DeepMind

JAX at DeepMind

NeurIPS 2020

Matteo Hessel, David Budden, Mihaela Rosca, Junhyuk Oh,
Fabio Viola, Theophane Weber, Paige Bailey

DeepMind

1. Why JAX?

Matteo Hessel (@matteohessel)

What is JAX?

JAX is a Python library designed for high-performance numerical computing

Among its key ingredients it supports:

- **Differentiation**:
    - Forward and reverse mode automatic differentiation of arbitrary numerical functions,
    - E.g: `grad`, `hessian`, `jacfwd` and `jacrev`.
- **Vectorisation**:
    - SIMD programming via automatic vectorisation,
    - E.g: `vmap`, `pmap`.
- **JIT-compilation**:
    - XLA is used to just-in-time (JIT)-compile and execute JAX programs,
    - faster CPU code, and transparent GPU and Cloud TPU acceleration..

What is JAX?

All these are implemented as **composable program transformations**

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

The **value** `fn` is evaluated like any python function

```
fn(1., 2.)  # (1**2 + 2) = 3
```

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

The **gradient** `df_dx = grad(fn)` is also a function

```
df_dx(1., 2.)  # df_dx = 2*x = 2*1 = 2
```

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

The **second-order gradient** `df2_dx = grad(grad(fn))` is also a function

```
df2_dx(1., 2.)  # df2_dx = d(2*x)_dx = 2
```

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

The **compiled second-order gradient** `df2_dx = jit(grad(grad(fn)))` is also a function

```
df2_dx(1., 2.)  # 2, also traces the code and compiles it
```

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

The **compiled second-order gradient** `df2_dx = jit(grad(grad(fn)))` is also a function

```
df2_dx(1., 2.)  # 2, also traces the code and compiles it
df2_dx(1., 2.)  # 2, executes the XLA pre-compiled code
```

But a much faster one after the first execution :)

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

The **batched compiled second-order gradient** `df2_dx = vmap(jit(grad(grad(fn))))` is also a function

```
xs = jnp.ones((batch_size,))
df2_dx(xs, 2 * xs)  # [2, 2], if batch_size=2
```

What is JAX?

All these are implemented as **composable program transformations**

Consider a numerical function:

```
def fn(x,y):
  return x**2 + y
```

So is its **multi-gpu batched compiled second-order gradient** `df2_dx = pmap(vmap(jit(grad(grad(fn)))))`

```
xs = jnp.ones((num_gpus, batch_size,))
df2_dx(xs, 2 * xs)  # [[2, 2], [2, 2]], if batch_size=2 and num_gpus=2
```

Why JAX?

JAX is simple but very flexible

- API for numerical functions is fully consistent with NumPy,
- Both Python and NumPy are widely used and familiar,
- Few abstractions (grad, jit, vmap, pmap) but powerful and composable!
- The functional programming style helps writing code that "looks like the math"
- Not a vertically integrated but with a rich ecosystem and community around it
- Battle tested extensively in the past year on projects ranging from Optimisation, SL, GANs, RL, ...

DeepMind

2. Our JAX Ecosystem

David Budden (@davidmbudden)

Why an Ecosystem?

DeepMind Researchers have had great initial success with JAX

- How can we continue to support and accelerate their work?

Considerations

- JAX is **not** a vertically integrated ML framework (this is a good thing!)
- Needs to support rapidly evolving DeepMind Research requirements
- Where possible, strive for consistency + compatibility with
    - Our TF ecosystem (Sonnet, TRFL, ...)
    - Our research frameworks (Acme, ...)

DeepMind JAX Ecosystem

- Libraries of reusable and un-opinionated JAX **components**
- Each library does one thing well and supports **incremental buy-in**
- Open source everything to enable research sharing + reproducibility

Haiku

Haiku is a tool
For building neural networks
Think: "Sonnet for JAX"

Motivation

- JAX programs are functional
- NN params/state better fit the OO paradigm

Haiku (github.com/deepmind/dm-haiku)

- Converts stateful modules to pure functions
- API-matches Sonnet, porting from TF is trivial
  - Have reproduced AlphaGo, AlphaStar, AlphaFold, ...
- Mature API and widely adopted

```python
import jax
import haiku as hk

@hk.transform
def loss_fn(images, labels):
  model = hk.nets.MLP([1000, 100, 10])
  logits = model(images)
  labels = one_hot(labels, 1000)
  return losses.softmax_cross_entropy(logits, labels)

images, labels = next(dataset)
params = loss_fn.init(rng_key, images, labels)
loss = loss_fn.apply(params, images, labels)
```

Optax

The artist formerly known as jax.experimental.optix

Motivation

- Gradient processing is fundamental to ML Research
- Like NNs, optimizers are stateful

Optax (github.com/deepmind/optax)

- Gradient processing and optimization library
- Comprehensive library of popular optimizers
- Simplifies gradient-based updates of NN params
    - Compatible with all popular JAX NN libraries
- Mature API and widely adopted

```
import jax
import optax

params = ... // a JAX tree

opt = optax.adam(learning_rate=1e-4)
state = opt.init(params)

@jax.jit
def step(state, params, data):
  dloss_dparams = jax.grad(loss_fn)(*data)
  updates, state = opt.update(dloss_dparams, state)
  params = optax.apply_updates(params, updates)
  return state, params

for data in dataset:
  state, params = step(state, params, data)
```

RLax

"RLax is the best RL textbook I've read!"
– Anonymous

Motivation

- Reinforcement Learning is hard, getting it wrong is easy
- Want a common substrate for *sharing* new ideas

RLax (github.com/deepmind/rlax)

- Library of mathematical operations related to RL
- Emphasis on readability
- Building blocks rather than complete algorithms
  - But, lots of full agent examples available
- Widely adopted for RL research

```python
import jax
import optax
import rlax

def loss_fn(params, o_tm1, a_tm1, r_t, d_t, o_t):
  q_tm1 = network.apply(params, o_tm1)
  q_t = network.apply(params, o_t)
  td_error = rlax.q_learning(q_tm1, a_tm1, r_t, d_t, q_t)
  return rlax.l2_loss(td_error)

@jax.jit
def learn(state, params, transition):
  dloss_dparams = jax.grad(loss_fn)(*transition)
  updates, state = opt.update(dloss_dparams, state)
  params = optax.apply_updates(params, updates)
  return state, params
```

The Future

Our ecosystem is evolving rapidly

- Graph neural networks (github.com/deepmind/jraph)
- Testing & reliability (github.com/deepmind/chex)
- ... plus others coming soon!

Checkout examples using DeepMind's JAX ecosystem

- Supervised Learning (github.com/deepmind/jaxline)
- Reinforcement learning (github.com/deepmind/acme)

Others are also building great stuff with JAX

- Neural Networks (github.com/google/flax)
- Molecular Dynamics (github.com/google/jax-md)
- Chemical Modelling (github.com/deepchem/jaxchem)

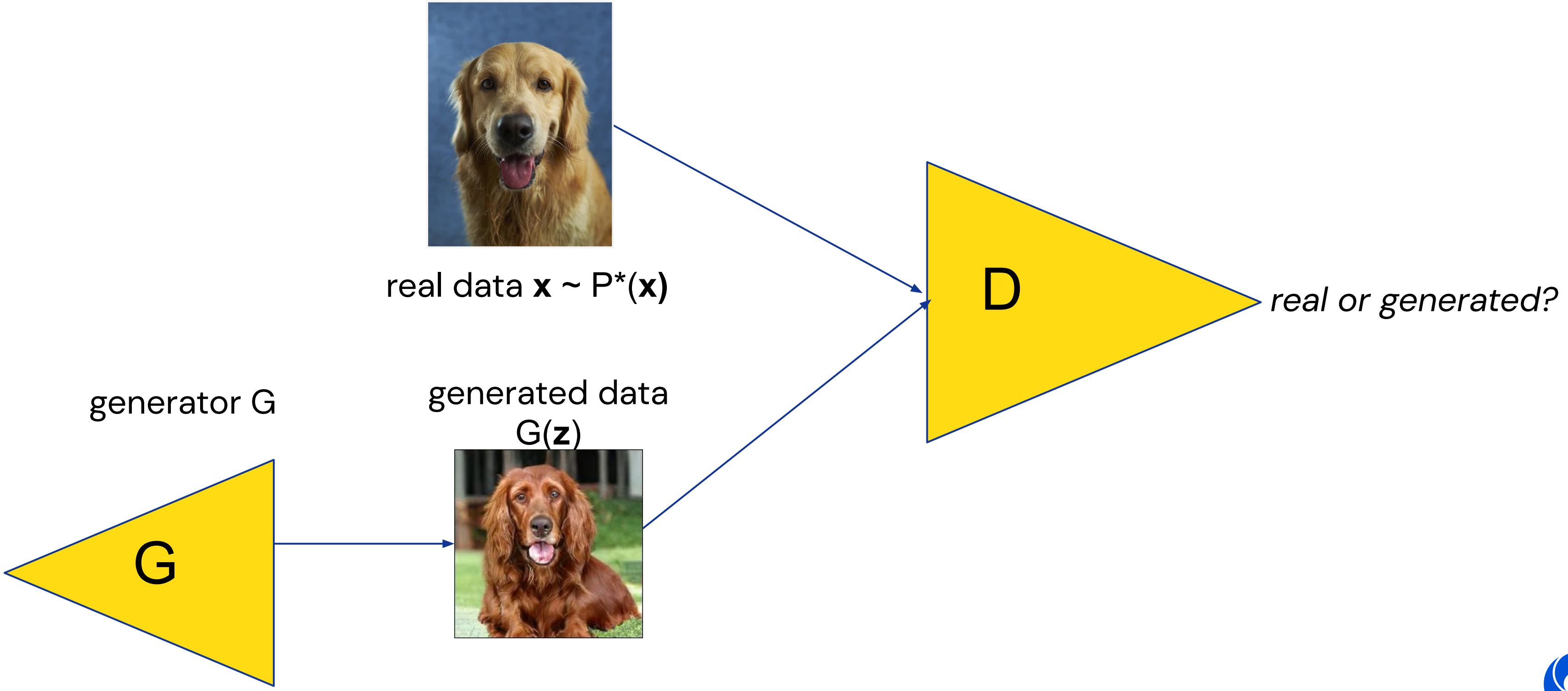
DeepMind

3. Generative Models & GANs

Mihaela Rosca

Slide thanks to Jeff Donahue.

GAN intro



Join the discussion on Twitter (#JAXecosystem)

GANs - Gradients as first order citizens

```
for _ in range(num_disc_updates):
  rng, rng_disc = jax.random.split(rng, 2)
  disc_grads = jax.grad(gan.disc_loss)(params.disc,   params.gen, data_batch, rng_disc)
  disc_update, disc_opt_state = optimizers.disc.update(disc_grads, opt_state.disc)
  new_disc_params = optax.apply_updates(params.disc, disc_update)
```

```
for _ in range(num_gen_updates):
    rng, rng_gen = jax.random.split(rng, 2)
    gen_grads = jax.grad(gan.gen_loss)(params.gen, new_disc_params, data_batch, rng_gen)
    gen_update, gen_opt_state = optimizers.gen.update(gen_grads, opt_state.gen)
    new_gen_params = optax.apply_updates(params.gen, gen_update)
```

Gradient as first order citizens - easy tracking

```
for _ in range(num_disc_updates):
  rng, rng_disc = jax.random.split(rng, 2)
  disc_grads = jax.grad(gan.disc_loss)(params.disc, params.gen, data_batch, rng_disc)
  disc_update, disc_opt_state = optimizers.disc.update(disc_grads, opt_state.disc)
  new_disc_params = optax.apply_updates(params.disc, disc_update)
```

Direct access to gradients (not hidden inside the optimizer)!
Can easily track gradients at different layers, effect of regularizers on gradients, etc.

With haiku, disc_grads is a dictionary from module name to variables:

```
disc_grads=  {
     ‘disc_net/layer1’: {‘w’: jnp.array(...)}, {‘b’: jnp.array(...)},
     ‘disc_net/layer2’: {‘w’: jnp.array(...)}, {‘b’: jnp.array(...)},
```

Having more control - easier to make the right decisions

```
def disc_loss(self, disc_params, gen_params, state, data_batch, rng):
  samples, gen_state = self.sample(gen_params, state.gen, rng, data_batch.shape[0])

  disc_inputs = jnp.concatenate((data_batch, samples), axis=0)
  disc_outpus, disc_state = self.disc.apply(disc_params, state.disc, disc_inputs)
  data_disc_output, samples_disc_output = jnp.split(disc_outpus, [data_batch.shape[0],], axis=0)

  loss = cross_entropy_disc_loss(data_disc_output, samples_disc_output)
  state = (disc_state, gen_state)
  return loss, state
```

Having more control - easier to make the right decisions

```
def disc_loss(self, disc_params, gen_params, state, data_batch, rng):
 samples, _ = self.sample(gen_params, state.gen, rng, data_batch.shape[0])

 disc_inputs = jnp.concatenate((data_batch, samples), axis=0)
 disc_outpus, disc_state = self.disc.apply(disc_params, state.disc, disc_inputs)
 data_disc_output, samples_disc_output = jnp.split(disc_outpus, [data_batch.shape[0],], axis=0)

 loss = cross_entropy_disc_loss(data_disc_output, samples_disc_output)
 state = (disc_state, state.gen)
 return loss, state
```

Functional approach makes code close to math

Reparametrization trick (GANs, VAEs, etc):

$$\nabla_\theta \mathbb{E}_{p_\theta(\mathbf{x})} f(\mathbf{x}) = \mathbb{E}_{p(\epsilon)} \nabla_\theta f(g_\theta(\epsilon)), \qquad \mathbf{x} = g_\theta(\epsilon)$$

```
def reparametrized_jacobians(function, params, dist_builder, rng, num_samples):
  def surrogate(params):
    dist = dist_builder(*params)
    return jax.vmap(function)(dist.sample((num_samples,), seed=rng))

  return jax.jacfwd(surrogate)(params)
```

Bonus! Easily to get jacobians - grad for each batch element!

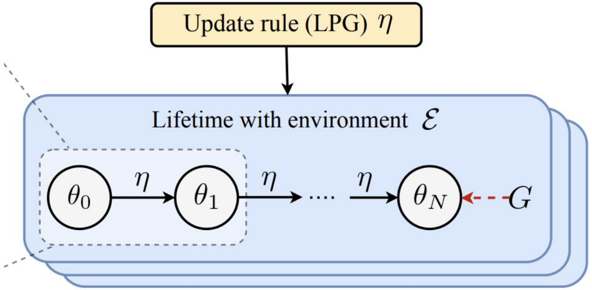
DeepMind

4. Meta-gradients

Junhyuk Oh (@junh_oh)

Discovering RL Algorithms (Oh et al., NeurIPS 2020)

Goal: Meta-learn a RL update rule from a distribution of agents and environments.

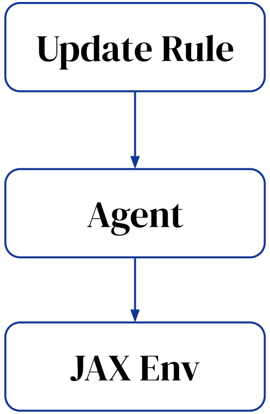


Technical Challenges

- **Parallel**: Simulate independent learning agents, each of which is interacting with its own environment.
- **Synchronous**: Apply the same update rule (i.e., meta-learner) to all learning agents.
- **Meta-gradient**: Calculate meta-gradient over the update procedure.
- **Scalability**: Increase the number of learning agents without introducing extra cost.

How did we implement?

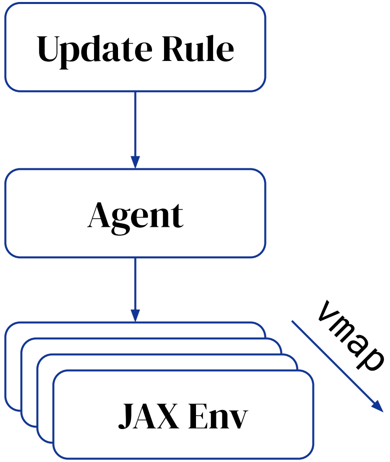
[Step 1] Implement a **single** update rule / **single** agent / **single** JAX environment interactions.



How did we implement?

[Step 1] Implement a **single** update rule / **single** agent / **single** JAX environment interactions.

[Step 2] Add **vmap** to implement a **single** update rule / **single** agent / **multi** JAX environment interactions.

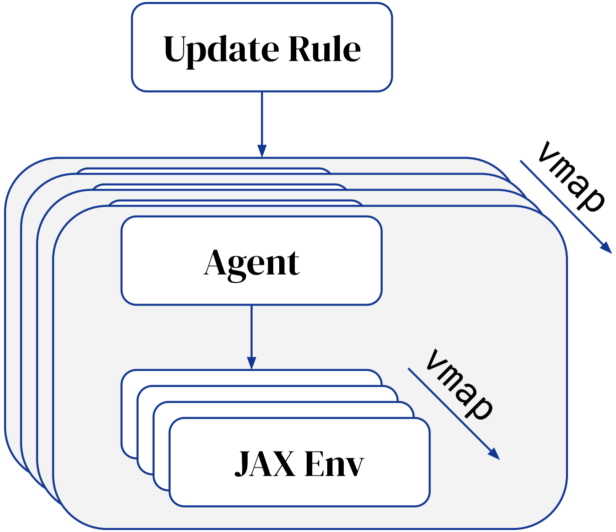


How did we implement?

[Step 1] Implement a **single** update rule / **single** agent / **single** JAX environment interactions.

[Step 2] Add **vmap** to implement a **single** update rule / **single** agent / **multi** JAX environment interactions.

[Step 3] Add **vmap** to implement a **single** update rule / **multi** agent / **multi** JAX environment interactions.



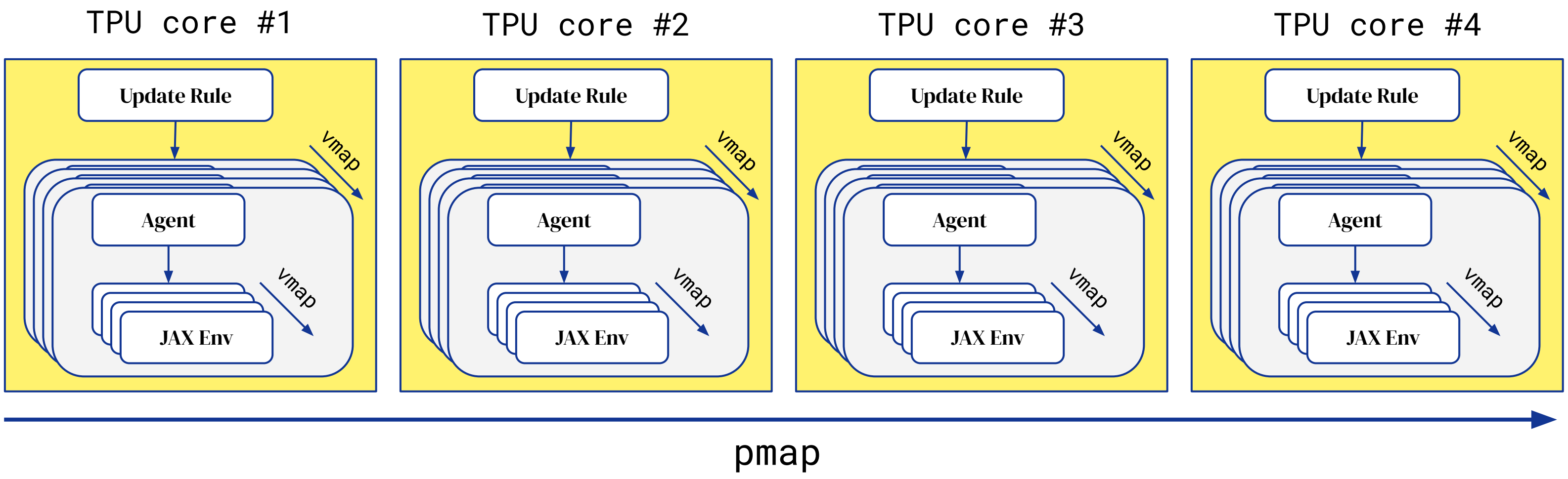
How did we implement?

[Step 1] Implement a **single** update rule / **single** agent / **single** JAX environment interactions.

[Step 2] Add **vmap** to implement a **single** update rule / **single** agent / **multi** JAX environment interactions.

[Step 3] Add **vmap** to implement a **single** update rule / **multi** agent / **multi** JAX environment interactions.

[Step 4] Add **pmap** to implement **multiple** copies of them across TPU cores with a **shared update rule**.



Pseudocode and Result

```
def inner_update(params, meta_params, rng, env_state):

  def inner_loss(params, meta_params, rng, env_state):
    # Generate rollout and apply update rule.
    rollout = jax.vmap(do_rollout, in_axes=(None, 0, 0))(
        params, rng, env_state)
    return jax.vmap(apply_update_rule, in_axes=(None, 0))(
        meta_params, rollout)

  # Calulate gradient and update parameters.
  g = jax.grad(inner_loss)(params, rollout, meta_out)
  new_params = jax.tree_multimap(lambda p, g: p - g, params, g)
  return new_params

def meta_grad(meta_params, params, rng, env_state):

  def outer_loss(meta_params, params, rng, env_state):
    new_params = jax.vmap(inner_update, in_axes=(0, None, 0, 0))(
        params, meta_params, rng, env_state)
    return jax.vmap(validate, in_axes=(0, None))(new_params, meta_params)

  # Calulate meta-gradient.
  meta_g = jax.grad(outer_loss)(meta_params, params, rng, env_state)
  return jax.lax.pmean(meta_g, ‘i’)
```

Using 16-core TPUv2

1K parallel learning agents

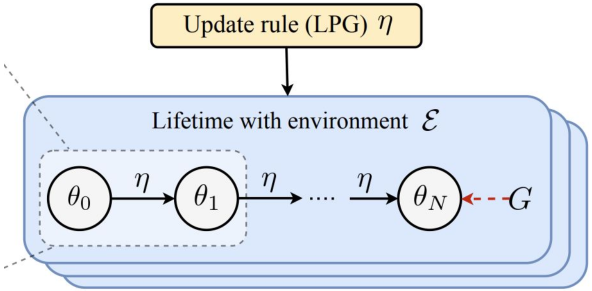
60K parallel environments

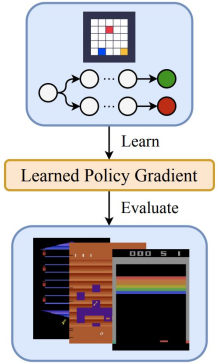
1 shared update rule

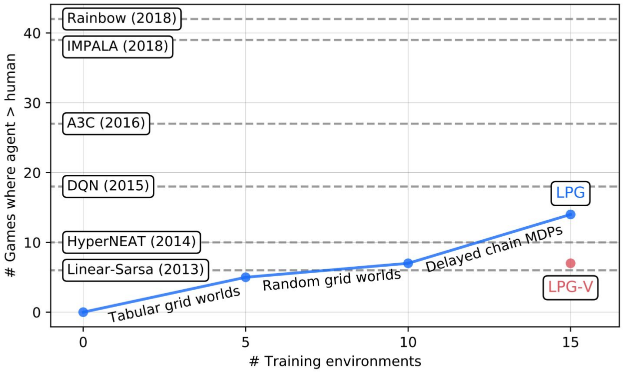
3M steps per second

Summary

Goal: Meta-learn a RL update rule from a distribution of agents and environments.







Technical Challenges

- **Parallel**: Simulate independent learning agents, each of which is interacting with its own environment.
- **Synchronous**: Apply the same update rule (i.e., meta-learner) to all learning agents.
- **Meta-gradient**: Calculate meta-gradient over the update procedure.
- **Scalability**: Increase the number of learning agents without introducing extra cost.

→ JAX + TPU helped address the above without requiring much engineering effort.

DeepMind

5. Search

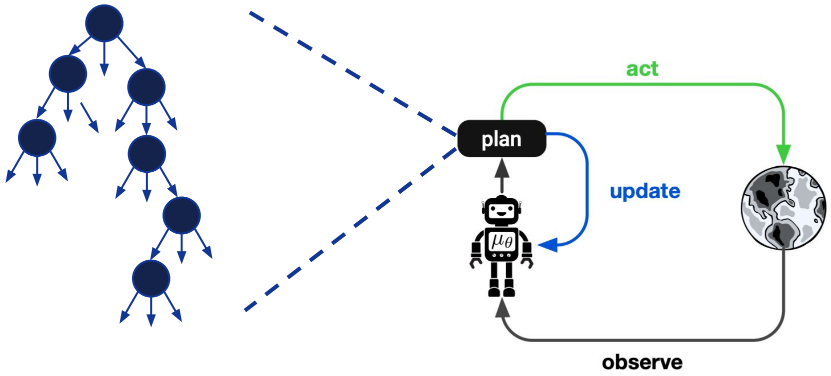
{Fabio Viola (@fabiointheuk), Theophane Weber(@theophaneweber)}

Shifting gears a bit: search and model-based RL in jax

So far, we have mostly looked at applications of Jax which leverage its gradient computation capabilities.

Is this all we can use Jax for?

Here we showcase another application where Jax enables fast research iteration:



Monte-Carlo Tree Search in a model-based RL setting, as seen in alphazero/muzero

Challenges:

- Integration of control logic and neural network machinery (tricky to debug!)
- Scalability and parallelism
- Typical model-based RL issues around data (use of replay, synthetic data, use of data for policy vs model, etc)

Why implement a model/search-based RL algorithm?

"Model-free algorithms are in turn far from the state of the art in domains that require ***precise and sophisticated lookahead***, such as chess and Go"
-*Schrittwieser et al. (2019)*

"By employing search, we can find strong move sequences potentially ***far away*** from the apprentice policy, accelerating learning in complex scenarios"
-*Anthony et al. (2017)*

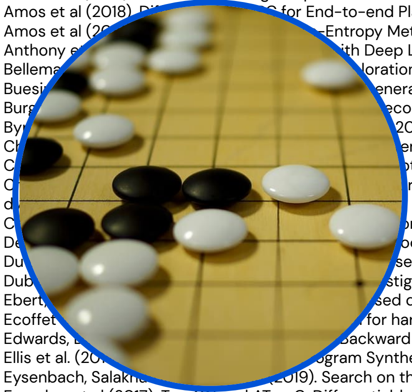
"....predictive models can enable a real robot to manipulate ***previously unseen*** objects and solve new tasks"
-*Ebert et al. (2018)*

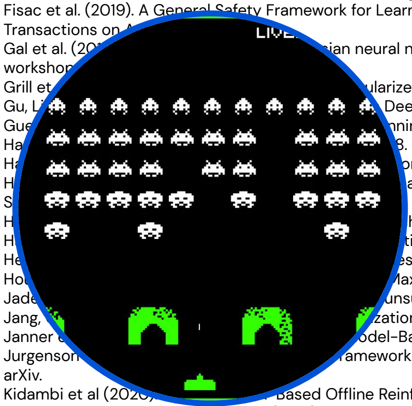
"Model-based planning is an essential ingredient of human intelligence, enabling ***flexible adaptation*** to new tasks and goals"
-*Lake et al. (2016)*

"...a flexible and general strategy such as mental simulation allows us to reason about a wide range of scenarios, even ***novel*** ones..."
-*Hamrick (2017)*

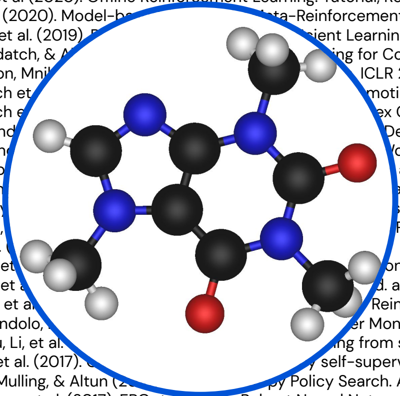
"...[models] enable better ***generalization*** across states, remain valid across tasks in the same environment, and exploit additional unsupervised learning signals..."
-*Weber et al. (2017)*

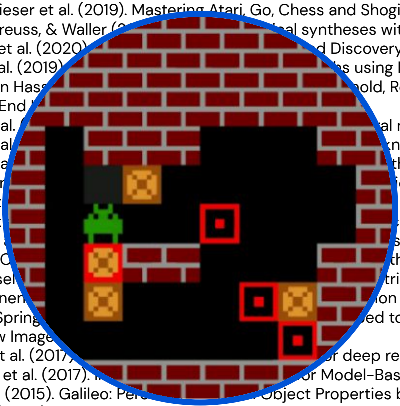
Abraham (2020). The Cambridge Handbook of the Imagination.
Agostinelli et al. (2019). Solving the Rubik's Cube with Deep Reinforcement Learning and Search. NMI.
Allen et al. (2019). The tools challenge: Rapid trial-and-error learning in physical problem solving. CogSci 2019
Amos et al (2018). ... for End-to-end Planning and Control. NeurIPS
Amos et al (20... -Entropy Method. arXiv.
Anthony et ... ith Deep Learning and Tree S
Bellema... loration and intrinsic m
Buesi... enerative Models fo
Burg... ecomposition an
By... 2019.
Ch... ent environm
C... ptimization f
C... rcement lea
d...
C... rcement Le
De... ochastic dyn
Du... sed Models. C
Dub... stigating human
Ebert... sed deep RL for visi
Ecoffet... for hard-exploration pr
Edwards, ... Backward Reinforcement Lea
Ellis et al. (20... ogram Synthesis with a REPL. Neur
Eysenbach, Salakh... 2019). Search on the replay buffer: Bridging pla... S.
Farquhar et al (2017). TreeQN and ATreeC: Differentiable Tree-Structured Models for Deep RL. ICLR 2018.
Fazeli et al. (2019). See, feel, act: Hierarchical learning for complex manipulation skills with multisensory fusion. Science Robotics, 4(26).
Finn & Levine (2017). Deep visual foresight for planning robot motion. ICRA.
Finn, Goodfellow, & Levine (2016). Unsupervised learning for physical interaction through video prediction. NeurIPS.
Fisac et al. (2019). A General Safety Framework for Learning-Based Control in Uncertain Robotic Systems. IEEE Transactions on ...
Gal et al. (20... sian neural network dynamics mo... Learning workshop...
Grill et ... ularized policy optimiz
Gu, L... Deep Q-Learning
Gue... ning. ICML 2019
Ha... 8.
Ha... ors by Laten
H... agination in
S...
H... h with Amo
H... tion-based
He... es by Stochas
Ho... Maximizing Expl
Jade... unsupervised aux
Jang, ... zation with Gumbel-
Janner ... odel-Based Policy Optin
Jurgenson... ramework for Goal-Directed ... on. arXiv.
Kidambi et al (2020... Based Offline Reinforcement Learning. arXiv.
Konidaris, Kaelbling, & Lozano-Pérez (2018). From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. JAIR.
Kurutach et al. (2018). Learning Plannable Representations with Causal InfoGAN. NeurIPS.

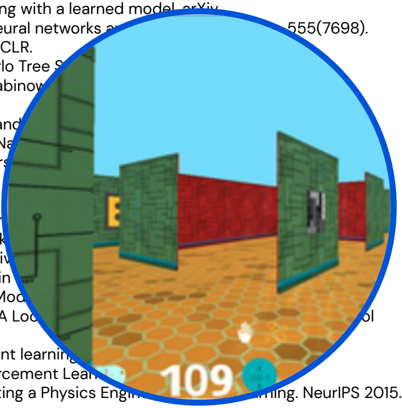




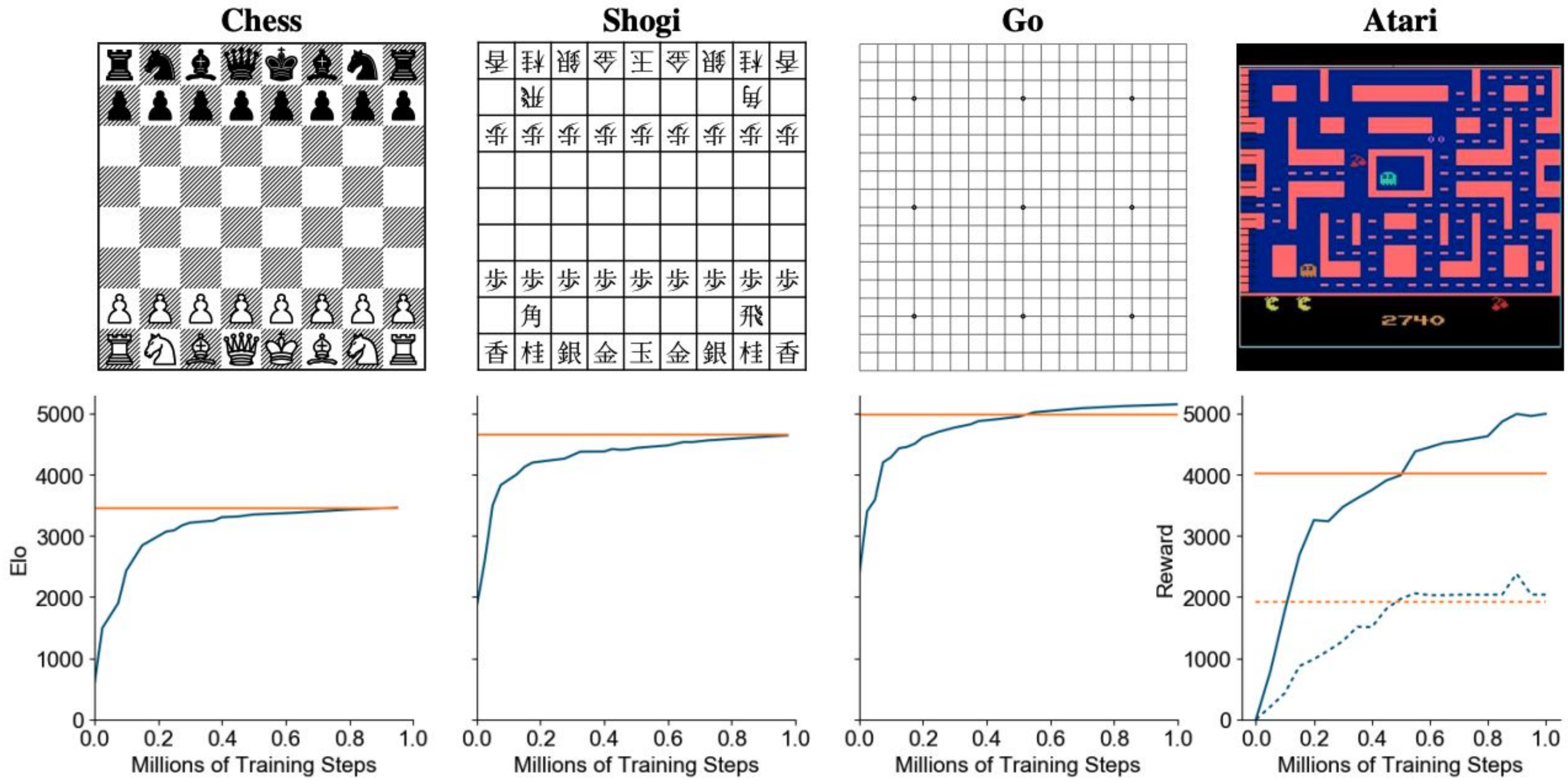
Laskin, Emmons, Jain, Kurutach, Abbeel, & Pathak (2020). Sparse Graphical Memory for Robust Planning. arXiv.
Levine, Wagener, & Abbeel (2015). Learning Contact-Rich Manipulation Skills with Guided Policy Search. ICRA 2015.
Levine et al (2020). Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. arXiv.
Lin et al (2020). Model-b... ta-Reinforcement Learning. arXiv.
Lowrey et al. (2019). P... ient Learning and Exploration via ... 9.
Lu, Mordatch, & A... ng for Continual Lifelong L
Maddison, Mni... ICLR 2017.
Mordatch et ... motion planning th
Mordatch e... ex Characters w
Nagaband... Dexterous Ma
Nagaban... orld Enviro
Nair, Bab... as Self-Su
Nair, Pon... agined Goa
Nasirian... . NeurIPS.
Oh, Guo, ... Prediction
Oh et al. ...
OpenAI et ... on. Internatio
OpenAI et ... d. arXiv.
Osband et a... Reinforcement Lea
Parascandolo, ... er Monte Carlo Tree Se
Pascanu, Li, et al. ... ng from scratch. arXiv.
Pathak et al. (2017). ... y self-supervised prediction. ICML.
Peters, Mulling, & Altun (2... py Policy Search. AAAI 2010.
Rajeswaran et al. (2017). EPOpt: Learning Robust Neural Network Policies Using Model Ensembles. ICLR 2017.
Rajeswaran et al. (2020). A Game Theoretic Framework for Model Based Reinforcement Learning. arXiv.
Sadigh et al. (2016). Planning for autonomous cars that leverage effects on human actions. RSS 2016.
Sanchez-Gonzalez et al. (2018). Graph Networks as Learnable Physics Engines for Inference and Control. ICML 2018.
Savinov, Dosovitskiy, & Koltun (2018). Semi-parametric topological memory for navigation. ICLR 2018.
Schrittwieser et al. (2019). Mastering Atari, Go, Chess and Shogi by planning with a learned model. arXiv.
Segler, Preuss, & Waller (2... al syntheses with deep neural networks a... 555(7698).
Sharma et al. (2020... d Discovery of Skills. ICLR.
Shen et al. (2019... s using Monte Carlo Tree S
Silver, van Hass... nold, Reichert, Rabinow
End-To-End ...
Silver et al. ... al networks and
Silver et al ... nowledge. Na
Silver et a... that masters
Sutton ar... ion.
Tamar et...
Tamar et ... MPC Impr
Tzeng et ... s with Weak
van den O... th Contrastiv
van Hassel... tric models in
Veerapanen... on in Visual Mod
Watter, Spring... ed to Control: A Loc... ol from Raw Image...
Weber et al. (2017... r deep reinforcement learning
Williams et al. (2017). ... r Model-Based Reinforcement Lea
Wu et al. (2015). Galileo: Per... Object Properties by Integrating a Physics Eng... ning. NeurIPS 2015.
Yu et al (2020). MOPO: Model-based Offline Policy Optimization. arXiv.
Zhang, Lerer, et al. (2018). Composable Planning with Attributes. ICML 2018.





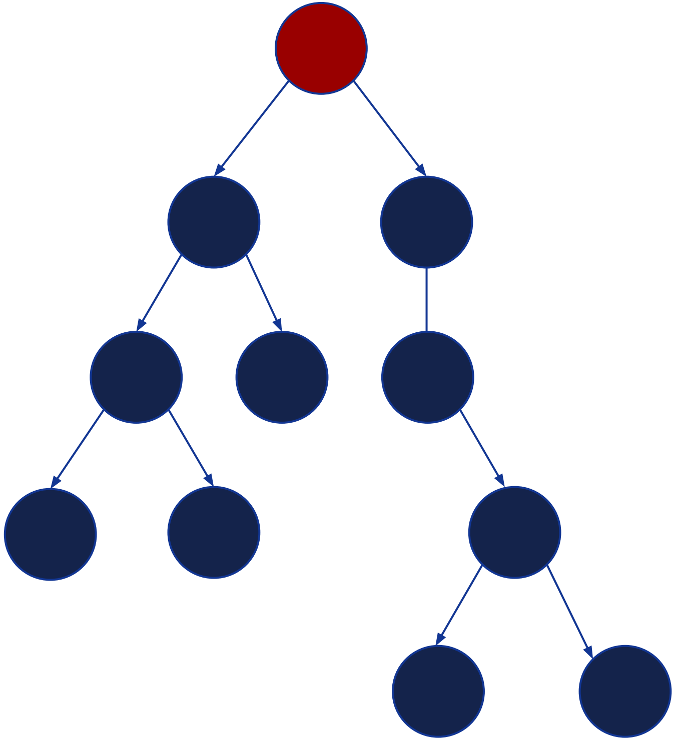


MuZero (Schrittwieser et al., 2019)



{Fabio Viola (@fabiointheuk), Theophane Weber (@theophaneweber)} Join the discussion on Twitter (#JAXecosystem)

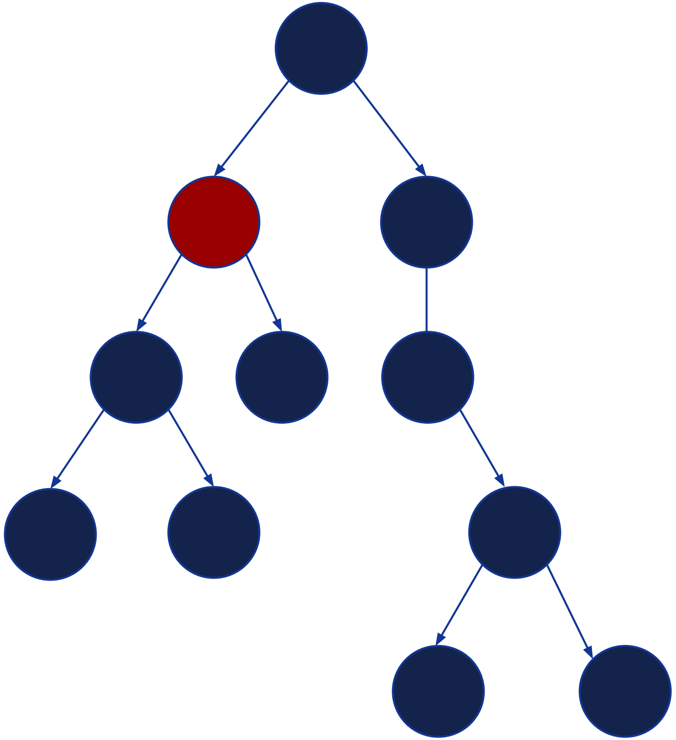
Neural-network guided MCTS in muzero



Neurally-guided MCTS:

1. Traverse tree using chosen heuristic In (alpha/mu)-zero, we use PUCT, which picks nodes with highest score, where the score combines policy prior, action values, and exploration bonus (derived from visit counts)

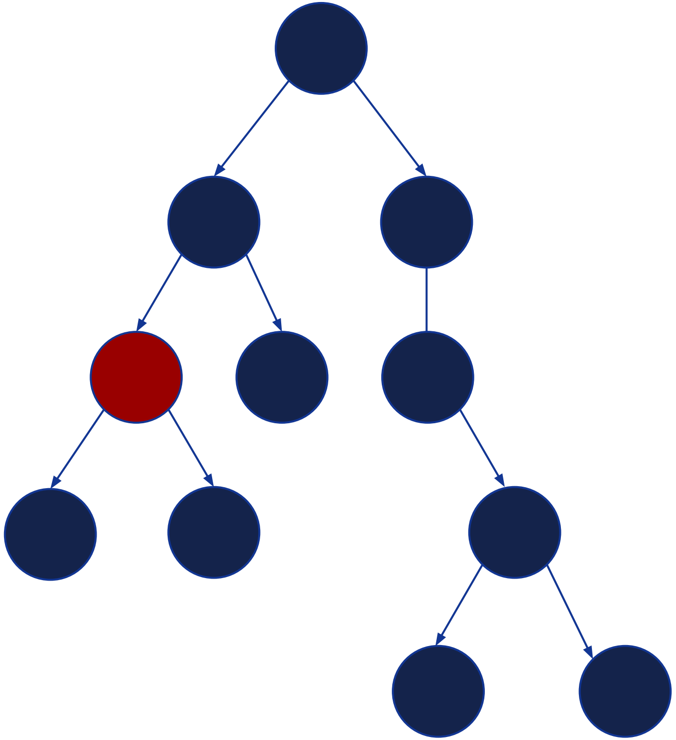
Neural-network guided MCTS in muzero



Neurally-guided MCTS:

1. Traverse tree using chosen heuristic
   In (alpha/mu)-zero, we use PUCT, which picks nodes with highest score, where the score combines policy prior, action values, and exploration bonus (derived from visit counts)

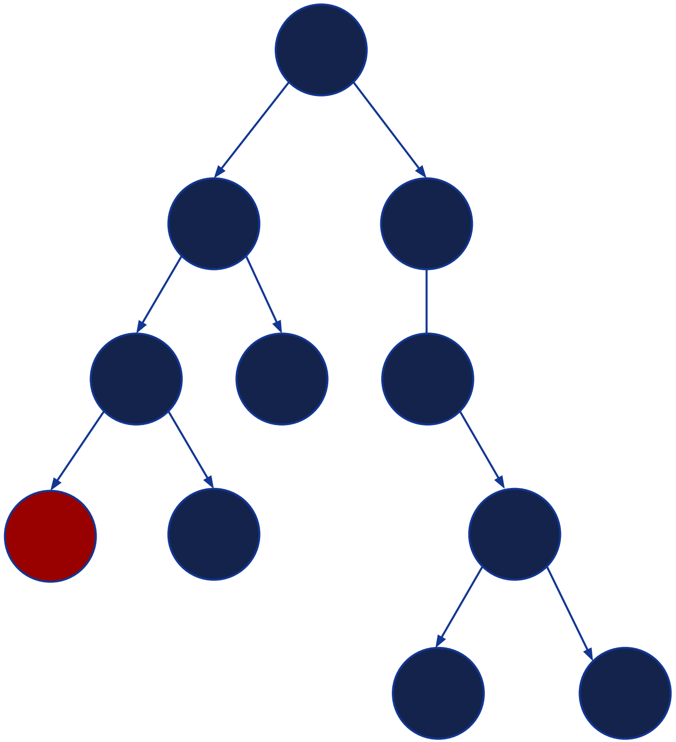
Neural-network guided MCTS in muzero



Neurally-guided MCTS:

1. Traverse tree using chosen heuristic
In (alpha/mu)-zero, we use PUCT, which picks nodes with highest score, where the score combines policy prior, action values, and exploration bonus (derived from visit counts)

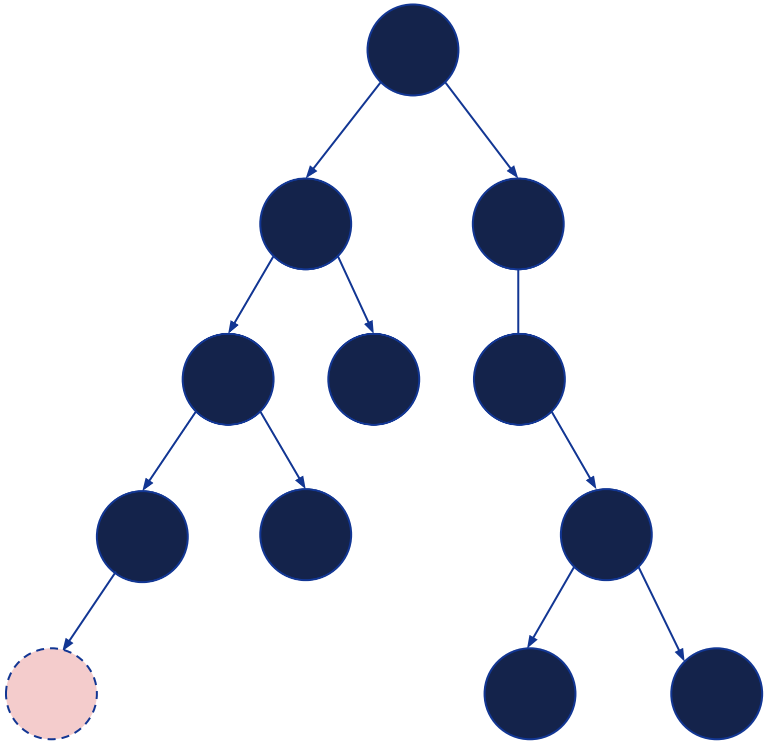
Neural-network guided MCTS in muzero



Neurally-guided MCTS:

1. Traverse tree using chosen heuristic
In (alpha/mu)-zero, we use PUCT, which picks nodes with highest score, where the score combines policy prior, action values, and exploration bonus (derived from visit counts)

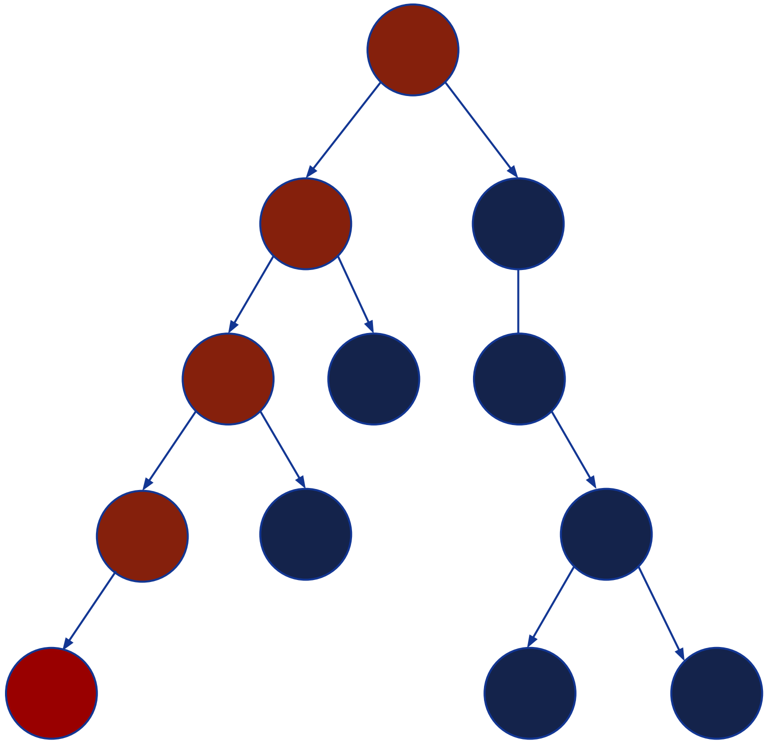
Neural-network guided MCTS in muzero



Neurally-guided MCTS:

1. Traverse tree using chosen heuristic In (alpha/mu)-zero, we use PUCT, which picks nodes with highest score, where the score combines policy prior, action values, and exploration bonus (derived from visit counts)
2. Expand node:
   a. Compute state transition, state value, and policy prior by calling model (neural network)
   b. Add node to tree

Neural-network guided MCTS in muzero



Neurally-guided MCTS:

1. Traverse tree using chosen heuristic In (alpha/mu)-zero, we use PUCT, which picks nodes with highest score, where the score combines policy prior, action values, and exploration bonus (derived from visit counts)
2. Expand node:
   a. Compute state transition, state value, and policy prior by calling model (neural network)
   b. Add node to tree
3. Backward step: propagate information from new leaf node to all ancestors in tree

Why is implementing efficient MCTS a challenging task?

- Some researchers don't want to use C++ day-to-day, and prefer higher level languages, like python
- Performing MCTS in batch in plain python can be slow
- Furthermore, vanilla MCTS is a essentially a sequential algorithm - each sim depends on the results of the previous sims - putting further constraints on how to parallelize computation*

One possible approach:

- Rely on just in time compilation to bridge the gap between interpreted and compiled languages - well aligned with the programming paradigm of JAX!

Why implementing search in JAX?

Expected advantages:

- Still performant once jitted and applied to batched data
- Save costs of moving data in and out of the accelerators
- Allows to easily jit and batch both acting and learning of RL agents
- Easiness to write and modify search components*
  - it's just JAX numpy
  - write for single batch element, use vmap to vectorize
  - nice to be able to inspect your algorithm with python workflow
- Potentially differentiable all the way through

**mileage may vary*

Why implementing search in JAX?

Expected disadvantages:

- Likely less efficient if no batches (e.g. if deploying a trained RL agent in a single environment setup)
- Use some of the accelerator compute and memory is used for the search (rather than just reserving all of it for inference)
- Search depth limited by accelerator memory
- Performance of concurrently running multiple searches will be constrained by slowest instance

**mileage may vary*

Code snippets: search

```
...

def search(self, params, rng_key, root, num_simulations, discount):

  def body_fun(sim, loop_state):
    rng_key, params, tree = loop_state
    rng_key, simulate_key, expand_key = jax.random.split(rng_key, 3)
    leaf_indices, unexplored_actions = simulate(
        simulate_key, tree, self._action_selection_fn, self._max_depth)
    leaf_index = sim + 1
    tree = expand(
        params, expand_key, tree, self._recurrent_fn, leaf_indices,
        unexplored_actions, leaf_index)
    tree = backward(tree, leaf_index)
    loop_state = rng_key, params, tree
    return loop_state

  tree = self._init_tree(root, discount)
  rng_key, _, tree = jax.lax.fori_loop(
      0, num_simulations, body_fun, (rng_key, params, tree))

  return tree.search_result()

...
```

Code snippet: node expansion

```
...

def expand(
    params, rng_key, tree, recurrent_fn, node_indices, actions, next_node_index):

  embeddings = tree.embeddings[tree.batch_range, node_indices]
  step, embeddings = recurrent_fn(params, rng_key, actions, embeddings)
  tree = update_node(
      tree, next_node_index, step.prior_probs, step.values, embeddings)

...
```

DeepMind

6. Questions & Debate

DeepMind

Thank you! 🙌

Please make sure to share your JAX projects on social media using the hashtag:

#JAXecosystem