Building Deep Reinforcement Learning Applications on Apache Spark with Analytics Zoo using BigDL

Yuhao Yang

Intel Data Analytics Technologies

Agenda

Analytics Zoo overview

Reinforcement learning overview

Reinforcement learning with Analytics zoo

future directions

Analytics Zoo

- ***Analytics + AI Platform for Apache Spark and BigDL***
  - Open source, Scala/Python, Spark 1.6 and 2.X

Analytics Zoo	High level API, Industry pipelines, App demo & Util
BigDL	MKL, Tensors, Layers, optim Methods, all-reduce
Apache Spark	RDD, DataFrame, Scala/Python

https://github.com/intel-analytics/analytics-zoo

Analytics Zoo

High level pipeline APIs

nnframes: Spark DataFrames and ML Pipelines for DL

Keras-style API

autograd: custom layer/loss using auto differentiation

Transfer learning

Analytics Zoo

Built-in deep learning pipelines & models

Object detection: API and pre-trained SSD and Faster-RCNN

Image classification: API and pre-trained VGG, Inception, ResNet, MobileNet, etc.

Text classification API with CNN, LSTM and GRU

Recommendation API with NCF, Wide and Deep etc.

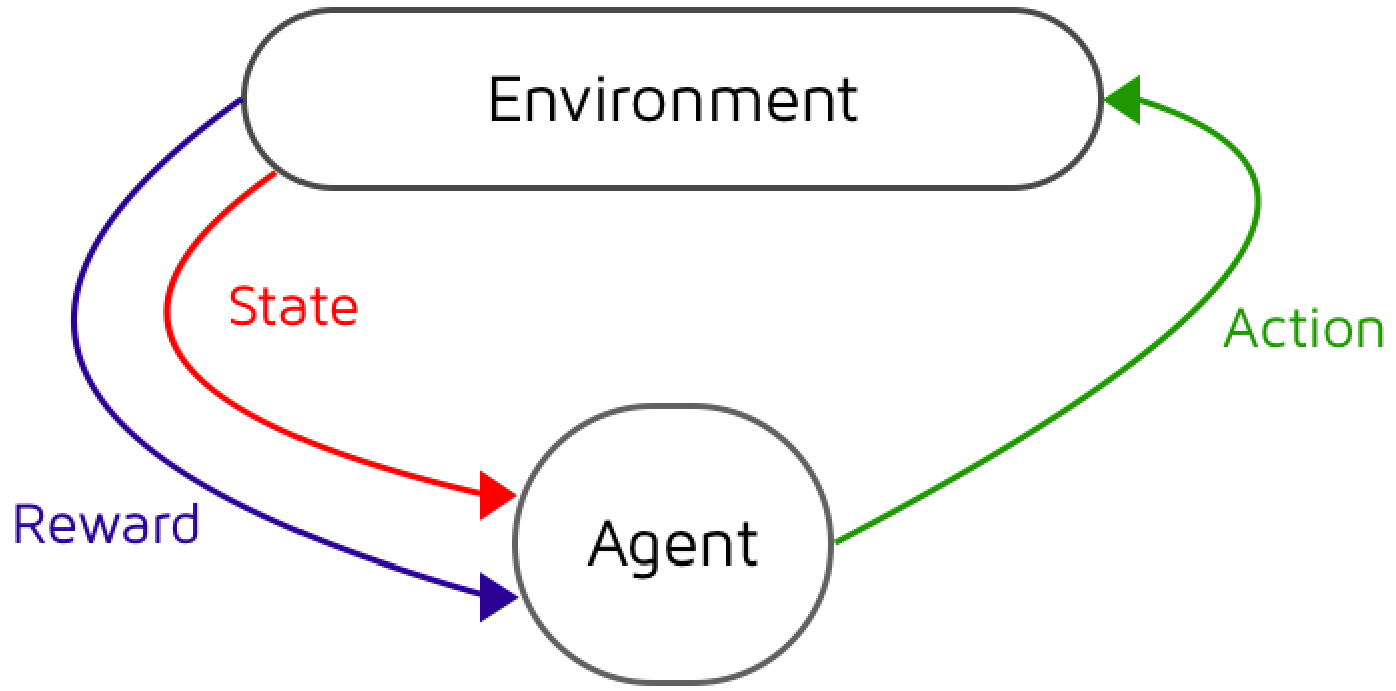
Analytics Zoo

End-to-end reference use cases

- reinforcement learning
- anomaly detection
- sentiment analysis
- fraud detection
- image augmentation
- object detection
- variational autoencoder
- …

Reinforcement Learning (RL)

- RL is for Decision-making

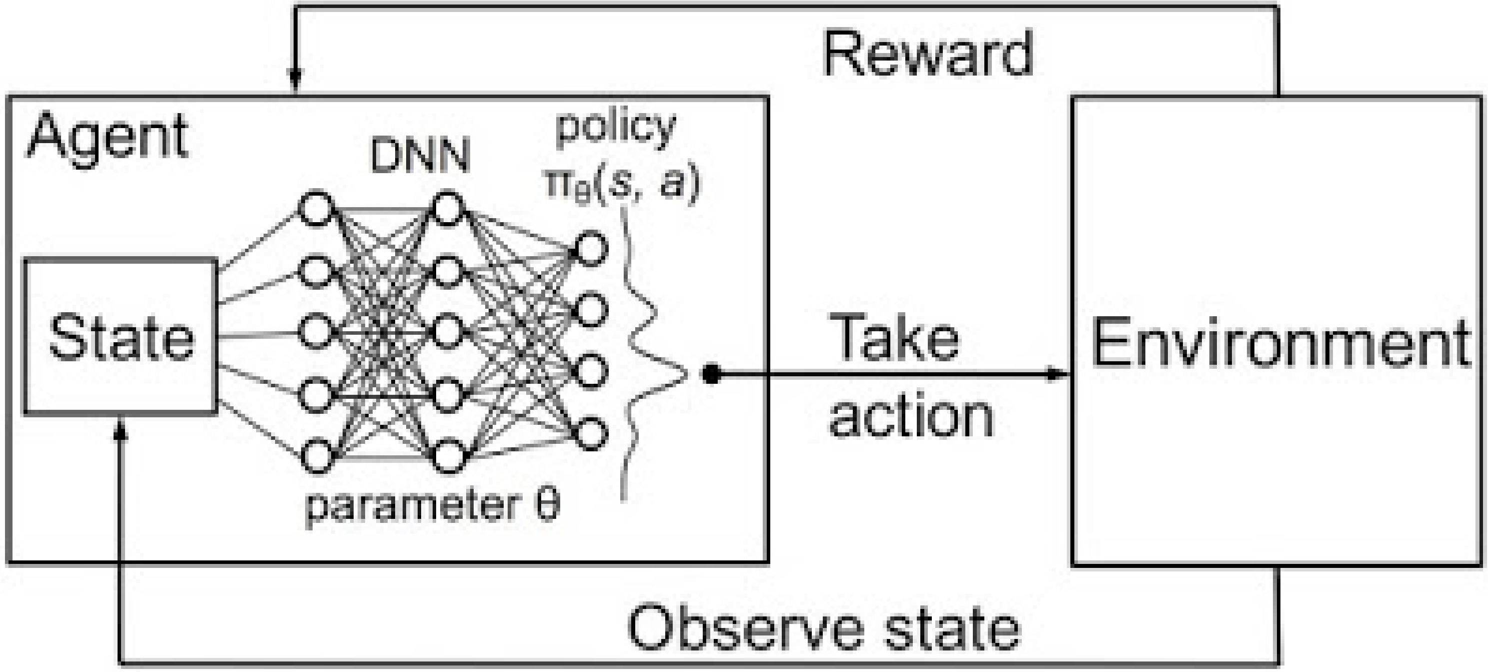


Examples of RL applications

- Play: Atari, poker, Go, ...
- Interact with users: recommend, Healthcare, chatbot, personalize, ..
- Control: auto-driving, robotics, finance, …

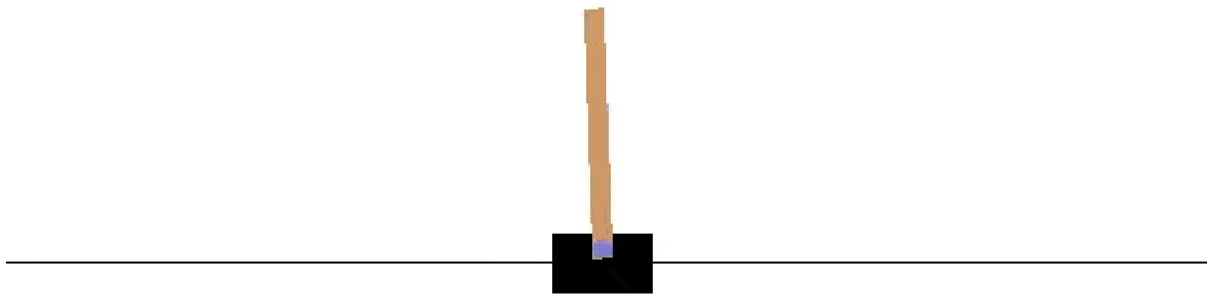
Deep Reinforcement Learning (DRL)

Agents take actions (a) in state (s) and receives rewards (R)
Goal is to find the policy (π) that maximized future rewards



http://people.csail.mit.edu/hongzi/content/publications/DeepRM-HotNets16.pdf

Cartpole



Observation

Type: Box(4)

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Actions

Type: Discrete(2)

Num	Action
0	Push cart to the left
1	Push cart to the right

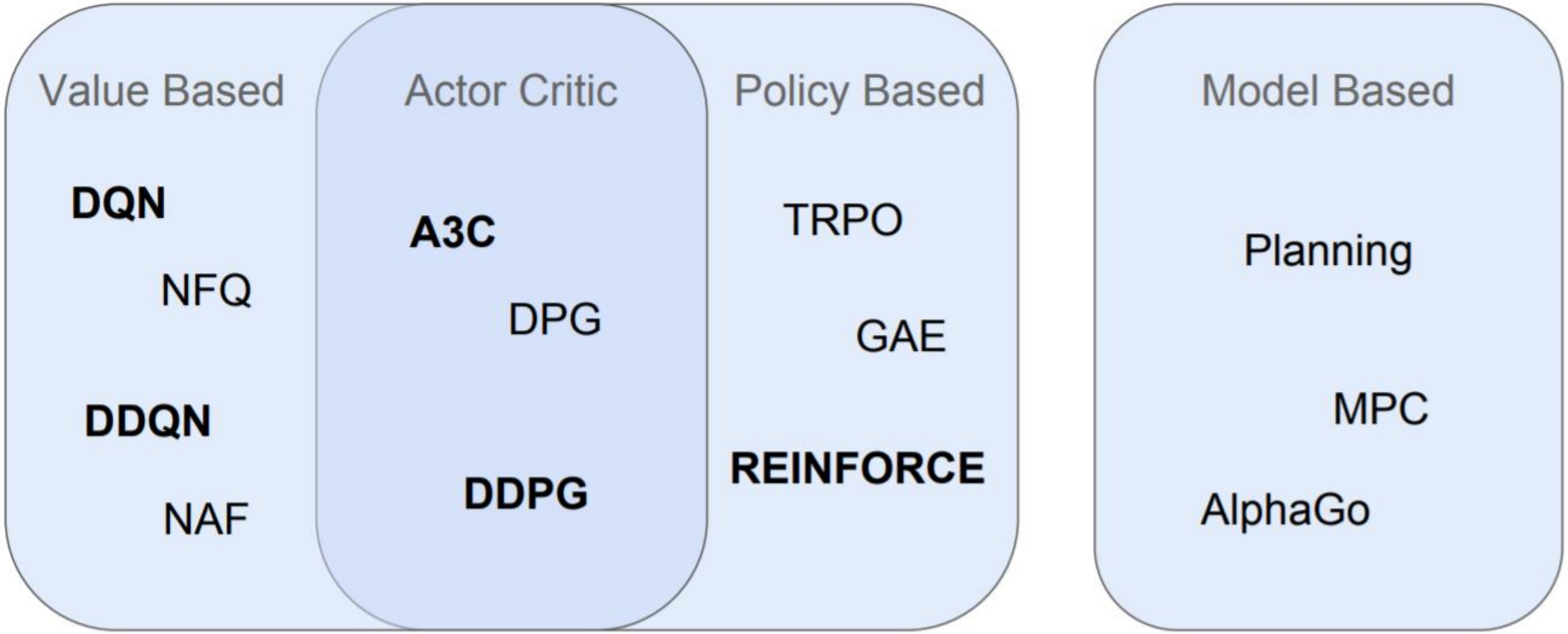
Reward

Reward is 1 for every step taken,

Approaches to Reinforcement Learning

- Value-based RL
  - Estimate the optimal value function Q*(S,A)
  - Output of the Neural network is the value for Q(S, A)
- Policy-based RL
  - Search directly for the optimal policy π*
  - Output of the neural network is the probability of each action.
- Model-based RL

DRL algo



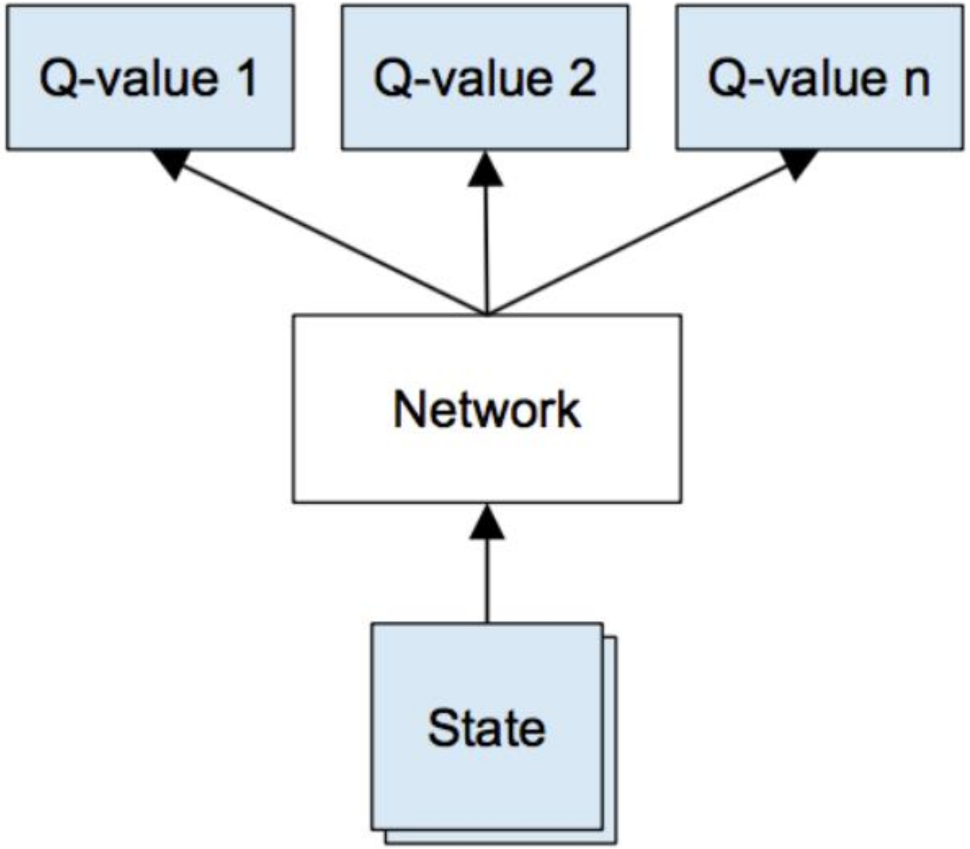
Value Based
DQN
NFQ
DDQN
NAF
Actor Critic
A3C
DPG
DDPG
Policy Based
TRPO
GAE
REINFORCE
Model Based
Planning
MPC
AlphaGo

Examples

- 1. Simple DQN to demo API and train with Spark RDD.

- 2. Distributed REINFORCE

Q-network



https://ai.intel.com/demystifying-deep-reinforcement-learning/

Bellman Equation

- Optimal value maximises over all decisions. Informally:

$$
\begin{aligned}
Q^*(s, a) &= r_{t+1} + \gamma \max_{a_{t+1}} r_{t+2} + \gamma^2 \max_{a_{t+2}} r_{t+3} + \ldots \\
&= r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})
\end{aligned}
$$

- Value iteration algorithms solve the Bellman equation

$$
Q_{i+1}(s, a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q_i(s', a') \mid s, a \right]
$$

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Resources_files/deep_rl.pdf

DQN critical routines

```
for e in range(EPISODES):
    state = env.reset()
    state = np.reshape(state, [1, state_size])
    for time in range(500):
        action = agent.act(state)       ε-greedy action selection
        next_state, reward, done, _ = env.step(action)
        reward = reward if not done else -10
        next_state = np.reshape(next_state, [1, state_size])
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        if len(agent.memory) > batch_size:
            agent.replay(batch_size)
```

Parallelize the neural network training

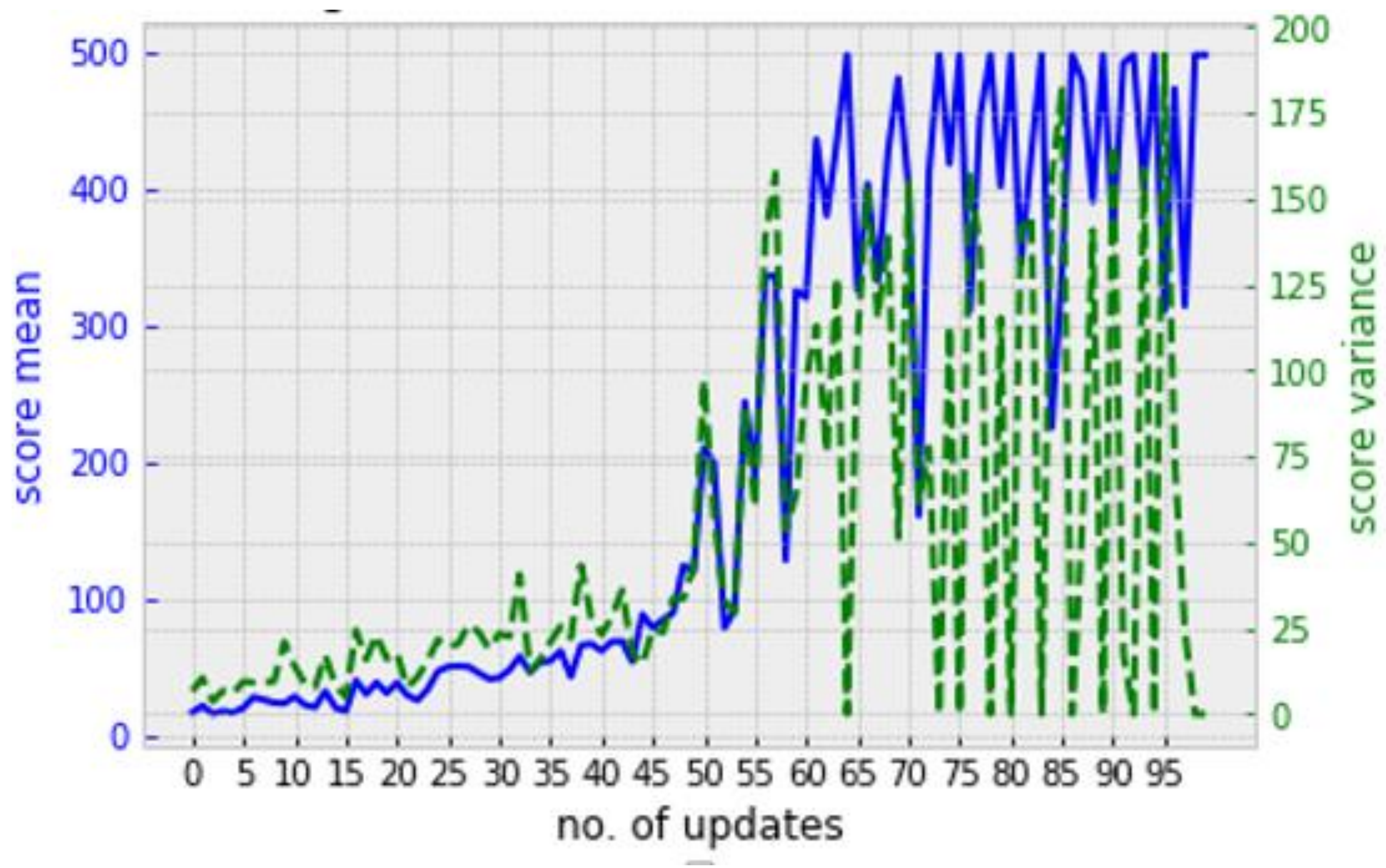
```
def replay(self, batch_size):
    X_batch = np.array([0,0,0,0])
    y_batch = np.array([0,0])
    minibatch = random.sample(self.memory, batch_size)      experience replay
    for state, action, reward, next_state, done in minibatch:
        target = reward
        if not done:
            target = (reward + self.gamma *
                      np.amax(self.model.predict_local(next_state)[0]))
        target_f = self.model.predict_local(state)
        target_f[0][action] = target
        X_batch = np.vstack((X_batch, state))
        y_batch = np.vstack((y_batch, target_f))

    rdd_sample = to_RDD(X_batch,y_batch)
    self.model.fit(rdd_sample, None, nb_epoch=10, batch_size=batch_size)
```

Analytics Zoo Keras-style Model

```
def _build_model(self):
    # Neural Net for Deep-Q learning Model
    model = Sequential()
    model.add(Dense(24, input_dim=self.state_size, activation='relu'))
    model.add(Dense(24, activation='relu'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mse',
                  optimizer=Adam(learningrate=self.learning_rate))
    return model
```

Vanilla DQN



score mean
score variance
no. of updates
0
100
200
300
400
500
0
25
50
75
100
125
150
175
200
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95

Policy gradients

- In Policy Gradients, we usually use a neural network (or other function approximators) to directly model the action probabilities.

- we tweak the parameters θ of the neural network so that "good" actions will be sampled more likely in the future.

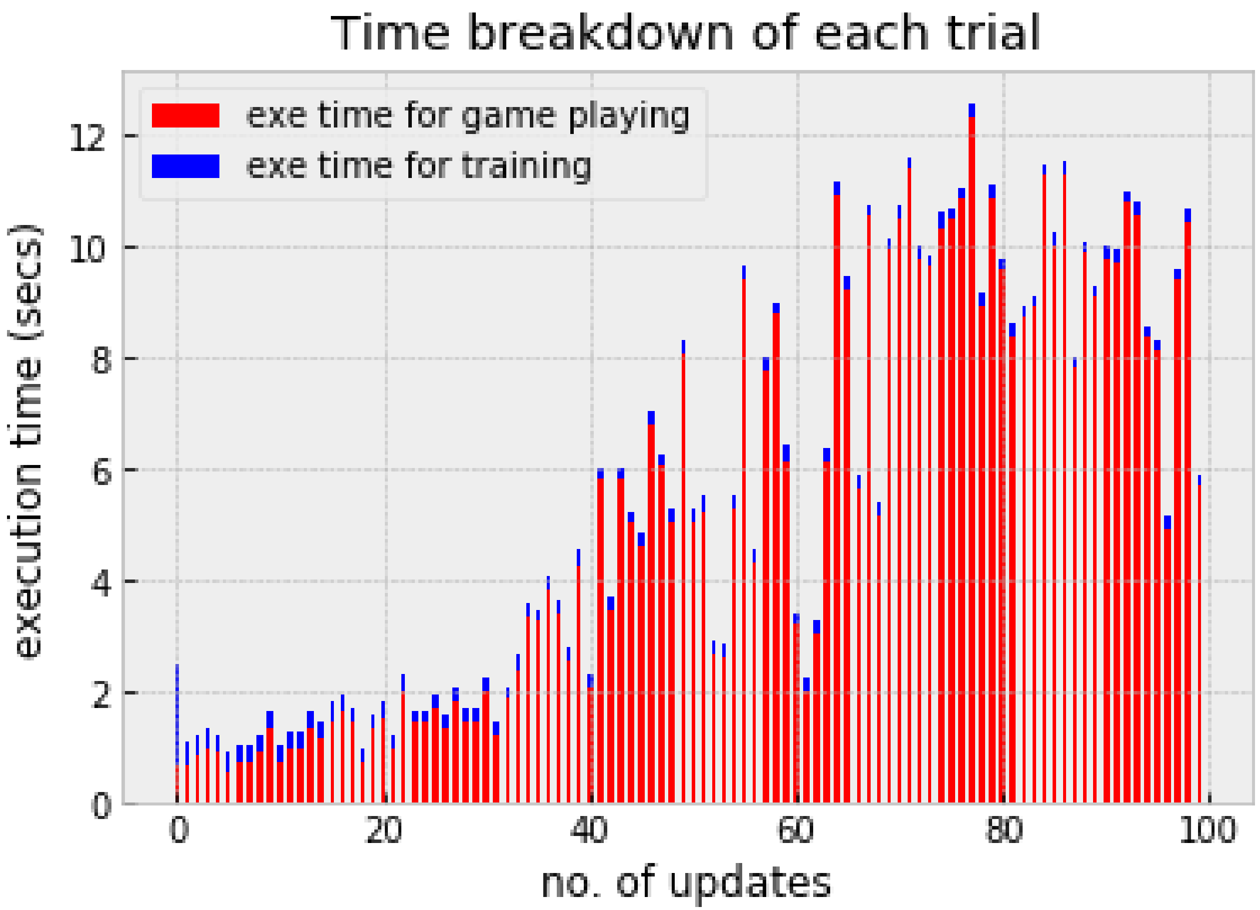
$$\nabla_\theta E[R_t] = E[\nabla_\theta log P(a) R_t]$$

REINFORCE

function REINFORCE
    Initialise θ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
        for $t = 1$ to $T - 1$ **do**
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return θ
end function

Time breakdown

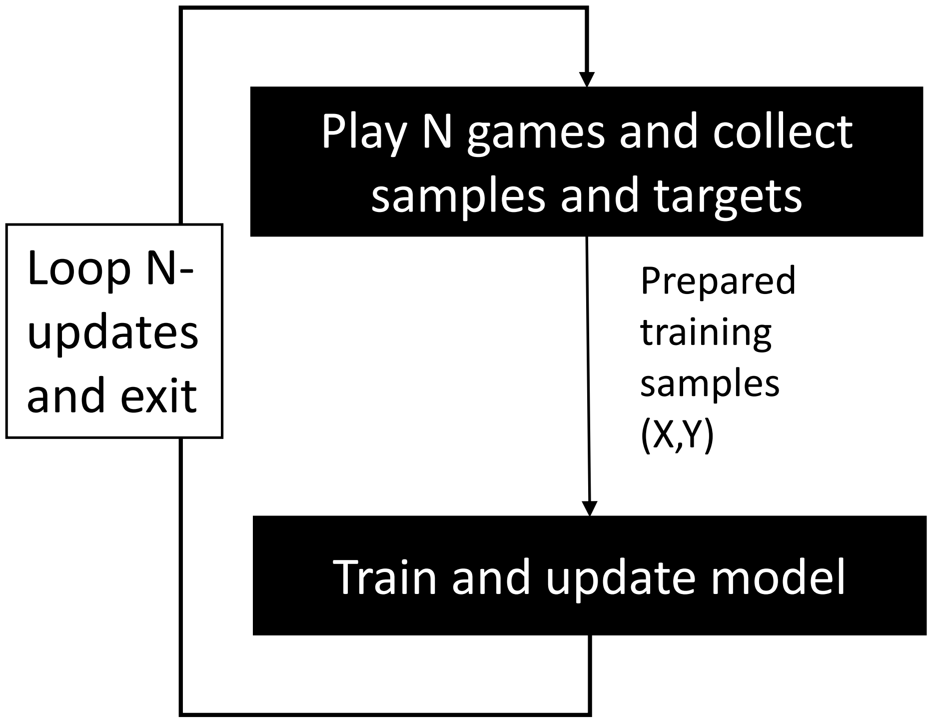
- Game playing takes the most time in each iteration



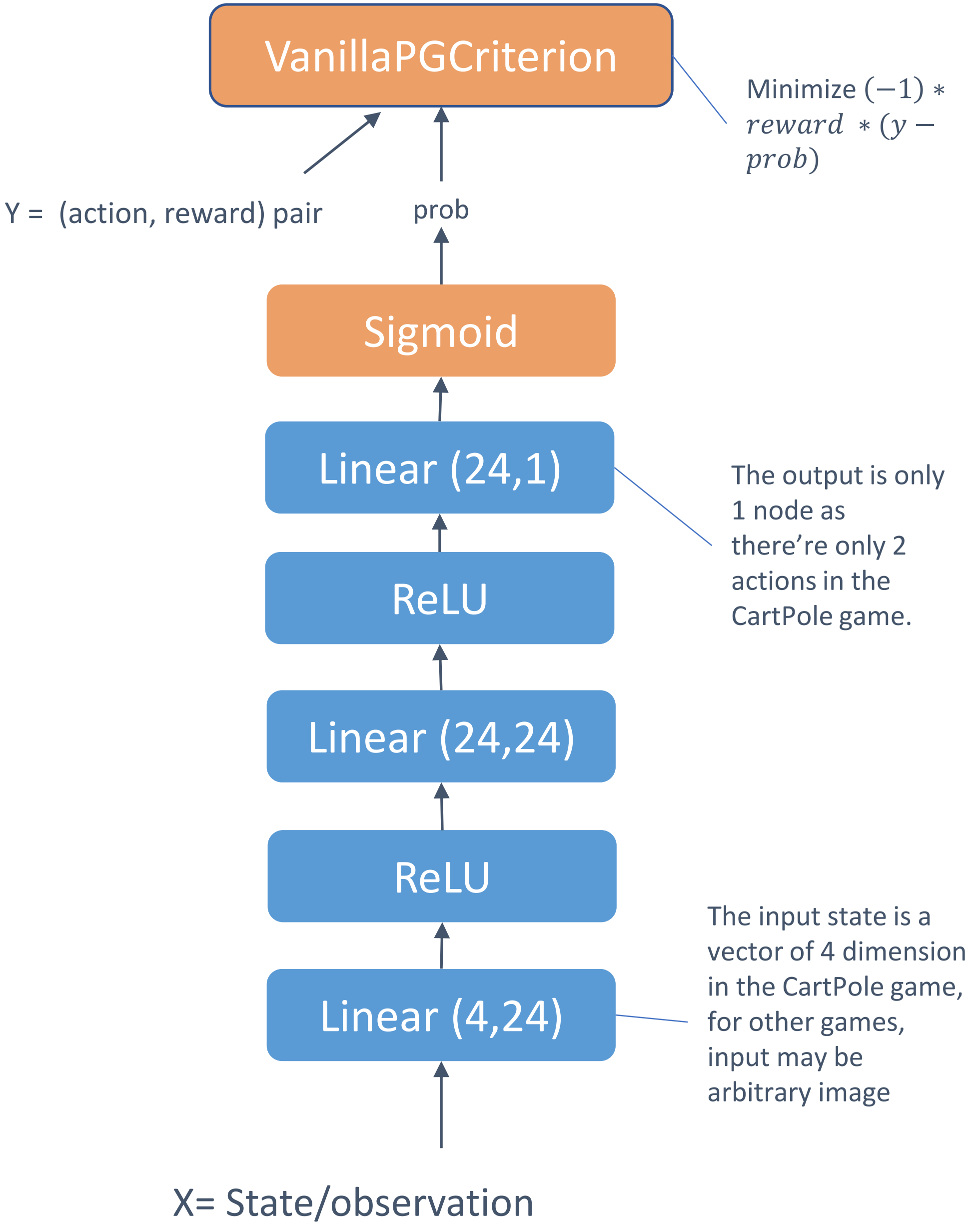
Distributed REINFORCE

```
# create and cache several agents on each partition as specified by parallelism
# and cache it
with DistributedAgents(sc, create_agent=create_agent, parallelism=parallelism) as a:
    agents = a.agents # a.agents is a RDD[Agent]
    optimizer = None
    num_trajs_per_part = int(math.ceil(15.0 / parallelism))
    mean_std = []
    for i in range(60):
        with SampledTrajs(sc, agents, model, num_trajs_per_part=num_trajs_per_part) as trajs:
            trajs = trajs.samples \ # samples is a RDD[Trajectory]
                .map(lambda traj: (traj.data["observations"],
                                   traj.data["actions"],
                                   traj.data["rewards"]))
```

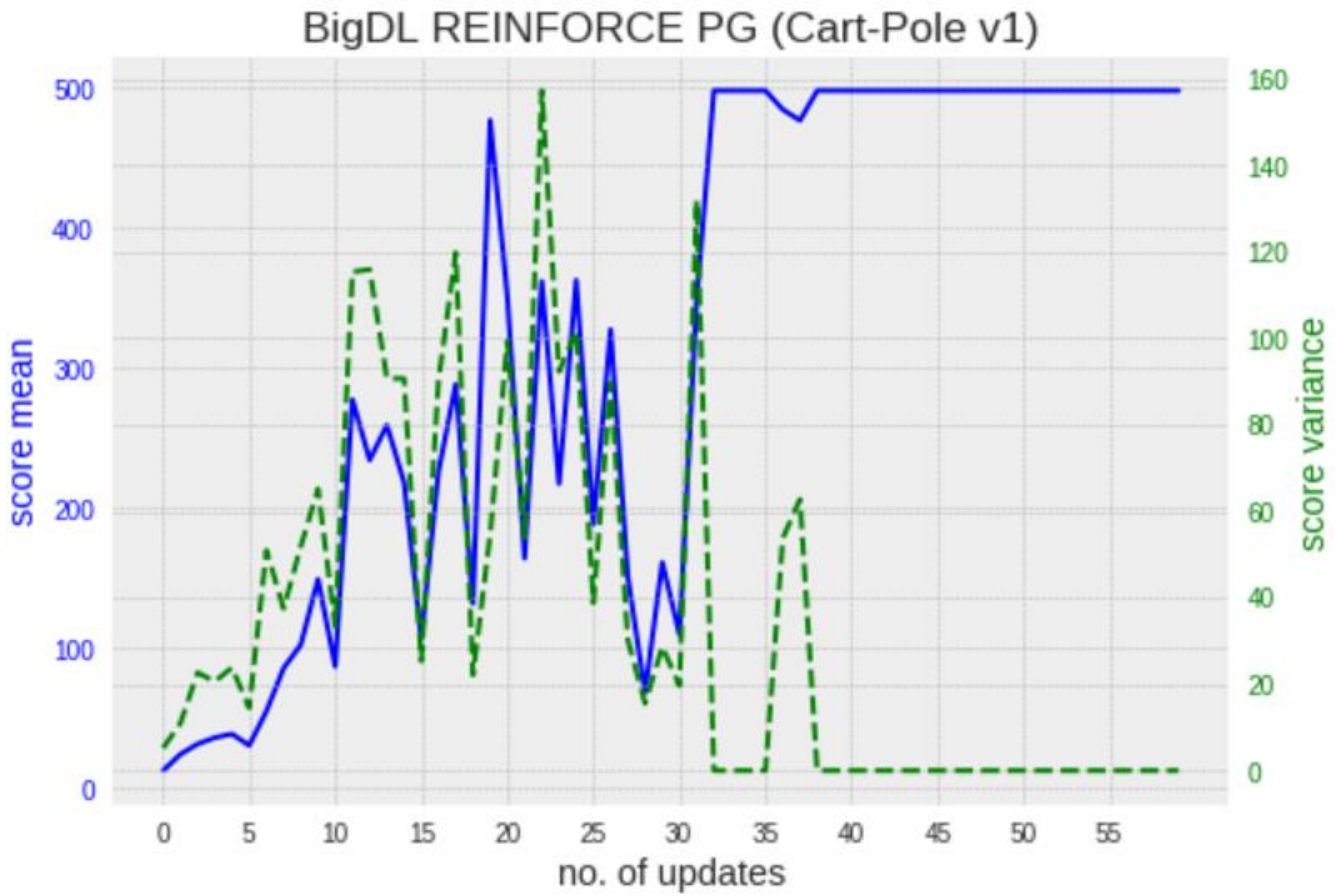
REINFORCE algorithm



Overflow of a PG program

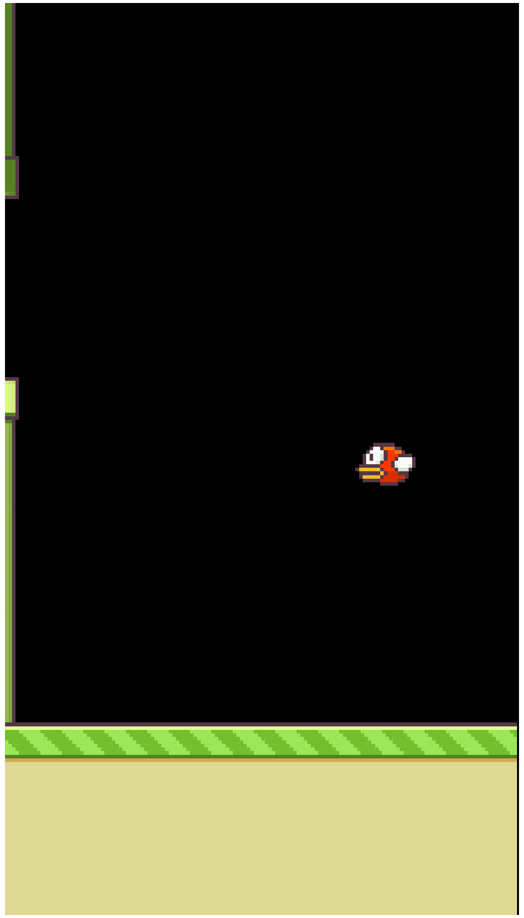


Distributed REINFORCE



Other RL algorithms

- Flappy bird with DQN
- Discrete and continuous PPO
- A2C (in roadmap)



Q & A

Analytics Zoo | High level API, Industry pipelines, App demo & Util

https://github.com/intel-analytics/analytics-zoo

Thanks Shane Huang and Yang Wang for working on RL implementations.