Boosting Reinforcement Learning With Elixir

Ricardo Corral-Corral
@doctorcorral

VP of Engineering
Chief Data Scientist at Suggestic
https://www.suggestic.com/

|> Stockholm
May, 2019

CODE BEAM STO

#CodeBEAMSTO

RL

|> Reinforcement Learning

|> Why Elixir?

|> implementation examples

Reinforcement

Reinforcement learning is a computational approach to understanding goal-directed learing and decision making. It is distinguished from other computational approaches by its emphasis on learning by an **agent** from direct interaction with the **environment**.

Reinforcement Learning
Richard S. Sutton, Andrew G. Barto

Learning

#CodeBEAMSTO

Reinforcement Learning

Crédito: **RISELab - UC Berkeley**

#CodeBEAMSTO

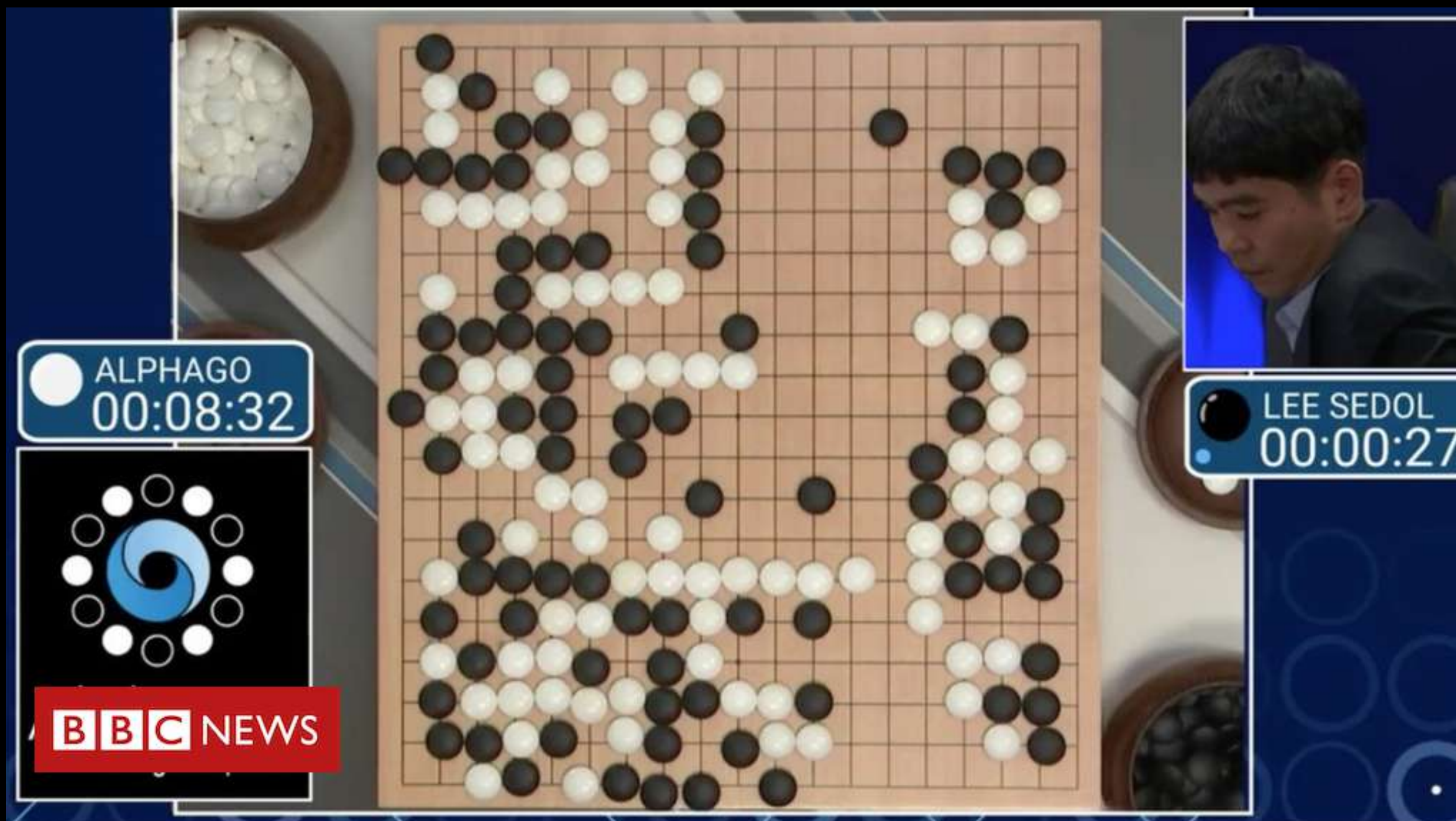
Reinforcement

Learning

Crédito: @**simoninithomas**

#CodeBEAMSTO

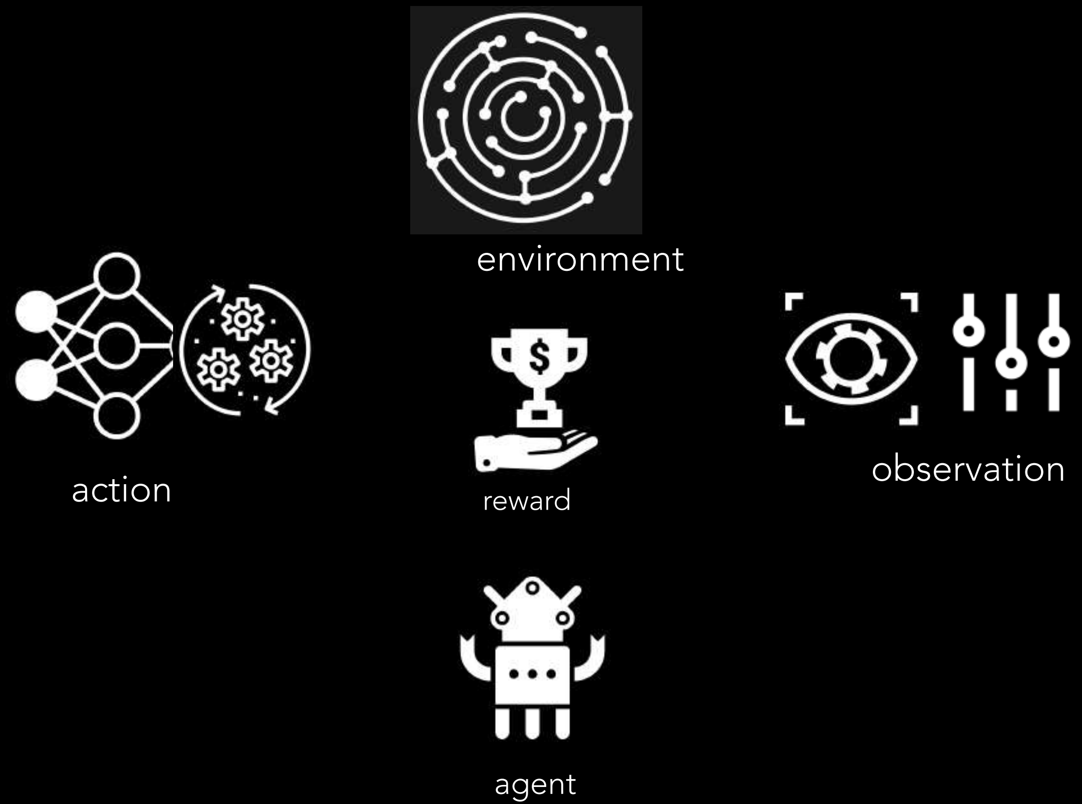
Reinforcement Learning



Google DeepMind Challenge Match
AlphaGo versus Lee Sedol
9 -15 March 2016

#CodeBEAMSTO

Reinforcement



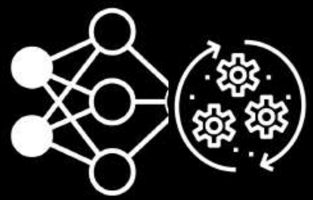
Learning

- Agent is the component that takes decisions
- Environment rewards the agent by its actions
- Agent observes environment and its changes
- Agent learns from experience

#CodeBEAMSTO

Markov Decision Process

Ony current state is considered for taking a decision - past states and actions are ignored.



action

$$q_\pi(s, a)$$

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

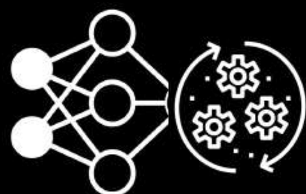
agent

Obtained knowledge can be used for estimate the future reward (*expected cumulative future discounted reward*)

$$q_\pi(s, a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a)(r + \gamma \sum_{a' \in \mathcal{A}(s')} \pi(a' | s') q_\pi(s', a'))$$

Ony current state is considered for taking a decision - past states and actions are ignored.

Markov Decision Process



action

$$q_\pi(s, a)$$

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

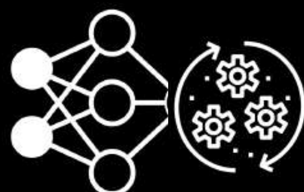
agent

Obtained knowledge can be used for estimate the future reward (*expected cumulative future discounted reward*)

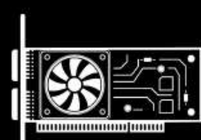
$$\boxed{q_*(s, a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a)(r + \gamma \max_{a' \in \mathcal{A}(s')} q_*(s', a'))}$$

Bellman's principle of optimality

Artificial Neural Networks



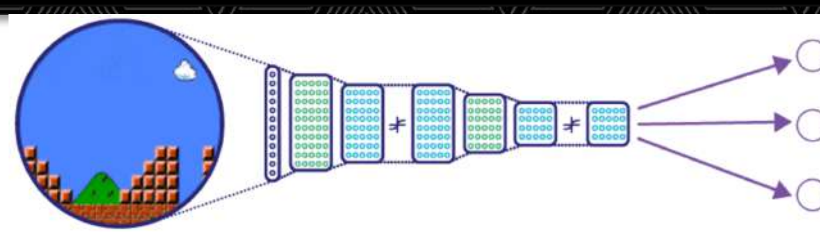
action



GPU intensive

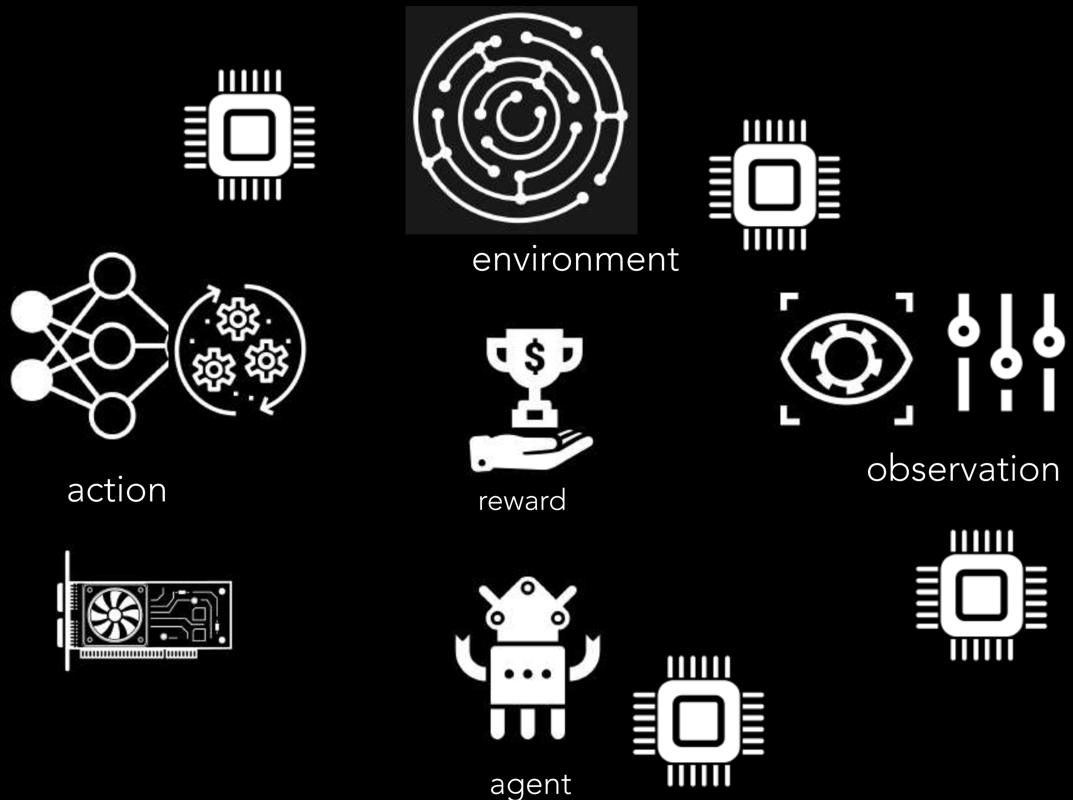
agent

Deep Q-Learning



State | dense and conv layers | Q values / policy

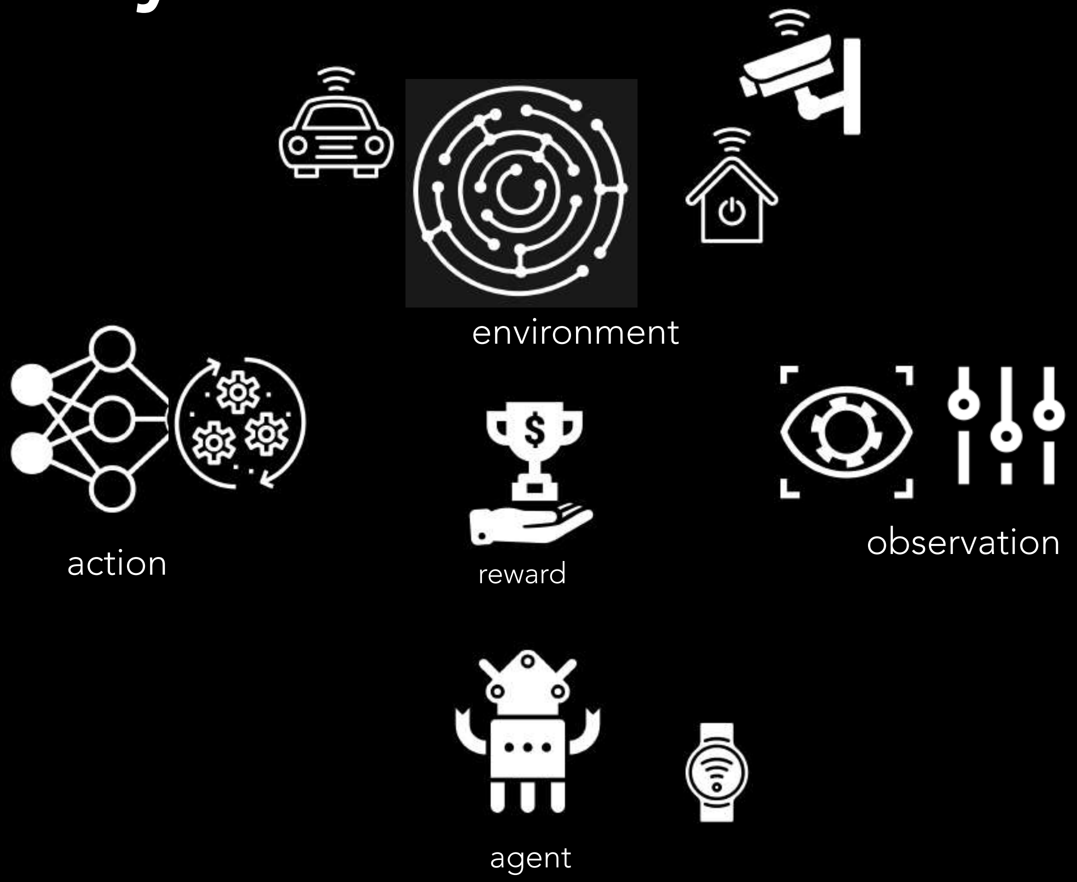
Why Elixir?



CPU intensive

The **environment** representation and its **reward** logic, as well as the **environment** state processing and transformation are CPU intensive tasks

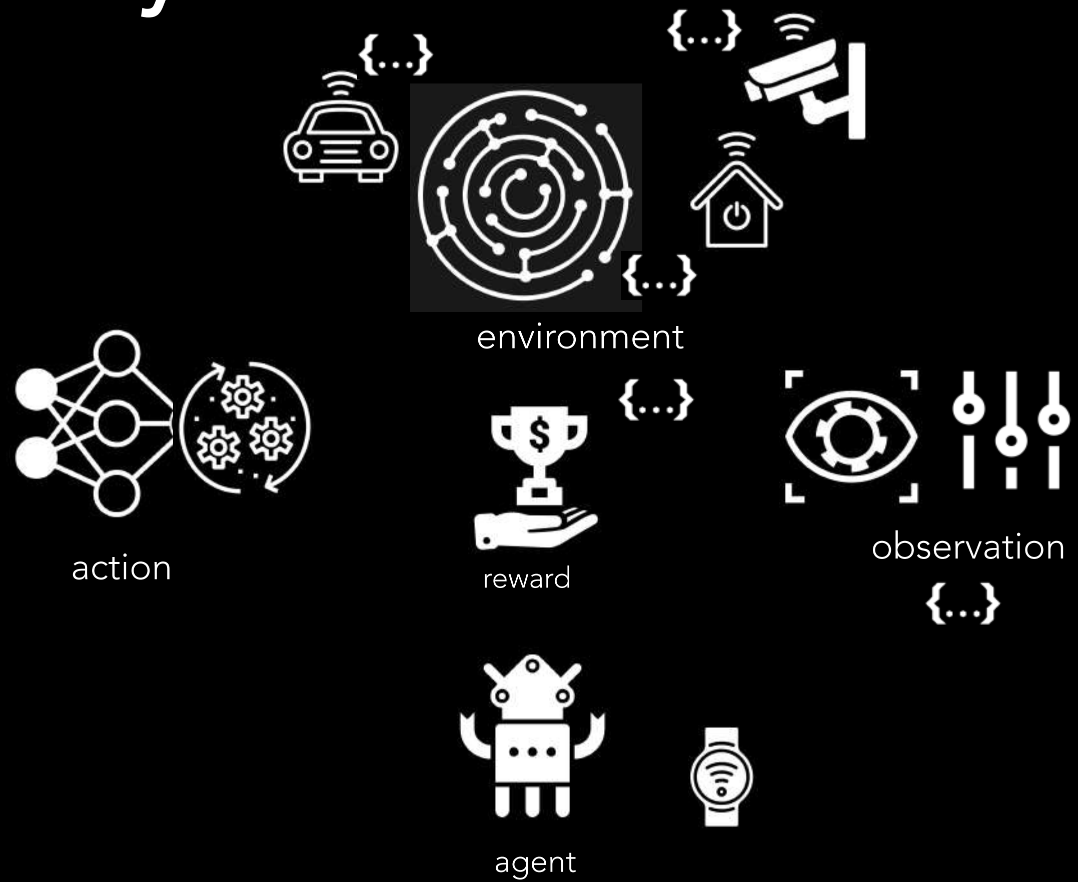
Why Elixir?



IoT

Implementation on real world physical systems require diverse interconnected computing units

Why Elixir?



Actor model

Facilidades de comunicación mediante paso de mensajes

Existing tooling

unity
Machine Learning Agents

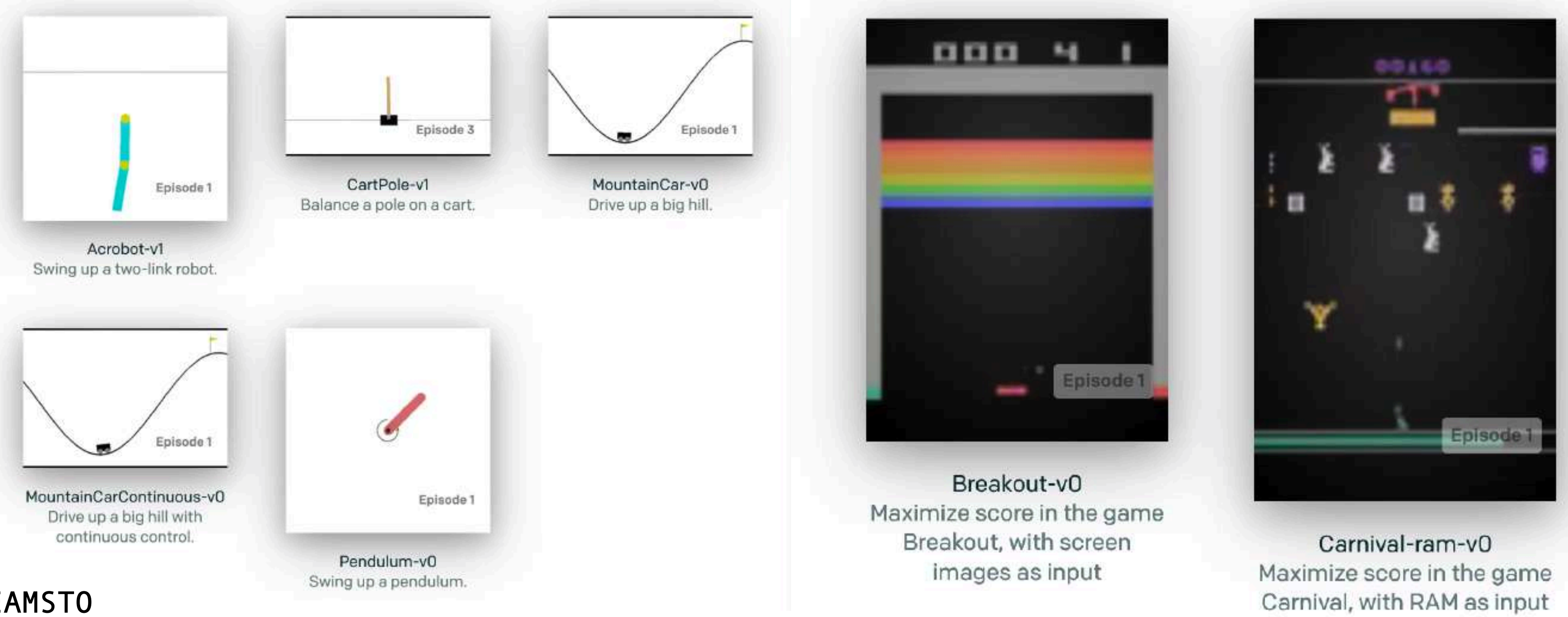
Dopamine

Horizon

OpenAI Gym

OpenAI Gym

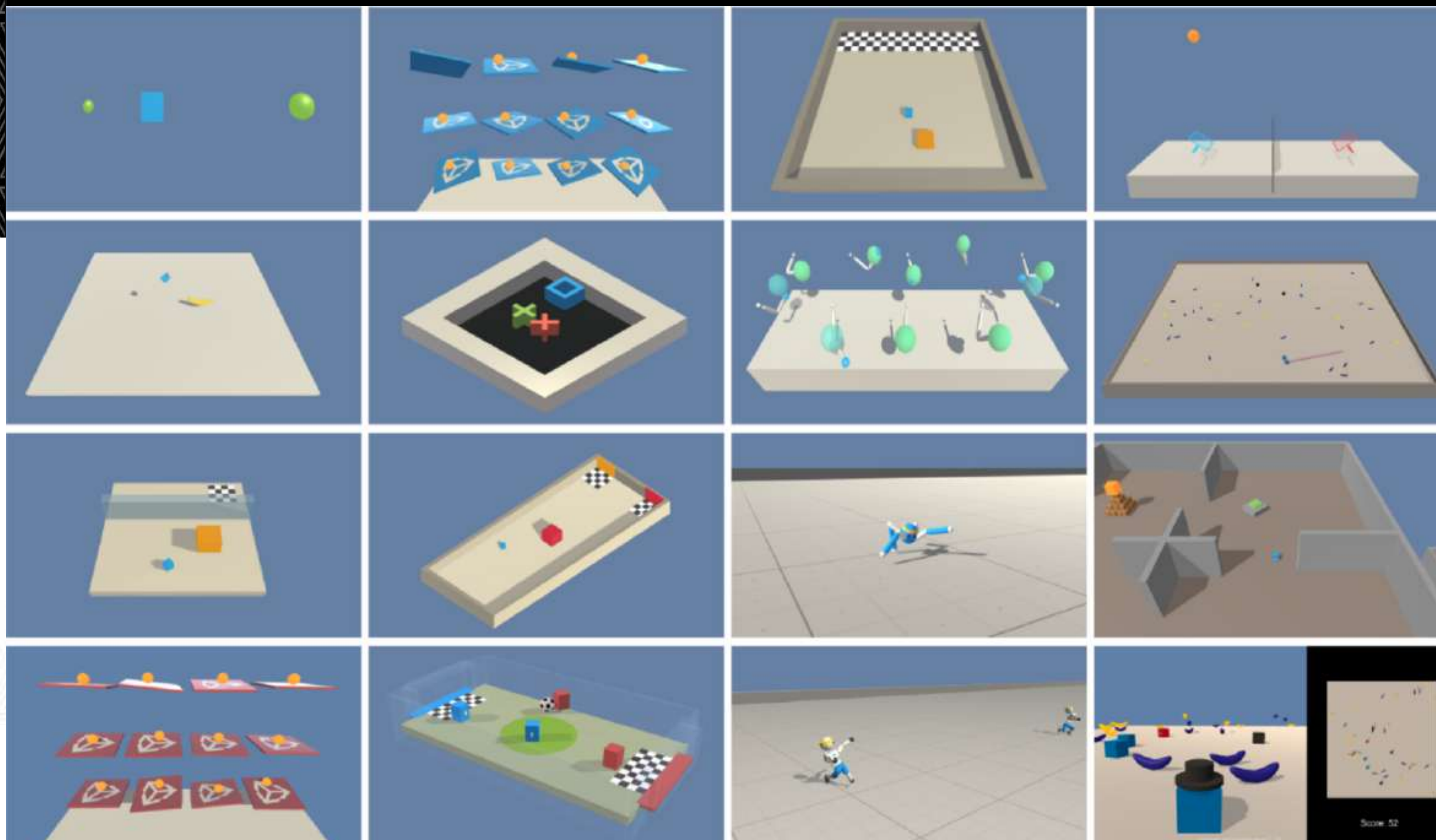
https://github.com/openai/gym



#CodeBEAMSTO

ML-Agents Toolkit

https://github.com/Unity-Technologies/ml-agents



Google Dopamine

https://github.com/google/dopamine

```
python -um dopamine.discrete_domains.train \
  --base_dir=/tmp/dopamine \
  --gin_files='dopamine/agents/dqn/configs/dqn.gin'
```

```
lman target value.
mma^N * Q'_t+1

x_a Q(S_t+1, a)
#            (or) 0 if S_t is a terminal state,
# and
#   N is the update horizon (by default, N=1).
return self._replay.rewards + self.cumulative_gamma * replay_next_qt_max * (
    1. - tf.cast(self._replay.terminals, tf.float32))

def _build_train_op(self):
  """Builds a training op.

  Returns:
    train_op: An op performing one step of training from replay data.
  """
  replay_action_one_hot = tf.one_hot(
      self._replay.actions, self.num_actions, 1., 0., name='action_one_hot')
  replay_chosen_q = tf.reduce_sum(
      self._replay_net_outputs.q_values * replay_action_one_hot,
      reduction_indices=1,
      name='replay_chosen_q')

  target = tf.stop_gradient(self._build_target_q_op())
  loss = tf.losses.huber_loss(
      target, replay_chosen_q, reduction=tf.losses.Reduction.NONE)
  if self.summary_writer is not None:
    with tf.variable_scope('Losses'):
      tf.summary.scalar('HuberLoss', tf.reduce_mean(loss))
  return self.optimizer.minimize(tf.reduce_mean(loss))
```

#CodeBEAMSTO

Facebook Horizon

https://github.com/facebookresearch/Horizon

output data table before running using a Hive command.

```
# Clear last run's spark data (in case of interruption)
rm -Rf spark-warehouse derby.log metastore_db preprocessing/spark-warehouse preprocessing/metastore_db preprocessi
```

Now that we are ready, let's run our spark job on our local machine. This will produce a massive amount of logging (because we are running many systems that typically are distributed across many nodes) and there will be some exception stack traces printed because we are running in a psuedo-distributed mode. Generally this is fine as long as the output data is generated:

```
# Run timelime on pre-timeline data
/usr/local/spark/bin/spark-submit \
  --class com.facebook.spark.rl.Preprocessor preprocessing/target/rl-preprocessing-1.1.jar \
  "`cat ml/rl/workflow/sample_configs/discrete_action/timeline.json`"

# Look at the first row of training & eval
head -n1 cartpole_discrete_training/part*

head -n1 cartpole_discrete_eval/part*
```

There are many output files. The reason for this is that Spark expects many input & output files: otherwise it wouldn't be able to efficiently run on many machines and output data in parallel. For this tutorial, we will merge all of this data into a single file, but in a production use-case we would be streaming data from HDFS during training.

```
# Merge output data to single file
mkdir training_data
cat cartpole_discrete_training/part* > training_data/cartpole_discrete_timeline.json
cat cartpole_discrete_eval/part* > training_data/cartpole_discrete_timeline_eval.json

# Remove the output data folder
rm -Rf cartpole_discrete_training cartpole_discrete_eval
```

Now that all of our data has been grouped into consecutive pairs, we can run the normalization pipeline

Elixir
implementation

Gyx.Core

Gyx.Core.Env

```
alias Gyx.Core.Exp

@type initial_state :: Exp.t()
@type observation :: any()
@type action :: any()

@doc "Sets the state of the environment to its default"
@callback reset() :: initial_state()
@doc "Gets an environment representation usable by the agent"
@callback observe() :: observation()
@doc """
Recieves an agent's `action` and responds to it,
informing the agent back with a reward, a modified environment
and a termination signal
"""
@callback step(action()) :: Exp.t() | {:error, reason :: String.t()}

@doc "Retrieves the parameters for current environment state"
@callback get_state() :: any()
```

Gyx.Core.Exp

```
@type t :: %__MODULE__{
        state: state(),
        action: action(),
        reward: float(),
        next_state: state(),
        done: boolean(),
        info: map()
      }
```

Gyx.Core

Gyx.Core.Spaces.Discrete

```elixir
defstruct n: nil, seed: {1, 2, 3}, random_algorithm: :exsplus

@type t :: %__MODULE__{
        n: integer(),
        random_algorithm: :exrop | :exs1024 | :exs1024s | :exs64 | :exsp | :exsplus,
        seed: {integer(), integer(), integer()}
      }
```

```elixir
defmacro __using__(_params) do
  quote do
    @before_compile Gyx.Core.Env
    @behaviour Gyx.Core.Env

    @enforce_keys [:action_space, :observation_space]
```

Gyx.Core.Env

Gyx.Core.Spaces

```elixir
@type space :: Discrete.t() | Box.t() | Tuple.t()
@type discrete_point :: integer
@type box_point :: list(list(float))
@type tuple_point :: list(discrete_point | box_point())
@type point :: box_point | discrete_point | tuple_point

@spec sample(space()) :: {atom(), point()}
def sample(space)

@spec contains?(space(), point()) :: bool()
def contains?(space, point)

defdelegate set_seed(space), to: Gyx.Core.Spaces.Shared
```

Gyx.Environments.FrozenLake

```
iex(1)> Gyx.Environments.FrozenLake.render()
SFFF
FHFH
FFFH
HFFG
{:ok, [position: {0, 0}]}
```

```
iex(2)> Gyx.Environments.FrozenLake.step(1)
%Gyx.Core.Exp{
  action: 1,
  done: false,
  info: %{},
  next_state: %{
    __struct__: Gyx.Environments.FrozenLake,
    action_space: %Gyx.Core.Spaces.Discrete{
      n: 4,
      random_algorithm: :exsplus,
      seed: {1, 2, 3}
    },
    col: 0,
    enumerated: 0,
    map: ["SFFF", "FHFH", "FFFH", "HFFG"],
    ncol: 4,
    nrow: 4,
    observation_space: %Gyx.Core.Spaces.Discrete{
      n: 16,
      random_algorithm: :exsplus,
      seed: {1, 2, 3}
    },
    row: 0
  },
  reward: 0.0,
  state: %{
```

```elixir
def init(map_name) do
  map = @maps[map_name]

  {:ok,
   %__MODULE__{
     map: map,
     row: 0,
     col: 0,
     nrow: length(map),
     ncol: String.length(List.first(map)),
     action_space: %Discrete{n: 4},
     observation_space: %Discrete{n: 16}
   }}
end

def start_link(_, opts) do
  GenServer.start_link(__MODULE__, "4x4", opts)
end

@impl Env
def reset() do
  GenServer.call(__MODULE__, :reset)
end

def render() do
  GenServer.call(__MODULE__, :render)
end

def handle_call(:render, _from, state) do
  printEnv(state.map, state.row, state.col)
  {:reply, {:ok, position: {state.row, state.col}}, state}
end

@impl true
def handle_call(:reset, _from, state) do
  new_env_state = %{state | row: 0, col: 0}
  {:reply, %Exp{next_state: new_env_state}, new_env_state}
end

def handle_call({:act, action}, _from, state) do
  new_state = rwo_col_step(state, action)
  current = get_position(new_state.map, new_state.row, new_state.col)

  {:reply,
   %Exp{
     state: env_state_transformer(state),
     action: action,
     next_state: env_state_transformer(new_state),
     reward: if(current == "G", do: 1.0, else: 0.0),
     done: current in ["H", "G"],
     info: %{}
   }, new_state}
```

#CodeBEAMSTO

Gyx.Gym.Environment

```
iex(1)> Gyx.Gym.Environment.make("FrozenLake-v0")
🐍🐍🐍-- Imporing Gym environment from Python:
⏩⏩⏩-- b'FrozenLake-v0'
{:"$erlport.opaque", :python,
 <<128, 2, 99, 110, 117, 109, 112, 121, 46, 99, 111, 114, 101, 46, 109, 117
   108, 116, 105, 97, 114, 114, 97, 121, 10, 115, 99, 97, 108, 97, 114, 10,
   0, 99, 110, 117, 109, 112, 121, 10, 100, 116, 121, 112, 101, 10, ...>>}
iex(2)> Gyx.Gym.Environment.render()

SFFF
FHFH
FFFH
HFFG
{:"$erlport.opaque", :python,
 <<128, 2, 99, 110, 117, 109, 112, 121, 46, 99, 111, 114, 101, 46, 109, 117
   108, 116, 105, 97, 114, 114, 97, 121, 10, 115, 99, 97, 108, 97, 114, 10,
   0, 99, 110, 117, 109, 112, 121, 10, 100, 116, 121, 112, 101, 10, ...>>}
iex(3)> Gyx.Gym.Environment.step(1)
%Gyx.Core.Exp{
  action: 1,
  done: false,
  info: %{
    gym_info: {:"$erlport.opaque", :python,
     <<128, 2, 125, 113, 0, 88, 4, 0, 0, 0, 112, 114, 111, 98, 113, 1, 71,
       213, 85, 85, 85, 85, 85, 85, 115, 46>>}
  },
  next_state: 1,
  reward: 0.0,
  state: {:"$erlport.opaque", :python,
   <<128, 2, 99, 110, 117, 109, 112, 121, 46, 99, 111, 114, 101, 46, 109, 1
     108, 116, 105, 97, 114, 114, 97, 121, 10, 115, 99, 97, 108, 97, 114, 1
     113, 0, 99, 110, 117, 109, 112, 121, 10, ...>>}
}
```

```elixir
defstruct env: nil,
          current_state: nil,
          session: nil,
          action_space: nil,
          observation_space: nil

@type space :: Discrete.t() | Box.t() | Tuple.t()
@type t :: %__MODULE__{
        env: any(),
        current_state: any(),
        session: pid(),
        action_space: space(),
        observation_space: space()
      }

@impl true
def init(_) do
  python_session = Python.start()
  Logger.warn("Gym environment not associated yet with current #{__MODULE__} process")
  Logger.info("In order to assign a Gym environment to this process,
  please use #{__MODULE__}.make(ENVIRONMENTNAME)\n")

  {:ok,
   %__MODULE__{
     env: nil,
     current_state: nil,
     session: python_session,
     action_space: nil,
     observation_space: nil
   }}
end

def start_link(_, opts) do
  GenServer.start_link(__MODULE__, %__MODULE__{action_space: nil, observation_space: nil}, opts)
end

def render() do
  GenServer.call(__MODULE__, :render)
end

def make(environment_name) do
  GenServer.call(__MODULE__, {:make, environment_name})
end

@impl true
def step(action) do
  GenServer.call(__MODULE__, {:act, action})
end

@impl true
def reset() do
  GenServer.call(__MODULE__, :reset)
end

def handle_call({:make, environment_name}, _from, state) do
  {env, initial_state, action_space, observation_space} =
```

#CodeBEAMSTO

Gyx.Agents

```elixir
def td_learn(sarsa) do
  GenServer.call(__MODULE__, {:td_learn, sarsa})
end

def handle_call(
      {:td_learn, {s, a, r, ss, aa}},
      _from,
      state = %{Q: qtable, alpha: alpha, gamma: gamma}
    ) do
  predict = qtable.q_get(s, a)
  target = r + gamma * qtable.q_get(ss, aa)
  expected_return = predict * (1-alpha) + target * alpha
  qtable.q_set(s, a, expected_return)
  {:reply, expected_return, state}
end
```

Gyx.Qstorage

```
iex(5)> Gyx.Qstorage.QGenServer.get_q
%{
  "0" => %{
    0 => 0.0019724988180791552,
    1 => 0.0011373872220860517,
    2 => 0.002130124695756522,
    3 => 0.0025732867557405844
  },
  "1" => %{
    0 => 4.307685923561203e-4,
    1 => 1.3004530595506697e-4,
    2 => 0.0016973683178757952,
    3 => 0.0011164073830185624
  },
  "10" => %{
    0 => 0.0012787082054930253,
    1 => 0.004795837104754758,
    2 => 0.0010135419945296069,
    3 => 0.005758543602934572
  },
```

#CodeBEAMSTO

Gyx.Core.ReplayMemory

```elixir
@type experience :: Gyx.Core.Exp.t()
@type experiences :: list(experience)
@type sampling_type :: :random | :latest
@type batch_size :: integer()

@callback add(experience()) :: :ok | {:error, reason :: String.t()}

@callback get_batch({batch_size(), sampling_type()}) :: experiences()

defmacro __using__(_params) do
  quote do
    @behaviour Gyx.Core.ReplayMemory

    @enforce_keys [:replay_capacity]
```

Gradient Based methods work best with i.i.d samples
(independent and identically distributed)

Replay Memory decouples learning from environment exploration

It is hard to evaluete efectiveness of a policy if it changes at the same time

Distributed Prioritized Experience Replay

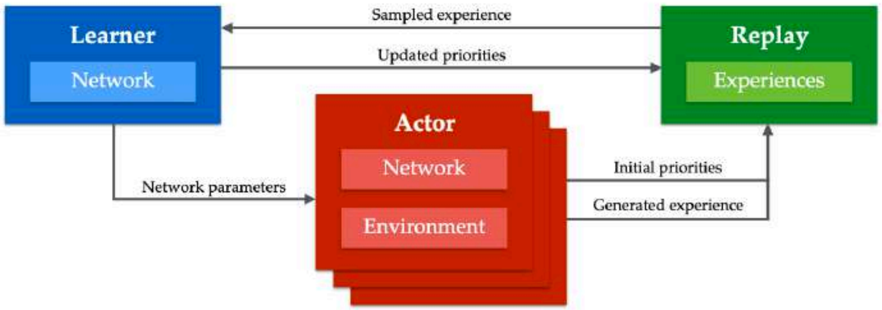
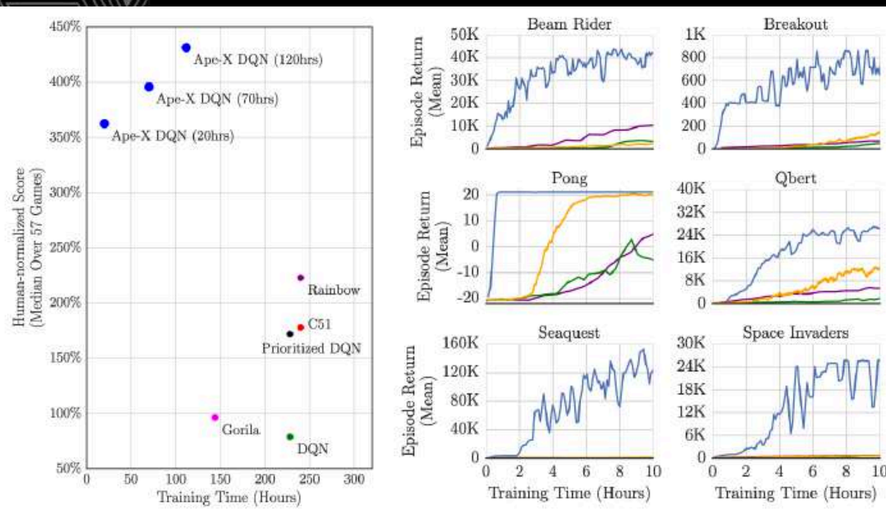


Figure 1: The Ape-X architecture in a nutshell: multiple actors, each with its own instance of the environment, generate experience, add it to a shared experience replay memory, and compute initial priorities for the data. The (single) learner samples from this memory and updates the network and the priorities of the experience in the memory. The actors' networks are periodically updated with the latest network parameters from the learner.



Published as a conference paper at ICLR 2018

DISTRIBUTED PRIORITIZED EXPERIENCE REPLAY

Dan Horgan
DeepMind
horgan@google.com

John Quan
DeepMind
johnquan@google.com

David Budden
DeepMind
budden@google.com

Gabriel Barth-Maron
DeepMind
gabrielbm@google.com

Matteo Hessel
DeepMind
mtthss@google.com

Hado van Hasselt
DeepMind
hado@google.com

David Silver
DeepMind
davidsilver@google.com

ABSTRACT

We propose a distributed architecture for deep reinforcement learning at scale, that enables agents to learn effectively from orders of magnitude more data than previously possible. The algorithm decouples acting from learning: the actors interact with their own instances of the environment by selecting actions according to a shared neural network, and accumulate the resulting experience in a shared experience replay memory; the learner replays samples of experience and updates the neural network. The architecture relies on prioritized experience replay to focus only on the most significant data generated by the actors. Our architecture substantially improves the state of the art on the Arcade Learning Environment, achieving better final performance in a fraction of the wall-clock training time.

#CodeBEAMSTO

gyx

https://github.com/doctorcorral/gyx

Thank you!

Input: positive integer *num_episodes*, small positive fraction α, GLIE $\{\epsilon_i\}$
Output: policy π ($\approx \pi_*$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
    $\epsilon \leftarrow \epsilon_i$
    $\pi \leftarrow \epsilon$-greedy(Q)
    Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
    for $t \leftarrow 0$ ***to*** $T - 1$ **do**
        if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
            $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$
    end
end
return π

por si las flais