Application of deep neural networks for decision making in first-person shooter

student: Andrei Kashin[1]
science advisor: Ilya Makarov[1]

NRU HSE

Table of content

Markov decision process

For discrete time $t \in \mathbb{N}$, set of states $s_t \in \mathcal{S}$, actions $a_t \in \mathcal{A}$ and rewards $r_t \in \mathbb{R}$ we have a random process $\{s_t, a_t, r_t\}_{t=0}^{\infty}$ defined as:

$$\begin{aligned} s_0 &\sim \mathcal{P}_0 \\ a_t &\sim \pi(s_t) \\ r_t &\sim \mathcal{R}(s_t, a_t) \\ s_{t+1} &\sim \mathcal{P}(s_t, a_t) \end{aligned}$$

The optimal control problem:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \; \mathbb{E}_{a_t \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

where $\gamma \in [0, 1)$ is a discount factor.

Value functions

State value function:

$$V^\pi(s) = \mathbb{E}_{s_0=s, a_t \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

State-action value function:

$$Q^\pi(s, a) = \mathbb{E}_{s_0=s, a_0=a, a_t \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

Optimal value functions:

$$V^*(s) = \max_\pi V^\pi(s)$$
$$Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

Bellman equations

- Bellman expectation equation:

$$Q^{\pi}(s,a) = \mathbb{E}_{s',a'}\left[r + \gamma Q^{\pi}(s',a') \mid s,a\right]$$

- Bellman optimality equation:

$$Q^{*}(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^{*}(s',a') \mid s,a\right]$$

- **Policy iteration** solves Bellman expectation equation:

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

- **Value iteration** solves Bellman optimality equation:

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

Dynamic programming

Use value iteration to estimate optimal value function and take actions greedily:

$$
\begin{aligned}
Q^\pi(s,a) &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)(\mathcal{R}(s,a) + \gamma \max_{a'} Q^\pi(s',a')) \\
\pi(s) &= \operatorname*{argmax}_a Q^\pi(s,a)
\end{aligned}
$$

Needs to exactly know the environment dynamics $\mathcal{P}$ and $\mathcal{R}$.

Reinforcement learning

Q-learning

Use value iteration to estimate optimal value function from Monte-Carlo sampling using the following update rule:

$$Q^*(s_t, a_t) = Q^*(s_t, a_t) + \alpha(r_t + \gamma(\max_a Q^*(s_{t+1}, a) - Q^*(s_t, a_t)))$$

Only needs to sample from $\mathcal{P}$ and $\mathcal{R}$.

Non-linear Q-learning

- Represent value function as neural network with weights θ:

$$Q^{\pi}(s, a) \approx Q(s, a, \theta)$$

- Define objective as a mean-squared error in Q-values:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta)\right)^2\right]$$

Very unstable, oscillates and diverges.

1. Data is sequential - successive samples are correlated, non-iid
2. Policy changes rapidly with slight changes to Q-values
3. Scale of rewards is unknown and can lead to large gradients

DQN (Deep Q-network)

Collection of techniques to improve the stability of Q-learning training procedure:

1. Experience replay — use random subset of collected data to train. Breaks correlation between training data.
2. Target network — replace loss function with:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', \theta^-) - Q(s, a, \theta)\right)^2\right]$$

   where θ^- are periodically synchronized with θ. Makes oscillations and divergence unlikely.
3. Clip or normalize rewards to $[-1, 1]$ range. Makes gradients more stable.

DQN

Architecture

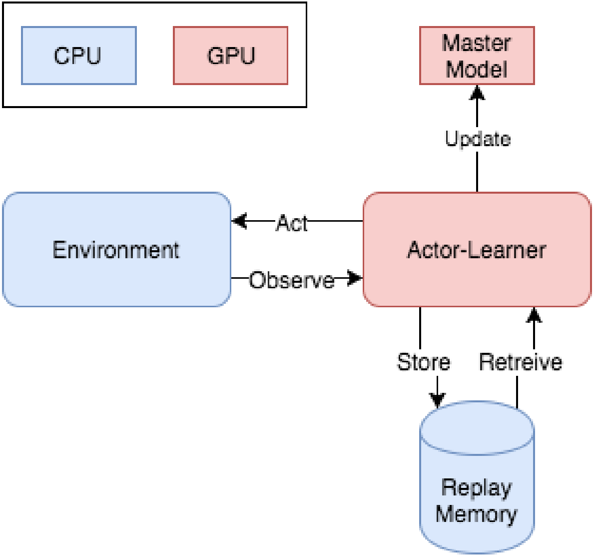


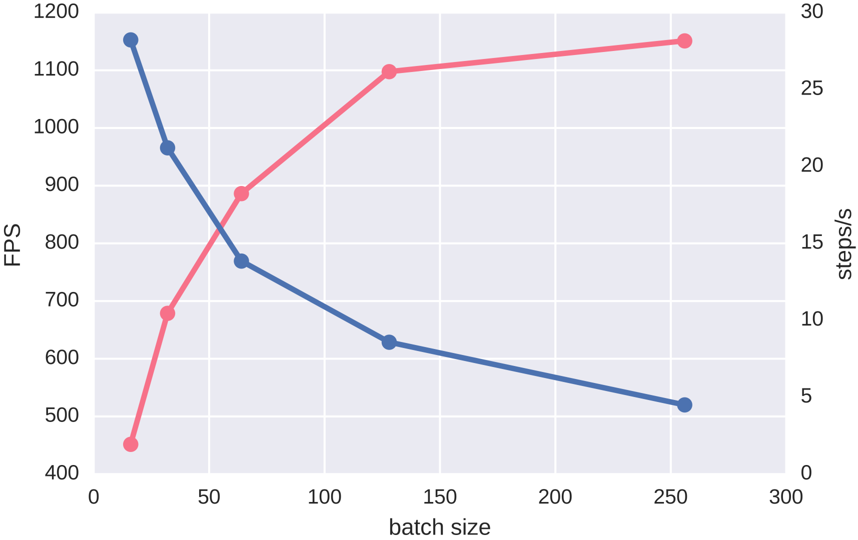
Table of content

Hardware performance

- GPU is much more suitable for training (10x boost) due to better parallelism
- CPU is not much worse for prediction
- On modern systems there are more independent CPUs than GPUs (8:1)

CPU Performance

Training



GPU Performance

Training

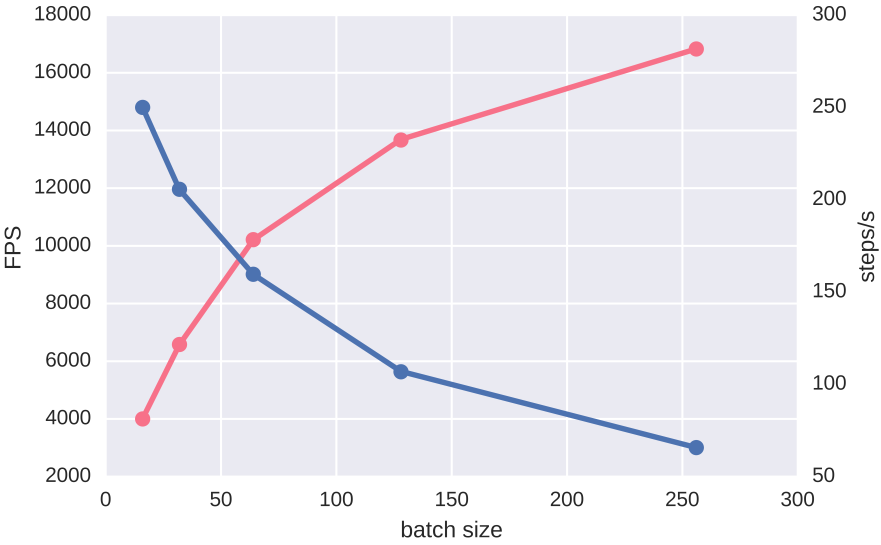
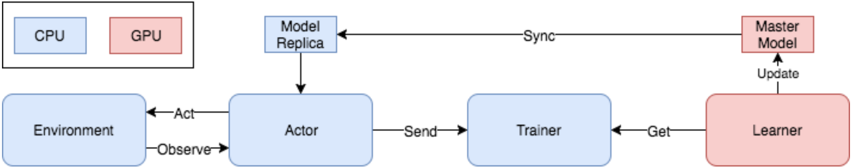


Table of content

Asynchronous Actor-Trainer-Learner

Arhitecture for hybrid CPU/GPU reinforcement learning training developed in this work.



- Can utilize CPU and GPU and scales with more compute power easily
- No blocking between actor and learner leads to higher throughput
- Flexibility in choosing a training procedure
- Support for multiple independent actors

Asynchronous Actor-Trainer-Learner

Components

- Actor — interacts with the environment by acting and collecting the experience. Periodically synchronizes with master replica. Runs on CPU.
- Trainer — collects the experience produced by actors and decides which samples will be used by learner to train the model.
- Learner — consumes samples provided by trainer to compute gradients and update the master model. Runs on GPU, as it needs to perform heavy batch computations.

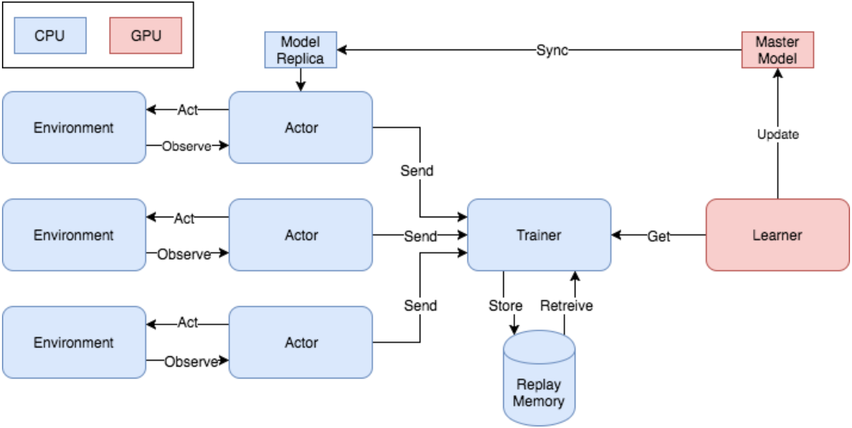
Asynchronous DQN

Implementation of DQN in Async Actor-Trainer-Learner architecture

- Uses DQN training procedure
- Uses experience replay to store samples
- Uses target network
- Uses CPU for actors and GPU for learner
- Actor and learner use separate model parameters

Asynchronous DQN

Architecture



Asynchronous DQN

Throughput

For batch size 64 and training frequency 4 for DQN

Method	Actor, steps/s	Learner, steps/s
DQN	227	57
AsyncDQN	300	167

Async DQN yields 30% faster actor and 300% faster learner.

Table of content

VizDoom


0
AMMO
22%
HEALTH
2 3 4
5 6 7
ARMS
0%
ARMOR
BULL 0 / 200
SHEL 0 / 50
RCKT 0 / 50
CELL 0 / 300

VizDoom

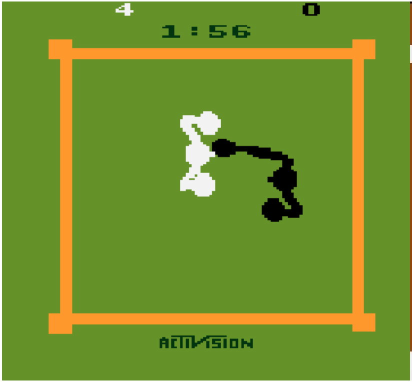
- Input: 160x120 RGB image
- Output: 3-43 actions depending on environment
- Episode lasts up to 2100 steps / 60 seconds of realtime
- RGB to Greyscale
- Frameskip of 4 used

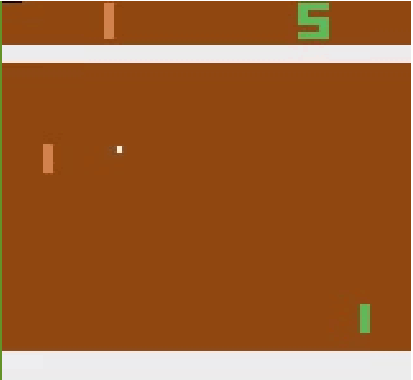
VizDoom

Hyperparameters search

- learning rate: $[1e-4, 5e-4, 1e-3]$
- batch size: $[16, 32, 64, 128, 256]$
- target update frequency: $[50, 100, 200, 400]$

Atari

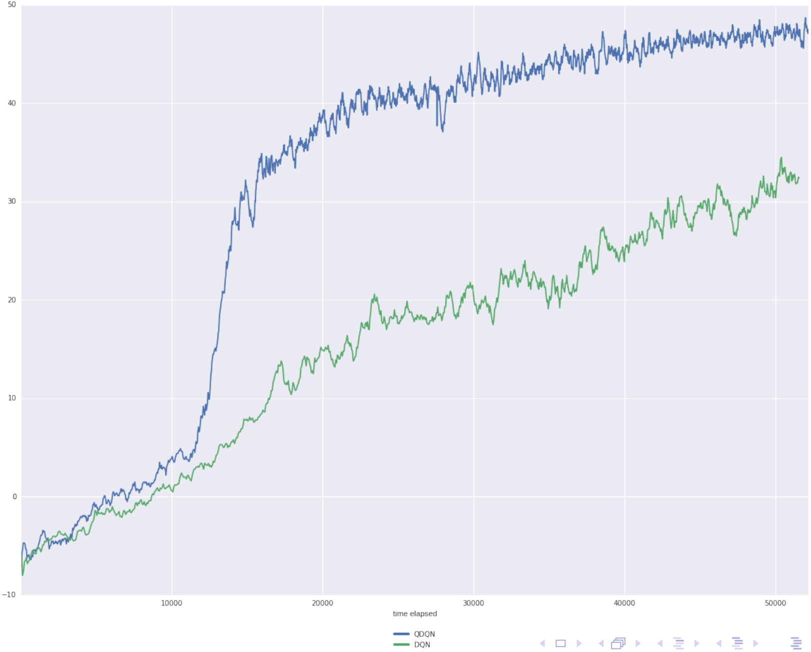




Atari

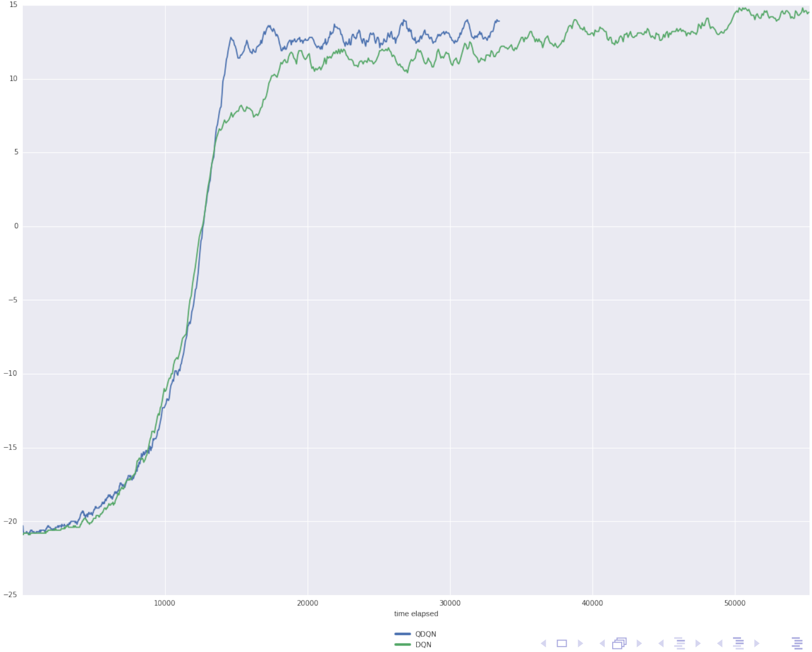
- Input: 210x160 RGB image
- Output: 18 actions
- Episode lasts up to 3600 frames / 2 minutes of realtime
- RGB to Greyscale
- Frameskip of 4 used

Boxing



50
40
30
20
10
0
−10
10000
20000
30000
40000
50000
time elapsed
QDQN
DQN

Pong



15
10
5
0
−5
−10
−15
−20
−25
10000
20000
30000
40000
50000
time elapsed
QDQN
DQN

Table of content

Future Work

- Improve sample efficiency
- Evaluate multiple parallel agents
- Extend to policy gradient methods