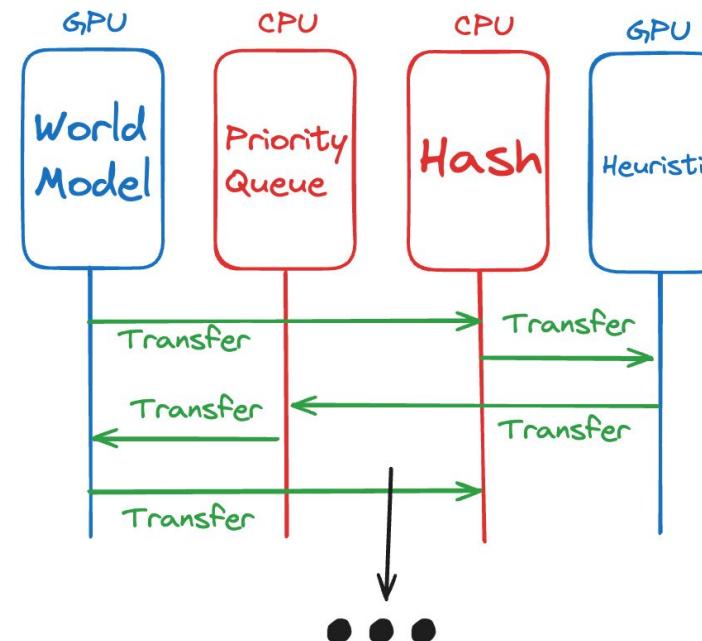
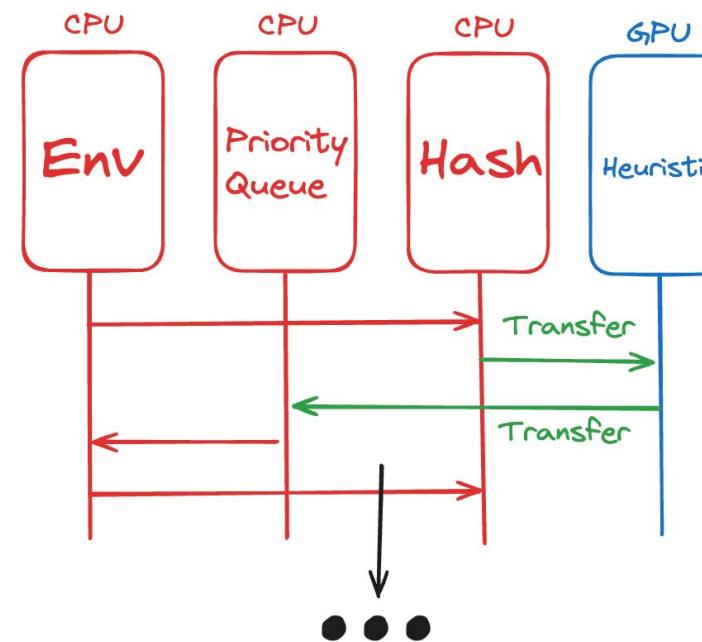
JA^{xtar} is a project with a JAX-native implementation of parallelizable a A* & Q* solver for neural heuristic search research. This project is inspired by [mctx](#) from Google DeepMind. If MCTS can be implemented entirely in pure JAX, why not A*?



RL 의 특수성: 인퍼런스의 복잡성

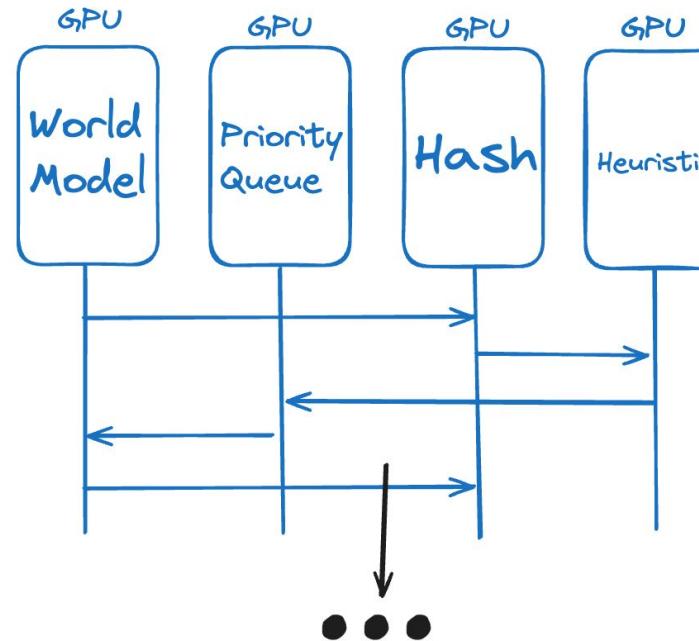
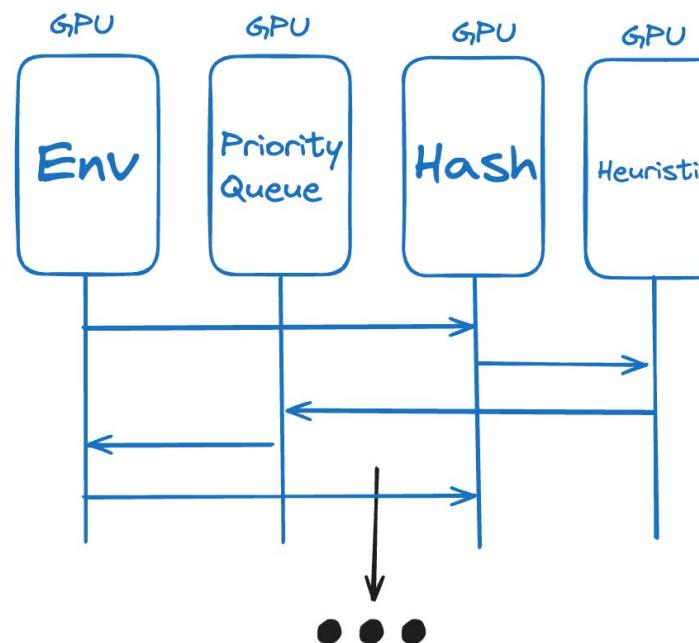
- 여기서 또한 데이터 전송 문제가 발생
- 뉴럴넷으로 동작하는 월드 모델을 사용할 경우 거의 모든 작업에

데이터 전송이 필요

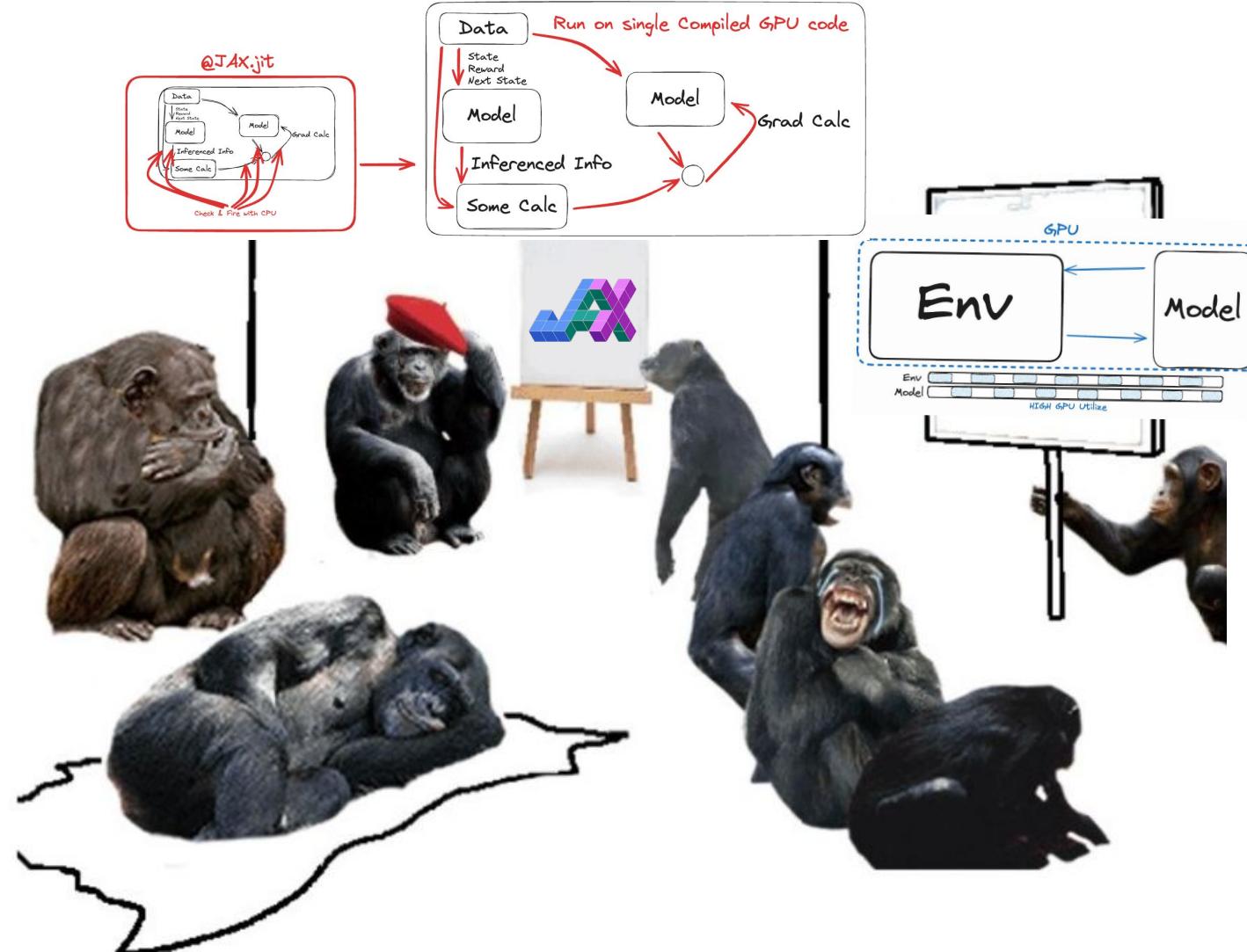


RL 의 특수성: 인퍼런스의 복잡성

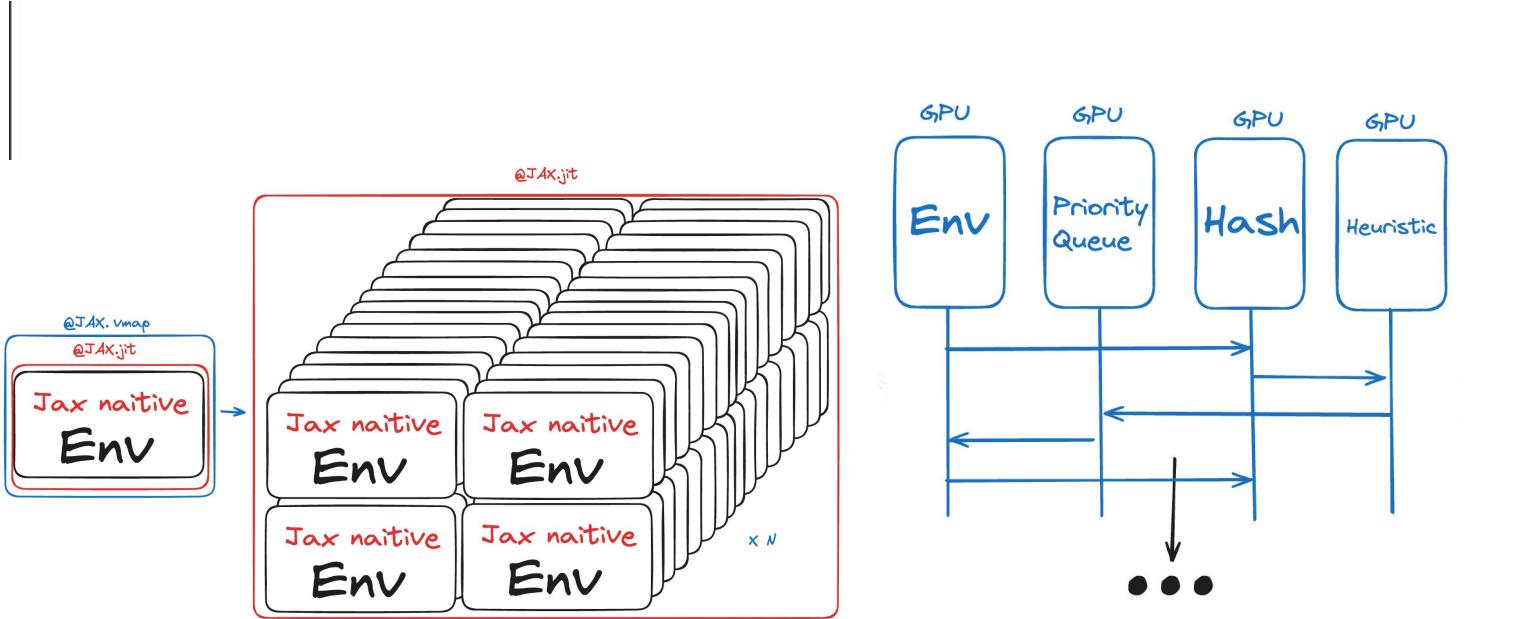
- 여기서도 GPU에서 전체 루프가 동작할 경우 이런 문제가 배제됨
- 이를 통해 상당한 성능적 이득을 낼 수 있음
- 월드 모델을 사용하는 A*의 경우 기존 논문의 결과(C++ & Pytorch)와 비교하여 300 배 이상의 성능 차이를 보임



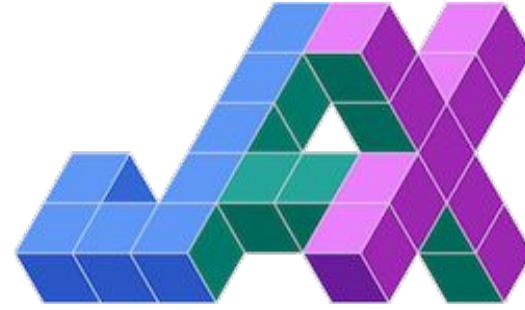
저희 속도 맙고 다른 생각을 해봅시다



저희 속도 맙고 다른 생각을 해봅시다



저희 속도 맙고 다른 생각을 해봅시다



저희 속도 맙고 다른 생각을 해봅시다

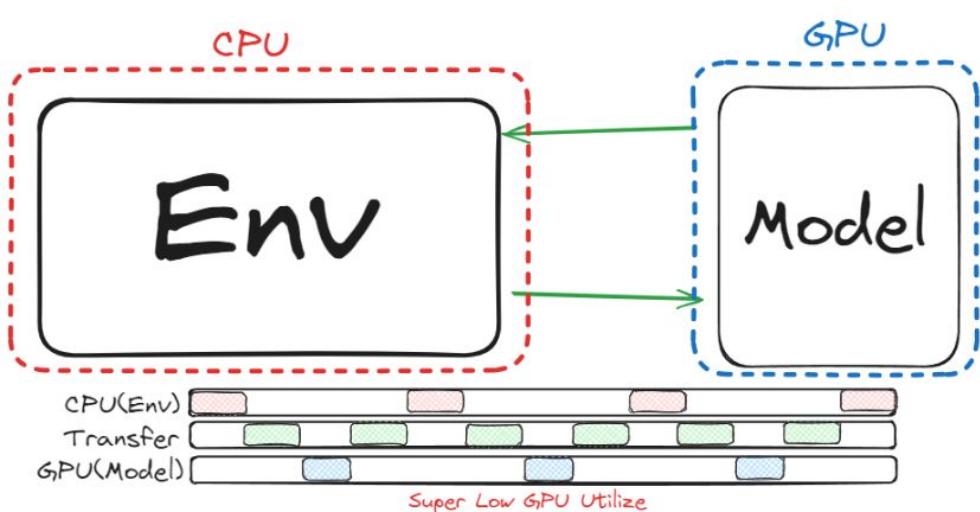
근데 이거 맞음?



Extrim Hard

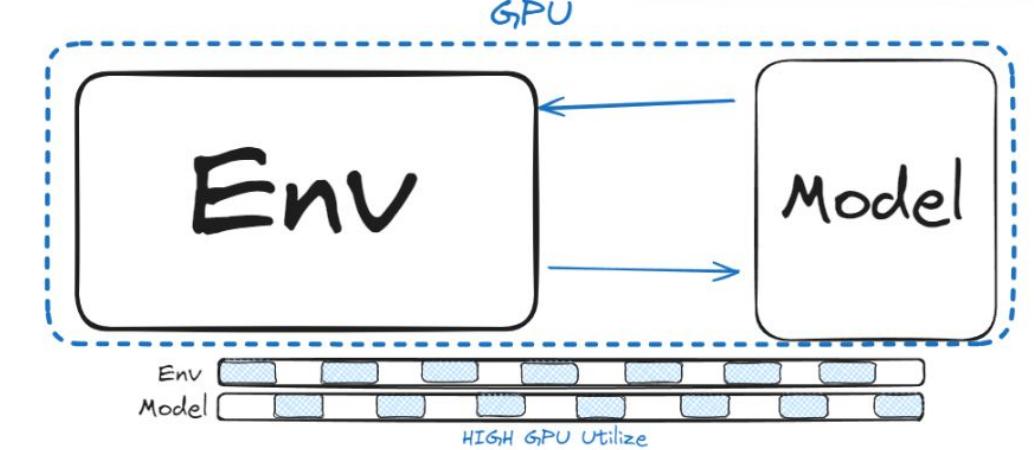
@JAX.jit

Extrim Slow



VS

Normal Gym
Env



저희 속도 말고 다른 생각을 해봅시다



- 환경을 빠르게 만들어서 해결하려고 하면 진짜 끝이 없다
- 왜? 차라리 모든 트라잭토리 다 모아서 DNN 없는 Q러닝으로 풀자고 하지
- 시뮬레이터의 원본 코드가 있어도 어려운 환경의 JAX 화 작업을 시뮬레이터를 어떻게 만들어야 하는지 막막한 실제 문제에 어떻게 적용할건가?
- 남은 평생을 새 도메인에 맞춘 시뮬레이션을 만들고 최적화하는 데에 보낼 건가?

저희 속도 말고 다른 생각을 해봅시다 - 갈!!!!!!

喝!!!!!!

!!!!!!

!!!!!!



자고로 '신앙'을 잃는것은
죽음을 의미하는법이며. . . !!



- 정말 '중요한' 문제는 시뮬레이터를 만들 가치가 있다

Ex) 문자 예측 시뮬레이션, 등등...

- 시뮬레이터를 만드는건 쉬운데, 푸는건 어려운 문제들도 많이 있다

Ex) 바둑, 조합최적화, 등등...

- Mujoco JAX, Nvidia Isaac sim, Genesis 등 어려운 3D 와 그런

시뮬레이션을 위한 GPU 시뮬레이터 들 잔뜩 나오는데 뭐가 문제야!

- 빠를수록 좋은건 당연한건데 왜 트집임? 설마 절마 '분탕' 아님?

저희 속도 말고 다른 생각을 해봅시다

아니 그... 빠른게 좋긴 한데... 근본적인 해결책은

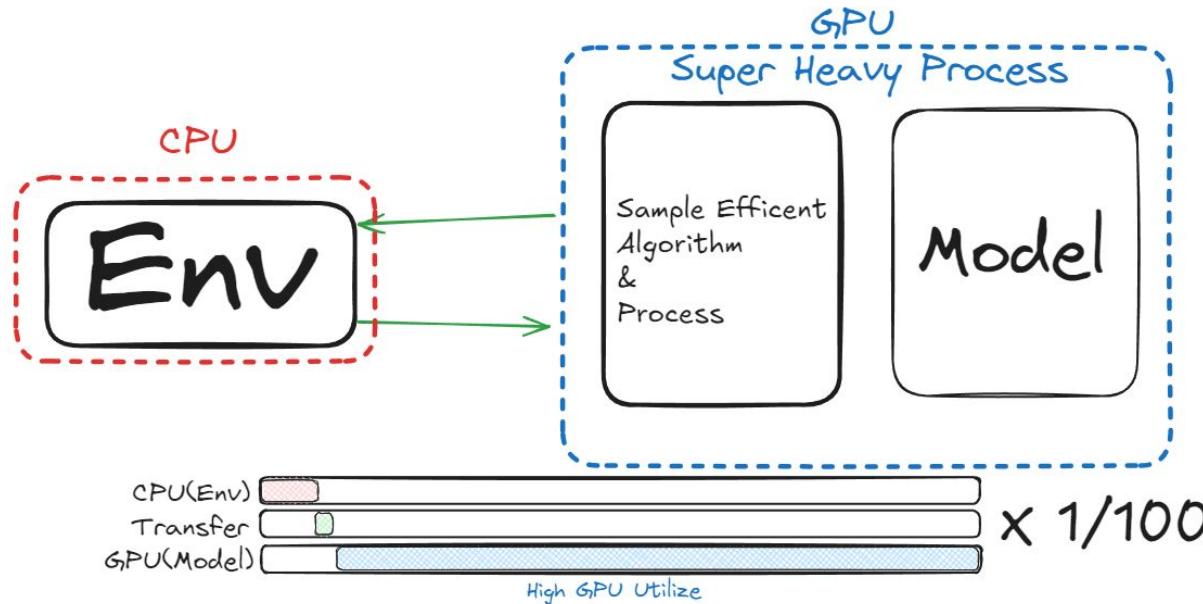
아니라는거죠...

샘플 많이 쓸 생각 하기 전에 샘플을 최대한 적게 쓰고

잘푸는법을 고민해야 하는거 아님?



저희 속도 맙고 다른 생각을 해봅시다



그래서 이렇게 하자고?

그럼 이대로만 갑시다!



저희 속도 말고 다른 생각을 해봅시다



근데 JAX 랑 RL 발표인데
이런 내용 들어가는게 맞음?

샘플 효율적 이면 다
된다는 이야기 잖아?

그 샘플 효율적인 알고리즘의 복잡한 연산
최적화는 조상님이 해주시나?

샘플 효율적 이려면 그만큼 몸비틀어서
알고리즘이 복잡해질텐데
복잡한 워크플로우와 거기서 오는
오버헤드를 JAX로 최적화 할수 있다니까?
MCTS에 A*도 짜고 저만큼 빨라지는데
안하는게 이상하지 않음?



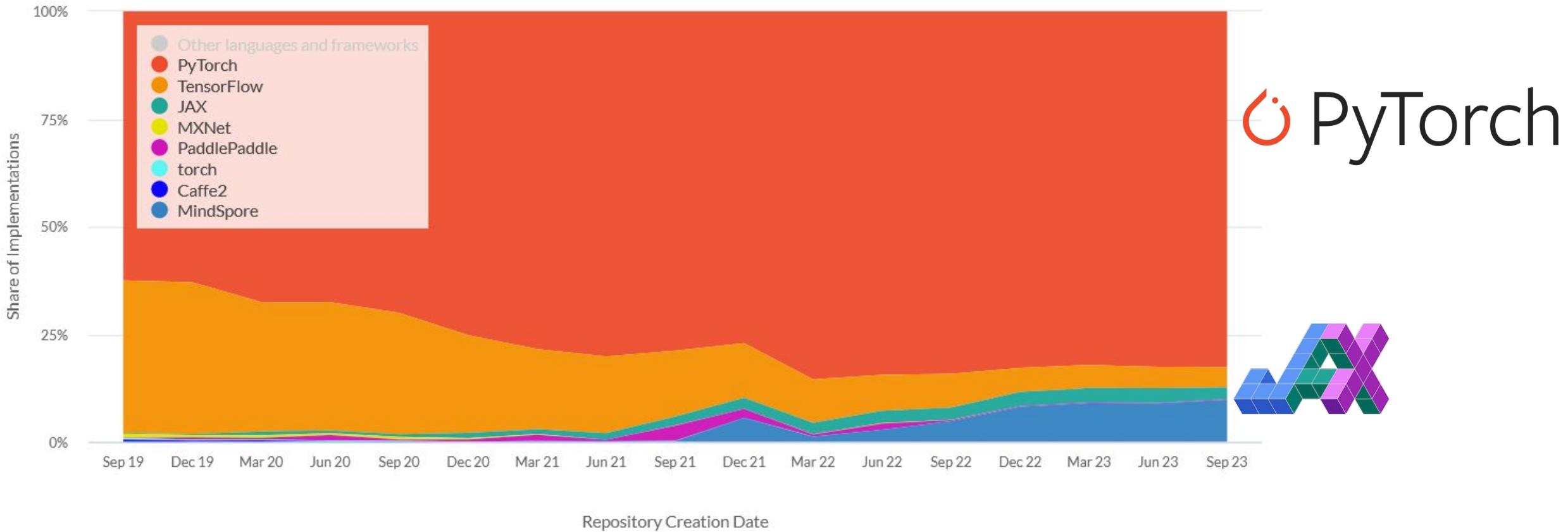
저희 속도 말고 다른 생각을 해봅시다

생각해보면 샘플 효율적 알고리즘은 환경의 속도가 아니라
학습 알고리즘의 최적화 정도 쪽에 주도권이 오니까 JAX로
최적화 하면 그 효과가 더 크겠네? 진짜 미친거 아니냐?

둘다 다 하자는 이야기구나!
크으으으~ ~



마치며



- 아직 ML 연구와 구현중 PyTorch는 무시할 수 없는 거인
- 하지만 JAX는 명확한 강점으로 점점 커져가고 있는 대안 중 하나
- 역으로 최신 RL에서는 이제 메인스트림에 가까워지고 있는 중

다시한번 말씀드리자면
우리 모두 제발 **RL**하면 **JAX** 합시다
아니어도 **JAX** 해보면 좋을 겁니다

감사합니다