Polytechnic University of Timișoara
2023-2024

REINFORCEMENT LEARNING LABORATORY
Atari Freeway-v0

Eng. Isac-Daniel ClOBOTĂ

June 2024

Table of Contents

1. Game Description

Atari Freeway is a classic game developed by David Crane and published Activision in 1981. The game was built for the Atari 2600 video game console. The goal is to help the chicken cross the road (a multi-lane highway) without being crushed by a car. The chicken starts from the bottom of the screen and has to cross ten lanes of traffic (five in each direction). The action space is discrete, having three possible actions: UP, DOWN and NOOP (not moving) [1]. If you choose difficulty to be easy, every time you collide with a car, you are knocked back one lane. Otherwise, if you choose difficulty to be hard, every time you collide with a car, you are knocked back to the starting position. Every time you manage to cross all the lanes, you are awarded 1 point. The score is visible at the top of the screen. The game also has a multiplayer mode. Two players can compete against each other. The one who has the most points at the end of the game will be the winner. The total duration of the game is 2 minutes. When you approach the final seconds, the score will start blinking [2]. Screenshots from the game can be observed in Figure 1.

You can select one of the eight available maps (Lake Shore Drive – Chicago – 3 A.M., Interstate 5 – Seattle – 6 A.M., Santa Monica Freeway - Los Angeles - 10 A.M., Bayshore Freeway - San Francisco – Midnight, John Lodge Expressway – Detroit - 9 P.M., The Beltway - Washington D.C. - 6 P.M., LBJ Freeway - Dallas - 5 P.M., Long Island Expressway - New York City - 3 A.M.) [2]. Each map has different traffic intensities, car speed and vehicle types (automobiles and trucks). Maps 5-8 are identical to the first maps, but the speeds of the vehicles are randomly increased or decreased.

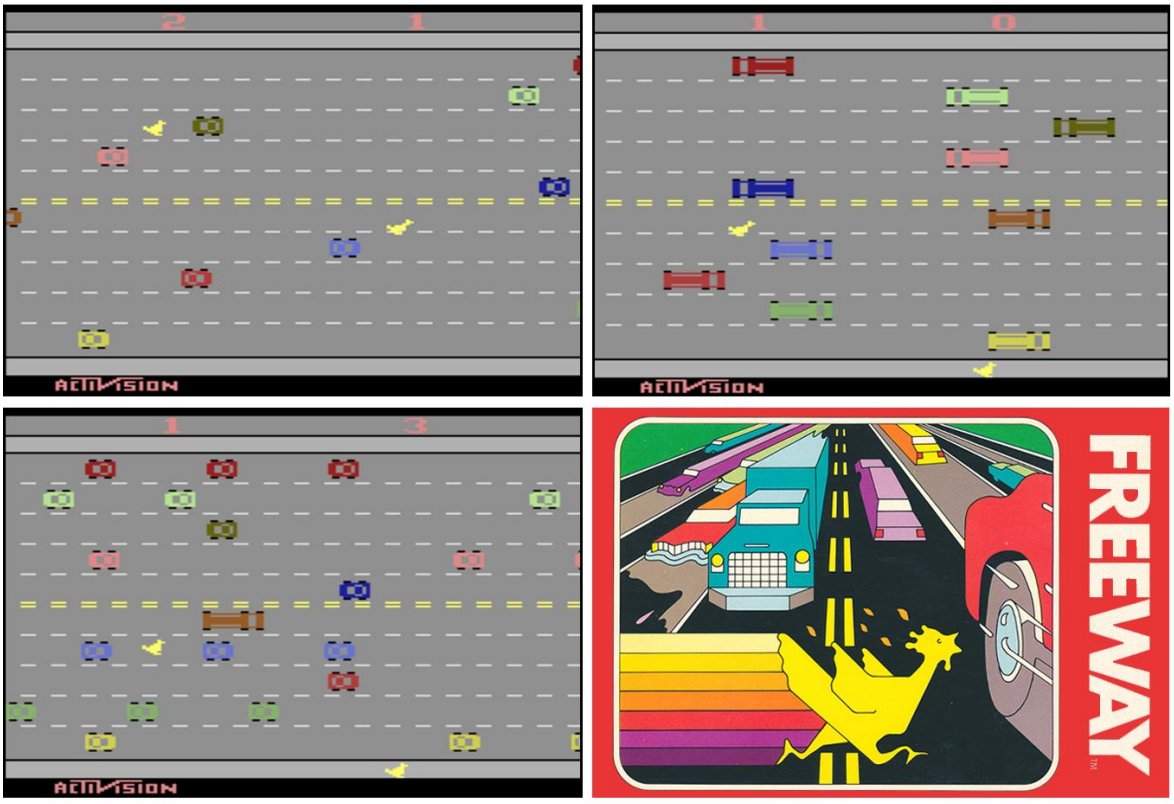


Figure 1. Screenshots from the game (top row and bottom-left). Illustration from the original box of the game (bottom-right). [3]

2. Algorithms

2.1 REINFORCE

The algorithm was developed by Ronald J. Williams in 1992. It is a basic algorithm. Actions that produced good outcomes will become more probable during learning. On the contrary, actions that produced bad outcomes will become less probable. So, we can say that it „reinforces" the correct actions. The agent will directly learn policies (mapping states to actions). The algorithm can be observed in Figure 2.

Initialize learning rate α
Initialize weights θ of a policy network π_θ
for *episode* $= 0, \ldots,$ *MAX_EPISODE* **do**
  Sample a trajectory $\tau = (s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)$
  Set $\nabla_\theta J(\pi_\theta) = 0$
  for $t = 0, \ldots, T$ **do**
    $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
    $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$
  end for
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$
end for

Figure 2. REINFORCE Algorithm [4]

2.2 SARSA

It was invented by Rummery and Niranjan in 1994. It is a value-based algorithm used for learning the Q-function and generating actions. The name stands for State Action Reward State Action highlighting the key components involved in updating the Q-values. The algorithm can be observed in Figure 3.

Initialize learning rate α
Initialize ε
Randomly initialize the network parameters, θ
for $m = 1, \ldots,$ *MAX_STEPS* **do**
  Gather N experiences $(s_i, a_i, r_i, s_i', a_i')$ using the current ε-greedy policy
  for $i = 1, \ldots, N$ **do**
    # Calculate target Q-values for each example
    $y_i = r_i + \delta_{s_i'} \gamma Q^{\pi_\theta}(s_i', a_i')$ where $\delta_{s_i'} = 0$ if s_i' is terminal, 1 otherwise
  end for
  # Calculate the loss, for example using MSE
  $L(\theta) = \frac{1}{N} \sum_i (y_i - Q^{\pi_\theta}(s_i, a_i))^2$
  # Update the network's parameters
  $\theta = \theta - \alpha \nabla_\theta L(\theta)$
  Decay ε
end for

Figure 3. SARSA Algorithm with ε-greedy policy. [4]

2.3 Deep Q-Networks (DQN)

Introduced in 2013 by Mnih et. al. It is similar to SARSA because it is a value-based temporal difference algorithm that learns the Q-function. The difference is that SARSA will approximate the Q-function for the current policy, while DQN will approximate the optimal Q-function, improving the stability and speed of learning. It does this by using not only the action taken in the next state, but all the possible actions from that state, making it an off-policy algorithm. There are several improvements over the "vanilla" version of the algorithm, which will be discussed in the next chapters. The algorithm can be observed in Figure 4.

Initialize learning rate α
Initialize τ
Initialize number of batches per training step, B
Initialize number of updates per batch, U
Initialize batch size, N
Initialize experience replay memory with max size, K
Randomly initialize the network parameters, θ
for $m = 1, \ldots,$ *MAX_STEPS* **do**
    Gather and store m experiences $(s_i, a_i, r_i, s_i', a_i')$ using the current policy
    for $b = 1, \ldots, B$ **do**
        Sample a batch, b, of experiences from the experience replay memory
        for $u = 1, \ldots, U$ **do**
            for $i = 1, \ldots, N$ **do**
                # Calculate target Q-values for each example
                $y_i = r_i + \delta_{s_i'} \gamma \max_{a_i'} Q^{\pi\theta}(s_i', a_i')$ where $\delta_{s_i'} = 0$ if s_i' is terminal, 1 otherwise
            end for
            # Calculate the loss, for example using MSE
            $L(\theta) = \frac{1}{N} \sum_i (y_i - Q^{\pi\theta}(s_i, a_i))^2$
            # Update the network's parameters
            $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$
        end for
    end for
    Decay τ
end for

Figure 4. DQN Algorithm with Boltzmann policy. [4]

2.4 DQN with target networks

It represents the first improvement of the DQN algorithm. In vanilla DQN, the same Q-network is used for both selecting and evaluating actions, which can lead to instability. To overcome this problem, a separate target network is used. The target network is a copy of the Q-network but is updated less frequently (lagged copy). The target Q-values are then calculated using this target network, which provides more stable targets for the training process. The algorithm is presented in Figure 5.

Initialize learning rate α
Initialize τ
Initialize number of batches per training step, B
Initialize number of updates per batch, U
Initialize batch size, N
Initialize experience replay memory with max size, K
Initialize target network update frequency, F
Randomly initialize the network parameters, θ
Initialize the target network parameters $\phi = \theta$
for $m = 1, \ldots,$ *MAX_STEPS* **do**
  Gather and store m experiences $(s_i, a_i, r_i, s_i', a_i')$ using the current policy
  for $b = 1, \ldots, B$ **do**
    Sample a batch, b of experiences from the experience replay memory
    for $u = 1, \ldots, U$ **do**
      for $i = 1, \ldots, N$ **do**
        # Calculate target Q-values for each example
        $y_i = r_i + \delta_{s_i'} \gamma \max_{a_i'} Q^{\pi_\phi}(s_i', a_i')$ where $\delta_{s_i'} = 0$ if s_i' is terminal, 1 otherwise
      end for
      # Calculate the loss, for example using MSE
      $L(\theta) = \frac{1}{N} \sum_i (y_i - Q^{\pi_\theta}(s_i, a_i))^2$
      # Update the network's parameters
      $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$
    end for
  end for
  Decay τ
  if $(m \bmod F) == 0$ **then**
    # Update the target network
    $\phi = \theta$
  end if
end for

Figure 5. DQN with target networks. [4]

2.5 Double DQN (DDQN)

DDQN represents the second improvement of the DQN algorithm. It uses double estimation for computing the Q-function by using a second Q-function trained using different experiences. The algorithm is presented in Figure 6.

$$Q^{\pi}_{tar:DQN}(s,a) = r + \gamma \max_{a'} Q^{\pi_\theta}(s',a')$$

$$Q^{\pi}_{tar:DQN}(s,a) = r + \gamma \max_{a'} Q^{\pi_\theta}(s',a')$$

Figure 6. DDQN vs DQN Q-function computation. [4]

2.6 Prioritized Experience Replay (DQN PER)

It represents the third and last improvement of the DQN algorithm. It was introduced in 2015 by Schaul et. al. It improves upon the standard experience replay mechanism used in DQN. In vanilla DQN, experiences are sampled uniformly from the replay memory. In contrast, PER prioritizes more informative experiences by assigning a priority to each experience. The only difference between DDQN and DQN is the way that the Q-function is computed. The differences can be observed in Figure 7.

Initialize learning rate α
Initialize τ
Initialize number of batches per training step, B
Initialize number of updates per batch, U
Initialize batch size, N
Initialize experience replay memory with max size, K
Initialize target network update frequency, F
Initialize the maximum priority, P
Initialize ε
Initialize the prioritization paramater η
Randomly initialize the network parameters, θ
Initialize the target network parameters $\phi = \theta$
for $m = 1, \ldots,$ *MAX_STEPS* **do**
    Gather and store m experiences $(s_i, a_i, r_i, s_i', p_i)$ where $p_i = P$
    for $b = 1, \ldots, B$ **do**
        Sample a **prioritized** batch, b of experiences from the experience replay memory
        for $u = 1, \ldots, U$ **do**
            for $i = 1, \ldots, N$ **do**
                # Calculate target Q-values for each example
                $y_i = r_i + \delta_{s_i'} \gamma Q^{\pi\phi}(s_i', \arg\max_{a_i'} Q^{\pi\theta}(s_i', a_i'))$ where $\delta_{s_i'} = 0$ if s_i' is terminal, 1 otherwise
                # Calculate the absolute TD error for each example
                $\omega_i = |y_i - Q^{\pi\theta}(s_i, a_i)|$
            end for
            # Calculate the loss, for example using MSE
            $L(\theta) = \frac{1}{N} \sum_i (y_i - Q^{\pi\theta}(s_i, a_i))^2$
            # Update the network's parameters
            $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$
            # Calculate the priorities for each example
            $p_i = \frac{(|\omega_i| + \varepsilon)^\eta}{\sum_j (|\omega_j| + \varepsilon)^\eta}$
            # Update the experience replay memory with the new priorities
        end for
    end for
    Decay τ
    if $(m \bmod F) == 0$ **then**
        # Update the target network
        $\phi = \theta$
    end if
end for

Figure 7. DQN PER Algorithm. [4]

2.7 Advantage Actor-Critic (A2C)

A2C stands for Advantage Actor-Critic. This type of algorithm involves two main components. The first component is an **actor** that learns a policy and tries to output the best action to take in each state. The second component is a **critic** that will learn a value function and will try to provide a baseline for the actor to compare against. The advantage function is used to decide how much better taking a particular action in state is compared to the average action taken in that state. The general A2C algorithm can be observed in Figure 8.

There are two different methods for estimating the advantage function: *n*-step returns and Generalized Advantage Estimation (GAE). A2C with *n*-step returns will use the returns from *n* future steps to update the value estimates and the policy. Instead of waiting until the end of an episode to calculate the total reward A2C updates its estimates after a fixed number of steps. Generalized Advantage Estimation (GAE) represents the second method for estimating the advantage function of an A2C algorithm. It doesn't require choosing a value for *n*. Instead, it will use multiple values for *n* and will average all the returns.

Set $\beta \geq 0$ # entropy regularization weight
Set $\alpha_A \geq 0$ # actor learning rate
Set $\alpha_C \geq 0$ # critic learning rate
Randomly initialize the actor and critic parameters θ_A, θ_C
for *episode* $= 0, \ldots,$ *MAX_EPISODE* **do**
  Gather and store data (s_t, a_t, r_t, s'_t) by acting in the environment using the current policy
  for $t = 0, \ldots, T$ **do**
    Calculate predicted V-value $\hat{V}^{\pi}(s_t)$ using the critic network θ_C
    Calculate the advantage $\hat{A}^{\pi}(s_t, a_t)$ using the critic network θ_C
    Calculate $V^{\pi}_{tar}(s_t)$ using the critic network θ_C and/or trajectory data
    Optionally, calculate entropy H_t of the policy distribution, using the actor network θ_A.
    Otherwise set $\beta = 0$
  end for
  Calculate value loss, for example using MSE:
  $L_{val}(\theta_C) = \frac{1}{T} \sum_{t=0}^{T} (\hat{V}^{\pi}(s_t) - V^{\pi}_{tar}(s_t))^2$
  Calculate policy loss:
  $L_{pol}(\theta_A) = \frac{1}{T} \sum_{t=0}^{T} (-\hat{A}^{\pi}(s_t, a_t) \log \pi_{\theta_A}(a_t|s_t) - \beta H_t)$
  Update critic parameters, for example using SGD:
  $\theta_C \leftarrow \theta_C - \alpha_C \nabla_{\theta_C} L_{val}(\theta_C)$
  Update actor parameters, for example using SGD:
  $\theta_A \leftarrow \theta_A - \alpha_A \nabla_{\theta_A} L_{pol}(\theta_A)$
end for

Figure 8. A2C Algorithm. [4]

2.8 Proximal Policy Optimization (PPO)

PPO was introduced by Schulman et al. in 2017. It improves policy gradient methods by introducing a surrogate objective function, which ensures stable learning by using monotonic policy updates. It also uses off-policy data to be reused (on-policy algorithms cannot reuse data). PPO's balance between simplicity and performance has made it a popular choice for a wide range of reinforcement learning tasks. It has two variants:

1. PPO with adaptive KL penalty: uses an objective function with a dynamically adjusted penalty based on KL divergence to control policy updates.
2. PPO with clipped surrogate objective: uses a clipped objective function to bound the updates directly. Outperformed the KL penalty version in the original paper.

Figure 9 shows the PPO algorithm with clipped surrogate objective.

Set $\beta \geq 0$ # entropy regularization weight
Set $\varepsilon \geq 0$ # the clipping variable
Set K the number of epochs
Set N the number of actors
Set T the time horizon
Set $M \leq NT$ the minibatch size
Set $\alpha_A \geq 0$ # actor learning rate
Set $\alpha_C \geq 0$ # critic learning rate
Randomly initialize the actor and critic parameters θ_A, θ_C
Initialize the "old" actor network θ_{Aold}
for $i = 1, 2, \ldots$ **do**
    Set $\theta_{Aold} = \theta_A$
    for *actor* $= 1, 2, \ldots, N$ **do**
        Run policy θ_{Aold} in environment for T time steps and collect the trajectories
        Compute advantages $A_1, \ldots, A_T$ using θ_{Aold}
        Calculate $V^{\pi}_{tar,1}, \ldots, V^{\pi}_{tar,T}$ using the critic network θ and/or trajectory data
    end for
    Let *batch* with size NT consist of the collected trajectories, advantages, and target V-values
    for *epoch* $= 1, 2, \ldots, K$ **do**
        for minibatch m in *batch* **do**
            The following are computed over the whole minibatch m:
            Calculate $r_m(\theta_A)$
            Calculate $J^{CLIP}_m(\theta_A)$ using the advantages A_m from the minibatch and $r_m(\theta_A)$
            Calculate entropies H_m using the actor network θ_A
            Calculate policy loss:
            $L_{pol} = -J^{CLIP}_m(\theta_A) - \beta H_m$
            Calculate predicted V-value $\hat{V}^{\pi}(s_m)$ using the critic network θ_C
            Calculate value loss using the V-targets from the minibatch:
            $L_{val}(\theta_C) = MSE(\hat{V}^{\pi}(s_m), V^{\pi}_{tar}(s_m))$
            Update actor parameters, for example using SGD:
            $\theta_A \leftarrow \theta_A - \alpha_A \nabla_{\theta_A} L_{pol}(\theta_A)$
            Update critic parameters, for example using SGD:
            $\theta_C \leftarrow \theta_C - \alpha_C \nabla_{\theta_C} L_{val}(\theta_C)$
        end for
    end for
end for

Figure 9. PPO with clipped surrogate objective. [4]

2.9 Asynchronous Advantage Actor-Critic (A3C)

It was introduced by Mnih et. al. and uses the A2C algorithm but speeds up the process by training agents in parallel. This leads to faster and more efficient learning. With asynchronous parallelization, the global network will update the parameters whenever it receives data from any worker and each worker will update itself from the global network. The A3C algorithm can be observed in Figure 10.

Set learning rate α
Initialize the global network, θ_G
Initialize N worker networks, $\theta_{W,1}, \theta_{W,2}, \ldots, \theta_{W,N}$
for each worker **asynchronously do**
    Pull parameters from the global network and set $\theta_{W,i} \leftarrow \theta_G$
    Collect trajectories
    Compute gradient $\nabla_{\theta_{W,i}}$
    Push $\nabla_{\theta_{W,i}}$ to the global network
end for
On receiving worker gradient, update the global network $\theta_G \leftarrow \theta_G - \alpha \nabla_{\theta_{W,i}}$

Figure 10. A3C Algorithm. [4]

3. Experimental Results

Every algorithm was trained for 500k steps, using the modular deep reinforcement learning framework in PyTorch called SLM Lab [5]. I will provide the full configuration file for each algorithm, and I will describe each file. Also, there will be graphs of the results, representing the mean returns and the moving average of the mean returns. However, some settings are the same for every file:

- **Network architecture**: Three layers of Convolutional Neural Network, followed by one fully connected layer, with ReLU activation function.
- **Environment**: OpenAI Gym's *Freeway-v0*
- **Training length**: 500,000 time steps

3.1 REINFORCE

The graphs can be observed in Figure 11 and the configuration file can be observed in Figure 12.

- **Algorithm**: The algorithm is Reinforce (line:6). Entropy with a linear decay schedule (line:13).
- **Optimizer**: Adam (line:43), with a learning rate of 0.002 (line:44). The learning rate is constant because the learning rate scheduler is set to null (line:46).
- **Training frequency**: Training is episodic, having *OnPolicyReplay* memory (line:22). Training frequency is set to 1 (line:19), meaning that the network will be trained every 1 episode.
- **Evaluation**: The agent is evaluated every 10,000 steps (line:65).

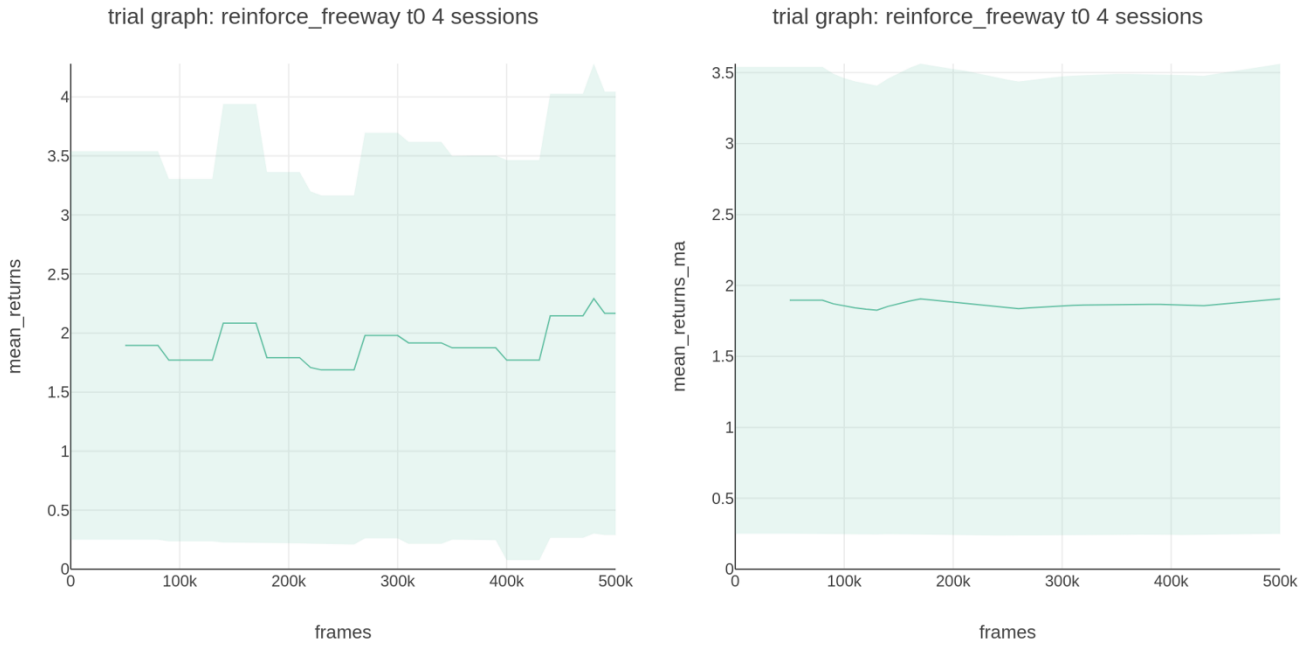


Figure 11. REINFORCE graphs.

```
{
  "reinforce_freeway": {
    "agent": [{
      "name": "Reinforce",
      "algorithm": {
        "name": "Reinforce",
        "action_pdtype": "default",
        "action_policy": "default",
        "center_return": true,
        "explore_var_spec": null,
        "gamma": 0.99,
        "entropy_coef_spec": {
          "name": "linear_decay",
          "start_val": 0.01,
          "end_val": 0.001,
          "start_step": 1000,
          "end_step": 100000
        },
        "training_frequency": 1
      },
      "memory": {
        "name": "OnPolicyReplay",
        "batch_size": 64
      },
      "net": {
        "type": "ConvNet",
        "shared": true,
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [32, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [512],
        "hid_layers_activation": "relu",
        "init_fn": "orthogonal_",
        "normalize": true,
        "batch_norm": false,
        "clip_grad_val": 0.5,
        "loss_spec": {
          "name": "MSELoss"
        },
        "optim_spec": {
          "name": "Adam",
          "lr": 0.002
        },
        "lr_scheduler_spec": null,
        "gpu": true
      }
    }],
    "env": [{
      "name": "Freeway-v0",
      "frame_op": "concat",
      "frame_op_len": 4,
      "reward_scale": "sign",
      "num_envs": 16,
      "max_t": null,
      "max_frame": 500000
    }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "eval_frequency": 10000,
      "log_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 12. REINFORCE config file.

3.2 SARSA

The graphs can be observed in Figure 13 and the configuration file can be observed in Figure 14.

- **Algorithm**: The algorithm is SARSA (line:6). Entropy with a linear decay schedule (line:13).
- **Optimizer**: Adam (line:41), with a learning rate of 0.001 (line:42). The learning rate is constant because the learning rate scheduler is set to null (line:44).
- **Training frequency**: Training is episodic, having *OnPolicyReplay* memory (line:20). Training frequency is set to 1 (line:19), meaning that the network will be trained every 1 episode.
- **Evaluation**: The agent is evaluated every 10,000 steps (line:63).

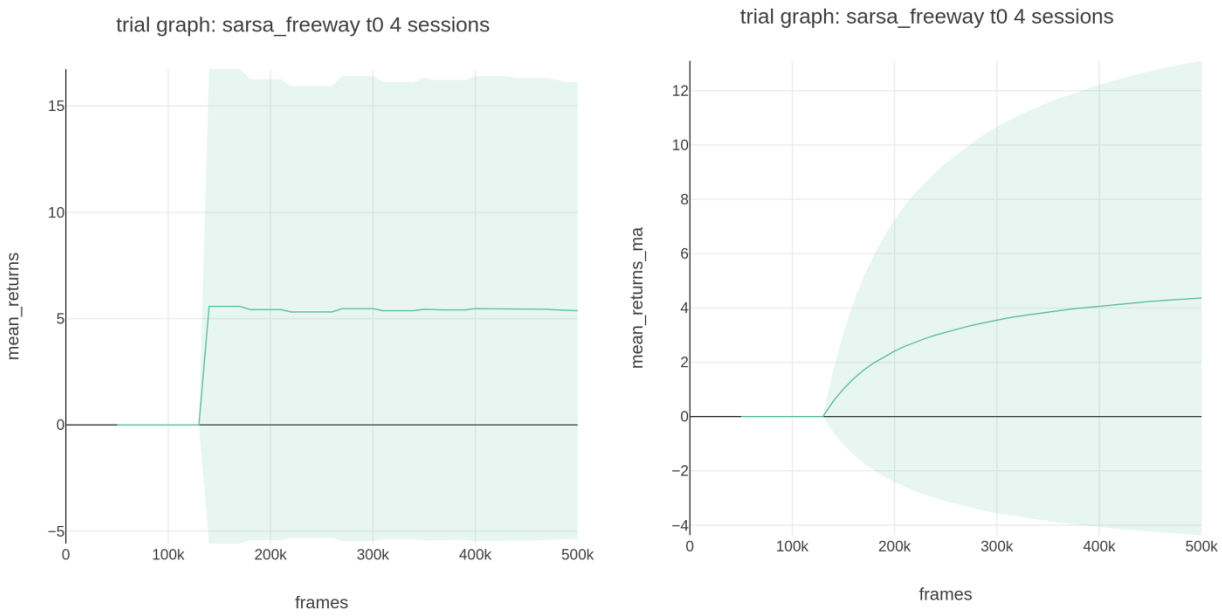


Figure 13. SARSA graphs.

```
{
  "sarsa_freeway": {
    "agent": [{
      "name": "SARSA",
      "algorithm": {
        "name": "SARSA",
        "action_pdtype": "Argmax",
        "action_policy": "epsilon_greedy",
        "entropy_coef_spec": {
          "name": "linear_decay",
          "start_val": 1.0,
          "end_val": 0.05,
          "start_step": 0,
          "end_step": 10000
        },
        "gamma": 0.99,
        "training_frequency": 32
      },
      "memory": {
        "name": "OnPolicyBatchReplay",
        "batch_size": 64
      },
      "net": {
        "type": "ConvNet",
        "shared": true,
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [32, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [512],
        "hid_layers_activation": "relu",
        "init_fn": "orthogonal_",
        "normalize": true,
        "batch_norm": false,
        "clip_grad_val": 0.5,
        "loss_spec": {
          "name": "MSELoss"
        },
        "optim_spec": {
          "name": "Adam",
          "lr": 0.001
        },
        "lr_scheduler_spec": null,
        "gpu": true
      }
    }],
    "env": [{
      "name": "Freeway-v0",
      "frame_op": "concat",
      "frame_op_len": 4,
      "reward_scale": "sign",
      "num_envs": 16,
      "max_t": null,
      "max_frame": 500000
    }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "eval_frequency": 10000,
      "log_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 14. SARSA config file.

3.3 DQN

The graphs can be observed in Figure 15 and the configuration file can be observed in Figure 16.

- **Algorithm**: The algorithm is VanillaDQN (line:7) (different from DQN, which uses target nets). The action policy is Boltzmann policy (line:9). Entropy with a linear decay schedule (line:11).
- **Optimizer**: Adam (line:47), with a learning rate of 0.001 (line:48). The learning rate is constant because the learning rate scheduler is set to null (line:50).
- **Training frequency**: Training starts after the agent has made 32 steps in the environment (line:21) and occurs every step from then on (line:20). *Replay* memory is used for sampling batches (line:24).
- **Evaluation**: The agent is evaluated every 10,000 steps (line:70).

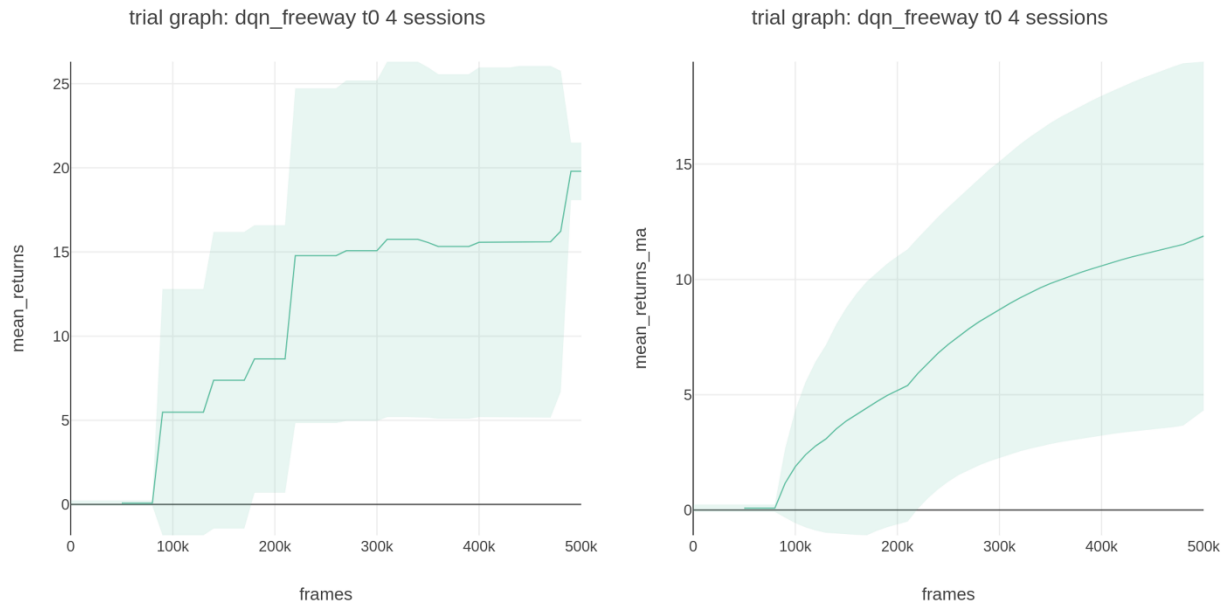


Figure 15. DQN graphs.

3.4 DQN with target networks

The graphs can be observed in Figure 17 and the configuration file can be observed in Figure 18.

- **Algorithm**: The algorithm is DQN (line:7). The action policy is Boltzmann policy (line:9). Entropy with a linear decay schedule (line:11).
- **Optimizer**: Adam (line:47), with a learning rate of 0.001 (line:48). The learning rate is constant because the learning rate scheduler is set to null (line:50).
- **Training frequency**: Training starts after the agent has made 32 steps in the environment (line:21) and occurs every step from then on (line:20). *Replay* memory is used for sampling batches (line:24).
- **Evaluation**: The agent is evaluated every 10,000 steps (line:70).

```
{
  "dqn_freeway": {
    "agent": [
      {
        "name": "VanillaDQN",
        "algorithm": {
          "name": "VanillaDQN",
          "action_pdtype": "Argmax",
          "action_policy": "boltzmann",
          "explore_var_spec": {
            "name": "linear_decay",
            "start_val": 5.0,
            "end_val": 0.5,
            "start_step": 0,
            "end_step": 1000
          },
          "gamma": 0.99,
          "training_batch_iter": 3,
          "training_iter": 3,
          "training_frequency": 1,
          "training_start_step": 32
        },
        "memory": {
          "name": "Replay",
          "batch_size": 32,
          "max_size": 10000,
          "use_cer": false
        },
        "net": {
          "type": "ConvNet",
          "shared": true,
          "conv_hid_layers": [
            [32, 8, 4, 0, 1],
            [64, 4, 2, 0, 1],
            [32, 3, 1, 0, 1]
          ],
          "fc_hid_layers": [512],
          "hid_layers_activation": "relu",
          "init_fn": "orthogonal_",
          "normalize": true,
          "batch_norm": false,
          "clip_grad_val": 0.5,
          "loss_spec": {
            "name": "SmoothL1Loss"
          },
          "optim_spec": {
            "name": "Adam",
            "lr": 0.001
          },
          "lr_scheduler_spec": null,
          "gpu": true
        }
      }
    ],
    "env": [{
        "name": "Freeway-v0",
        "frame_op": "concat",
        "frame_op_len": 4,
        "reward_scale": "sign",
        "num_envs": 16,
        "max_t": null,
        "max_frame": 500000
      }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "eval_frequency": 10000,
      "log_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 16. DQN config file.

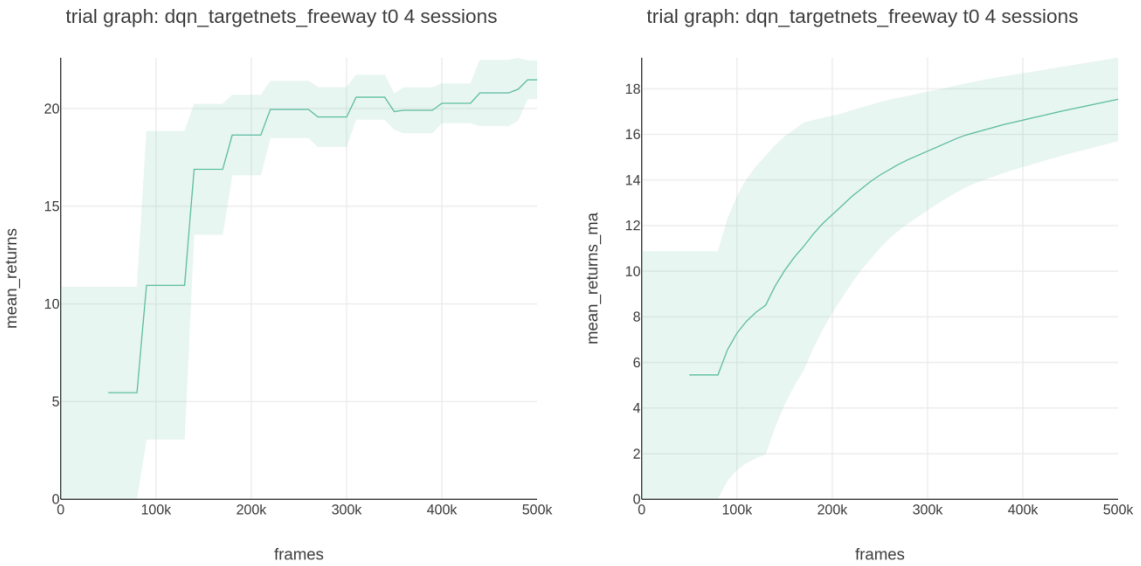


Figure 17. DQN with target nets graphs.

```
{
  "dqn_targetnets_freeway": {
    "agent": [
      {
        "name": "DQN",
        "algorithm": {
          "name": "DQN",
          "action_pdtype": "Argmax",
          "action_policy": "boltzmann",
          "explore_var_spec": {
            "name": "linear_decay",
            "start_val": 5.0,
            "end_val": 0.5,
            "start_step": 0,
            "end_step": 1000
          },
          "gamma": 0.99,
          "training_batch_iter": 3,
          "training_iter": 3,
          "training_frequency": 1,
          "training_start_step": 32
        },
        "memory": {
          "name": "Replay",
          "batch_size": 32,
          "max_size": 10000,
          "use_cer": false
        },
        "net": {
          "type": "ConvNet",
          "shared": true,
          "conv_hid_layers": [
            [32, 8, 4, 0, 1],
            [64, 4, 2, 0, 1],
            [32, 3, 1, 0, 1]
          ],
          "fc_hid_layers": [512],
          "hid_layers_activation": "relu",
          "init_fn": "orthogonal_",
          "normalize": true,
          "batch_norm": false,
          "clip_grad_val": 0.5,
          "loss_spec": {
            "name": "SmoothL1Loss"
          },
          "optim_spec": {
            "name": "Adam",
            "lr": 0.001
          },
          "lr_scheduler_spec": null,
          "gpu": true
        }
      }
    ],
    "env": [{
        "name": "Freeway-v0",
        "frame_op": "concat",
        "frame_op_len": 4,
        "reward_scale": "sign",
        "num_envs": 16,
        "max_t": null,
        "max_frame": 500000
      }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "eval_frequency": 10000,
      "log_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 18. DQN with target nets config file.

3.5 DDQN

The graphs can be observed in Figure 19 and the configuration file can be observed in Figure 20.

- **Algorithm**: The algorithm is Double DQN (line:7). The action policy is Boltzmann policy (line:9). Entropy with a linear decay schedule (line:11).
- **Optimizer**: Adam (line:47), with a learning rate of 0.001 (line:48). The learning rate is constant because the learning rate scheduler is set to null (line:50).
- **Training frequency**: Training starts after the agent has made 32 steps in the environment (line:21) and occurs every step from then on (line:20). *Replay* memory is used for sampling batches (line:24).
- **Evaluation**: The agent is evaluated every 10,000 steps (line:70).

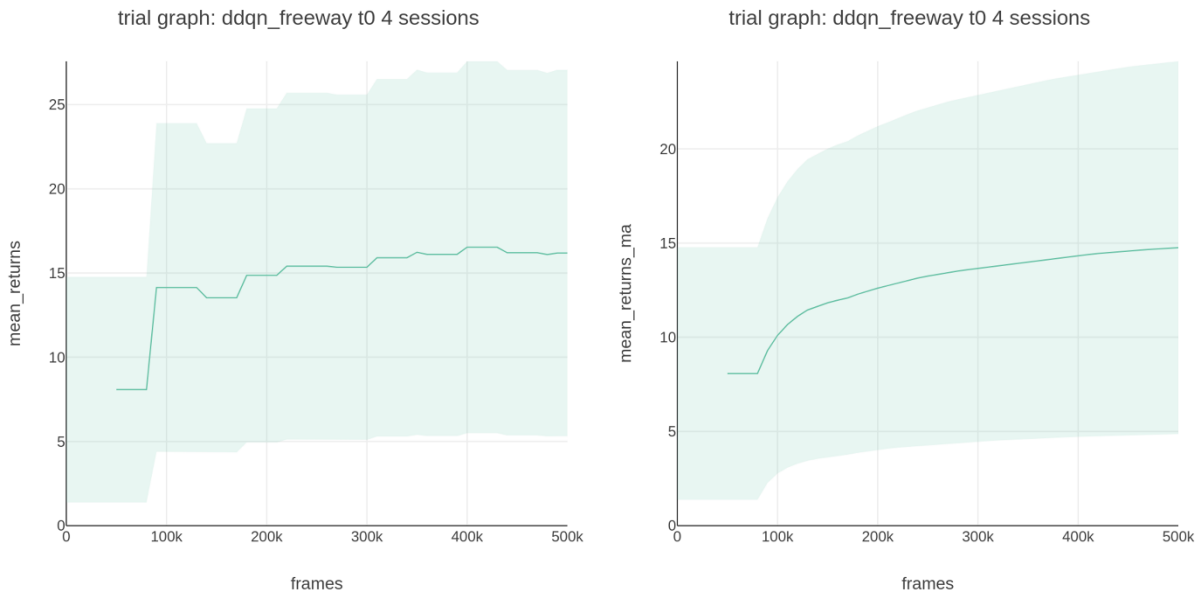


Figure 19. DDQN graphs.

```
{
  "ddqn_freeway": {
    "agent": [
      {
        "name": "DoubleDQN",
        "algorithm": {
          "name": "DoubleDQN",
          "action_pdtype": "Argmax",
          "action_policy": "boltzmann",
          "explore_var_spec": {
            "name": "linear_decay",
            "start_val": 5.0,
            "end_val": 0.5,
            "start_step": 0,
            "end_step": 1000
          },
          "gamma": 0.99,
          "training_batch_iter": 3,
          "training_iter": 3,
          "training_frequency": 1,
          "training_start_step": 32
        },
        "memory": {
          "name": "Replay",
          "batch_size": 32,
          "max_size": 10000,
          "use_cer": false
        },
        "net": {
          "type": "ConvNet",
          "shared": true,
          "conv_hid_layers": [
            [32, 8, 4, 0, 1],
            [64, 4, 2, 0, 1],
            [32, 3, 1, 0, 1]
          ],
          "fc_hid_layers": [512],
          "hid_layers_activation": "relu",
          "init_fn": "orthogonal_",
          "normalize": true,
          "batch_norm": false,
          "clip_grad_val": 0.5,
          "loss_spec": {
            "name": "SmoothL1Loss"
          },
          "optim_spec": {
            "name": "Adam",
            "lr": 0.001
          },
          "lr_scheduler_spec": null,
          "gpu": true
        }
      }
    ],
    "env": [{
        "name": "Freeway-v0",
        "frame_op": "concat",
        "frame_op_len": 4,
        "reward_scale": "sign",
        "num_envs": 16,
        "max_t": null,
        "max_frame": 500000
      }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "eval_frequency": 10000,
      "log_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 20. DDQN config file.

3.6 DQN PER

The graphs can be observed in Figure 21 and the configuration file can be observed in Figure 22.

- **Algorithm**: The algorithm is DQN PER (line:7). The action policy is Boltzmann policy (line:9). Entropy with a linear decay schedule (line:11).
- **Optimizer**: Adam (line:49), with a learning rate of 0.001 (line:50). The learning rate is constant because the learning rate scheduler is set to null (line:52).
- **Training frequency**: *PrioritizedExperienceReplay* memory is used for sampling batches (line:24).
- **Evaluation**: The agent is evaluated every 10,000 steps (line:72).

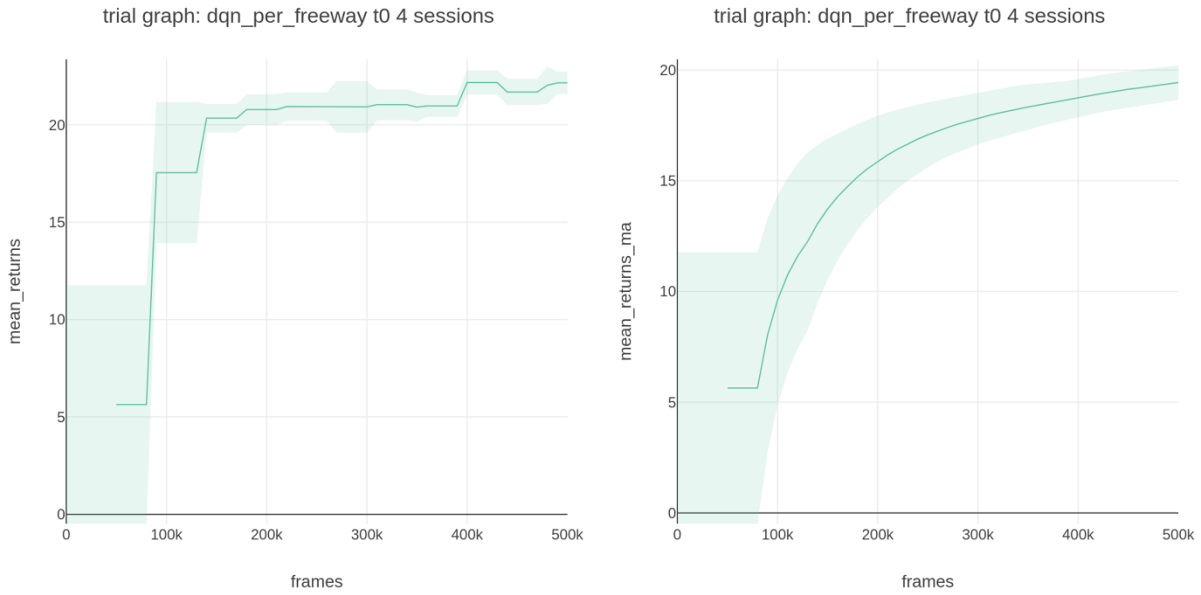


Figure 21. DQN PER graphs.

3.7 A2C n-step returns

The graphs can be observed in Figure 23 and the configuration file can be observed in Figure 24.

- **Algorithm**: The algorithm is A2C (line:6) with n-step returns (line:12). Entropy with no decay (line:14).
- **Optimizer**: Same settings for both actor and critic. RMSprop (line:45, line:51), with a learning rate of 0.0007 (line:46, line:52). The learning rate is constant because the learning rate scheduler is set to null (line:56).
- **Training frequency**: Training is episodic, having *OnPolicyReplay* memory (line:24). Training frequency is set to 5 (line:19), meaning that the network will be trained every 5 episodes.
- **Evaluation**: The agent is evaluated every 10,000 steps (line:75).

Observation: I tried various settings in the configuration file but none of them lead to the agent actually learning. The total reward was always 0. Fortunately, I had more luck with A2C GAE.


```
{
  "dqn_per_freeway": {
    "agent": [
      {
        "name": "DQN",
        "algorithm": {
          "name": "DQN",
          "action_pdtype": "Argmax",
          "action_policy": "boltzmann",
          "explore_var_spec": {
            "name": "linear_decay",
            "start_val": 5.0,
            "end_val": 0.5,
            "start_step": 0,
            "end_step": 1000
          },
          "gamma": 0.99,
          "training_batch_iter": 3,
          "training_iter": 3,
          "training_frequency": 1,
          "training_start_step": 32
        },
        "memory": {
          "name": "PrioritizedReplay",
          "alpha": 0.6,
          "epsilon": 0.0001,
          "batch_size": 32,
          "max_size": 10000,
          "use_cer": false
        },
        "net": {
          "type": "ConvNet",
          "shared": true,
          "conv_hid_layers": [
            [32, 8, 4, 0, 1],
            [64, 4, 2, 0, 1],
            [32, 3, 1, 0, 1]
          ],
          "fc_hid_layers": [512],
          "hid_layers_activation": "relu",
          "init_fn": "orthogonal_",
          "normalize": true,
          "batch_norm": false,
          "clip_grad_val": 0.5,
          "loss_spec": {
            "name": "SmoothL1Loss"
          },
          "optim_spec": {
            "name": "Adam",
            "lr": 0.001
          },
          "lr_scheduler_spec": null,
          "gpu": true
        }
      }
    ],
    "env": [{
        "name": "Freeway-v0",
        "frame_op": "concat",
        "frame_op_len": 4,
        "reward_scale": "sign",
        "num_envs": 16,
        "max_t": null,
        "max_frame": 500000
      }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "eval_frequency": 10000,
      "log_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```


Figure 22. DQN PER config file.

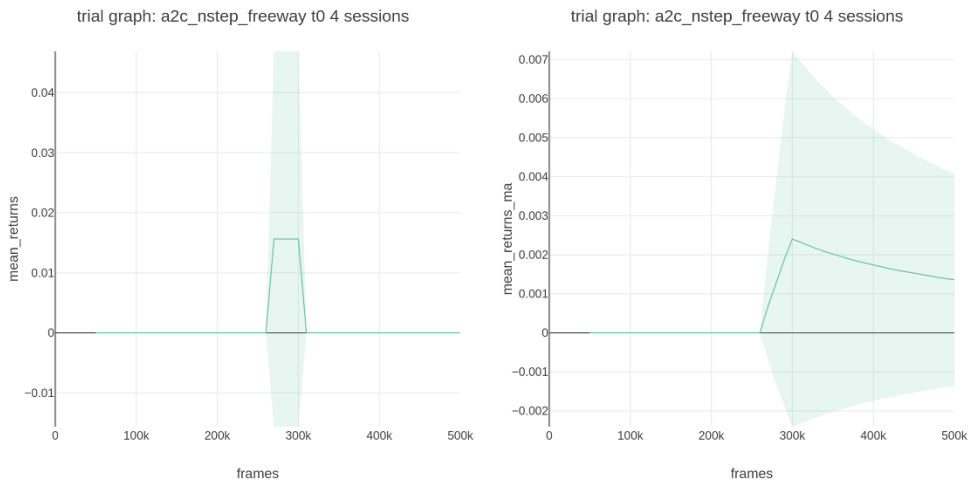
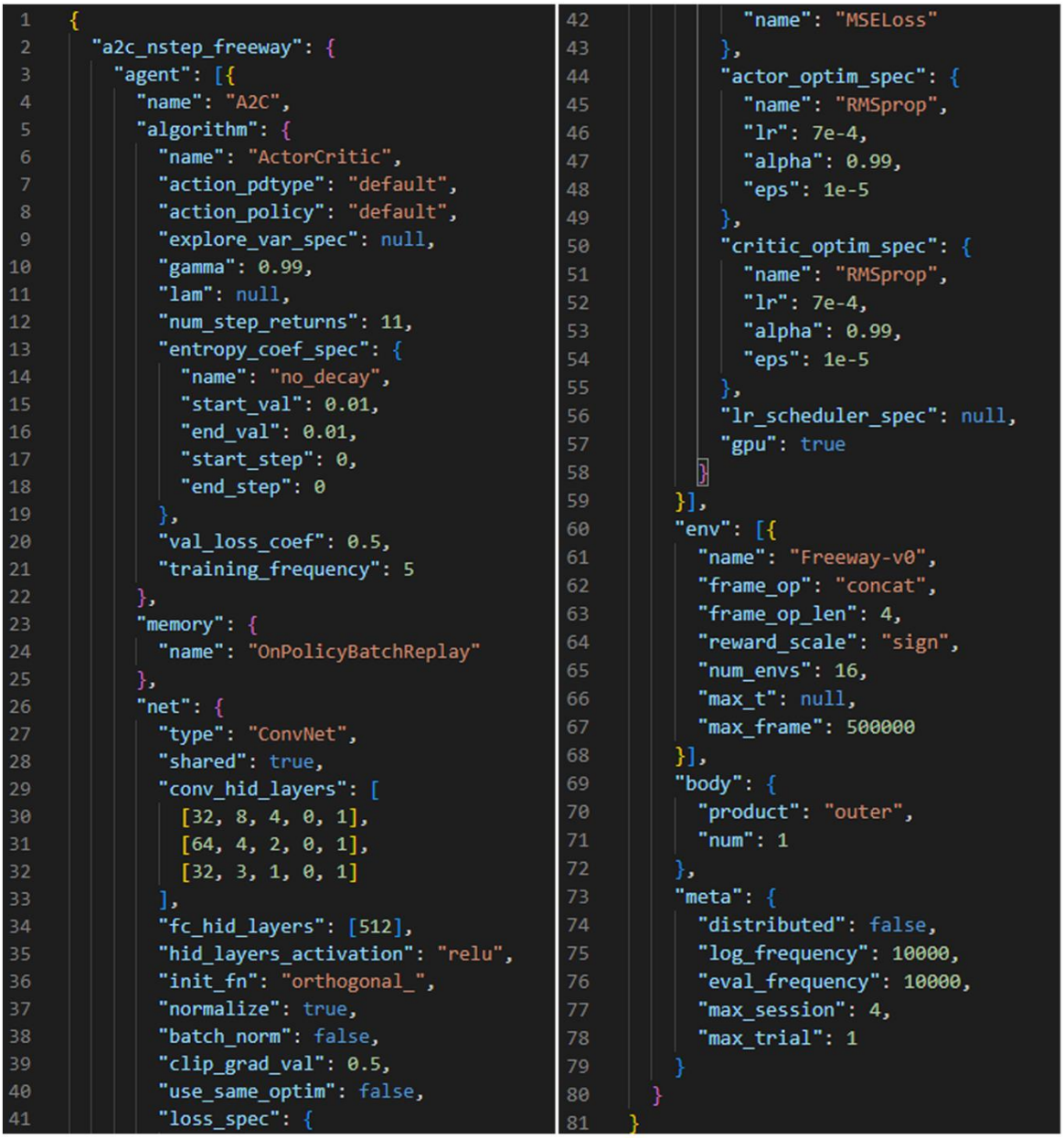


Figure 23. A2C n-step returns graphs.



```
{
  "a2c_nstep_freeway": {
    "agent": [{
      "name": "A2C",
      "algorithm": {
        "name": "ActorCritic",
        "action_pdtype": "default",
        "action_policy": "default",
        "explore_var_spec": null,
        "gamma": 0.99,
        "lam": null,
        "num_step_returns": 11,
        "entropy_coef_spec": {
          "name": "no_decay",
          "start_val": 0.01,
          "end_val": 0.01,
          "start_step": 0,
          "end_step": 0
        },
        "val_loss_coef": 0.5,
        "training_frequency": 5
      },
      "memory": {
        "name": "OnPolicyBatchReplay"
      },
      "net": {
        "type": "ConvNet",
        "shared": true,
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [32, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [512],
        "hid_layers_activation": "relu",
        "init_fn": "orthogonal_",
        "normalize": true,
        "batch_norm": false,
        "clip_grad_val": 0.5,
        "use_same_optim": false,
        "loss_spec": {
          "name": "MSELoss"
        },
        "actor_optim_spec": {
          "name": "RMSprop",
          "lr": 7e-4,
          "alpha": 0.99,
          "eps": 1e-5
        },
        "critic_optim_spec": {
          "name": "RMSprop",
          "lr": 7e-4,
          "alpha": 0.99,
          "eps": 1e-5
        },
        "lr_scheduler_spec": null,
        "gpu": true
      }
    }],
    "env": [{
      "name": "Freeway-v0",
      "frame_op": "concat",
      "frame_op_len": 4,
      "reward_scale": "sign",
      "num_envs": 16,
      "max_t": null,
      "max_frame": 500000
    }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "log_frequency": 10000,
      "eval_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 24. A2C n-step returns config file.

3.8 A2C GAE

The graphs can be observed in Figure 25 and the configuration file can be observed in Figure 26.

- **Algorithm**: The algorithm is ActorCritic (line:6) with Generalized Advantage Estimation (line:11). Entropy with no decay (line:14).
- **Optimizer**: Same settings for both actor and critic. Adam (line:45, line:49), with a learning rate of 0.002 (line:46, line:50). The learning rate is constant because the learning rate scheduler is set to null (line:52).
- **Training frequency**: Training is episodic, having *OnPolicyReplay* memory (line:24). Training frequency is set to 32 (line:21), meaning that the network will be trained every 32 episodes.
- **Evaluation**: The agent is evaluated every 10,000 steps (line:71).

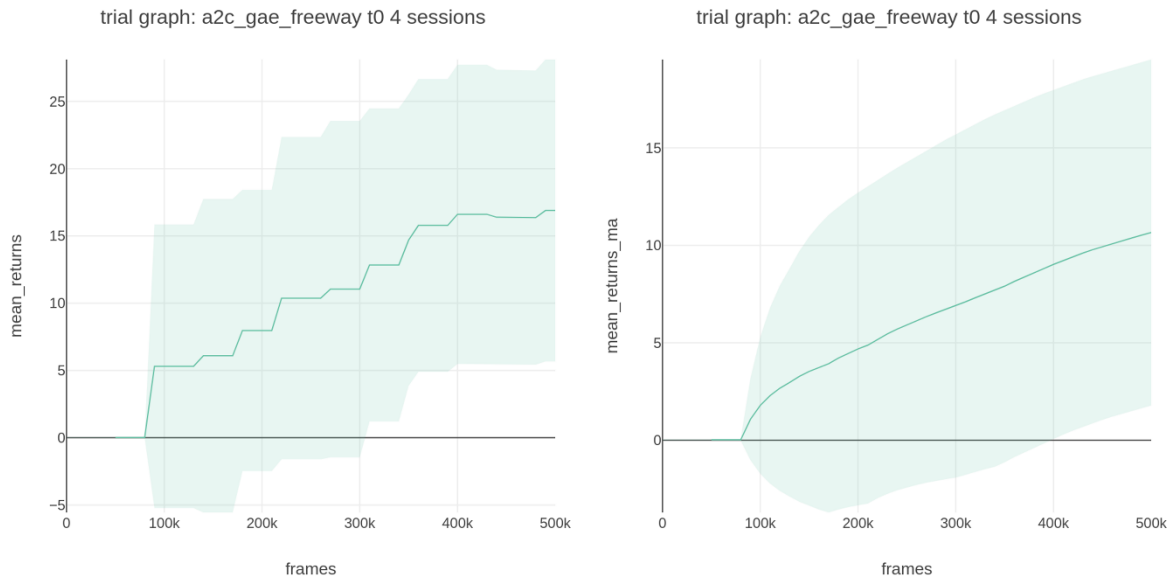


Figure 25. A2C GAE graphs.

```
{
  "a2c_gae_freeway": {
    "agent": [{
      "name": "A2C",
      "algorithm": {
        "name": "ActorCritic",
        "action_pdtype": "default",
        "action_policy": "default",
        "explore_var_spec": null,
        "gamma": 0.99,
        "lam": 0.95,
        "num_step_returns": null,
        "entropy_coef_spec": {
          "name": "no_decay",
          "start_val": 0.01,
          "end_val": 0.01,
          "start_step": 0,
          "end_step": 0
        },
        "val_loss_coef": 0.5,
        "training_frequency": 32
      },
      "memory": {
        "name": "OnPolicyBatchReplay"
      },
      "net": {
        "type": "ConvNet",
        "shared": true,
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [32, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [512],
        "hid_layers_activation": "relu",
        "init_fn": "orthogonal_",
        "normalize": true,
        "batch_norm": false,
        "clip_grad_val": 0.5,
        "use_same_optim": false,
        "loss_spec": {
          "name": "MSELoss"
        },
        "actor_optim_spec": {
          "name": "Adam",
          "lr": 0.002
        },
        "critic_optim_spec": {
          "name": "Adam",
          "lr": 0.002
        },
        "lr_scheduler_spec": null,
        "gpu": true
      }
    }],
    "env": [{
      "name": "Freeway-v0",
      "frame_op": "concat",
      "frame_op_len": 4,
      "reward_scale": "sign",
      "num_envs": 16,
      "max_t": null,
      "max_frame": 500000
    }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "log_frequency": 10000,
      "eval_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 26. A2C GAE config file.

3.9 PPO

The graphs can be observed in Figure 27 and the configuration file can be observed in Figure 28.

- **Algorithm**: The algorithm is PPO (line:6) and the action policy is the default policy (line:8). GAE is used to estimate advantages (line:11). Entropy with no decay (line:13).
- **Optimizer**: Same settings for both actor and critic. Adam (line:53, line:57), with a learning rate of 0.001 (line:54, line:58). The learning rate is constant because the learning rate scheduler is set to null (line:60).
- **Training frequency**: Training is episodic, having *OnPolicyReplay* memory (line:24).
- **Evaluation**: The agent is evaluated every 10,000 steps (line:80).

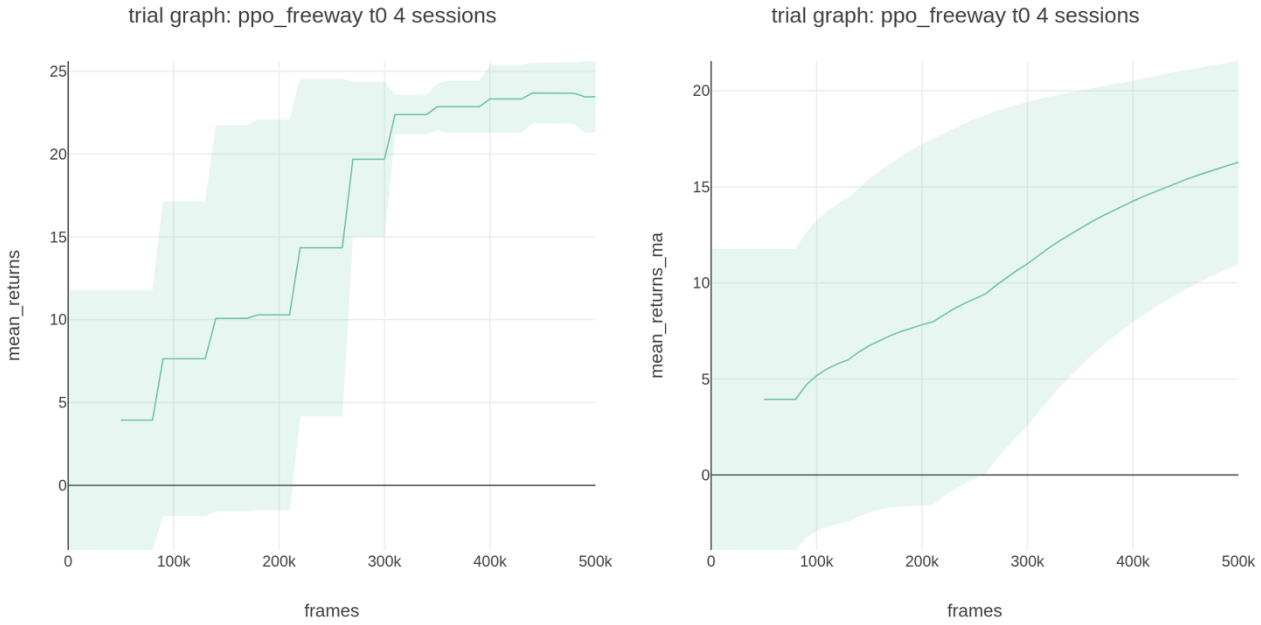


Figure 27. PPO graphs.

3.10 A3C

It was trained two times, both with 8 and 16 max sessions (line:73). The graphs for the 8 sessions can be observed in Figure 29 and the graphs for the 16 sessions can be observed in Figure 30. The configuration file can be observed in Figure 31.

- **Algorithm**: The algorithm is ActorCritic (line:6) with Generalized Advantage Estimation (line:11). Entropy with no decay (line:14).
- **Optimizer**: Same settings for both actor and critic. GlobalAdam (line:45, line:49), with a learning rate of 0.007 (line:46, line:50). The learning rate is constant because the learning rate scheduler is set to null (line:52).
- **Training frequency**: Training is episodic, having *OnPolicyReplay* memory (line:24). Training frequency is set to 32 (line:21), meaning that the network will be trained every 32 episodes.
- **Evaluation**: The agent is evaluated every 1,000 steps (line:71).

```
{
  "ppo_freeway": {
    "agent": [{
      "name": "PPO",
      "algorithm": {
        "name": "PPO",
        "action_pdtype": "default",
        "action_policy": "default",
        "explore_var_spec": null,
        "gamma": 0.99,
        "lam": 0.7,
        "clip_eps_spec": {
          "name": "no_decay",
          "start_val": 0.1,
          "end_val": 0.1,
          "start_step": 0,
          "end_step": 0
        },
        "entropy_coef_spec": {
          "name": "no_decay",
          "start_val": 0.01,
          "end_val": 0.01,
          "start_step": 0,
          "end_step": 0
        },
        "val_loss_coef": 0.5,
        "time_horizon": 128,
        "minibatch_size": 256,
        "training_epoch": 4
      },
      "memory": {
        "name": "OnPolicyBatchReplay"
      },
      "net": {
        "type": "ConvNet",
        "shared": true,
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [32, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [512],
        "hid_layers_activation": "relu",
        "init_fn": "orthogonal_",
        "normalize": true,
        "batch_norm": false,
        "clip_grad_val": 0.5,
        "use_same_optim": false,
        "loss_spec": {
          "name": "MSELoss"
        },
        "actor_optim_spec": {
          "name": "Adam",
          "lr": 0.001
        },
        "critic_optim_spec": {
          "name": "Adam",
          "lr": 0.001
        },
        "lr_scheduler_spec": null,
        "gpu": true
      }
    }],
    "env": [{
      "name": "Freeway-v0",
      "frame_op": "concat",
      "frame_op_len": 4,
      "reward_scale": "sign",
      "num_envs": 16,
      "max_t": null,
      "max_frame": 500000
    }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "log_frequency": 10000,
      "eval_frequency": 10000,
      "max_session": 4,
      "max_trial": 1
    }
  }
}
```

Figure 28. PPO config file.

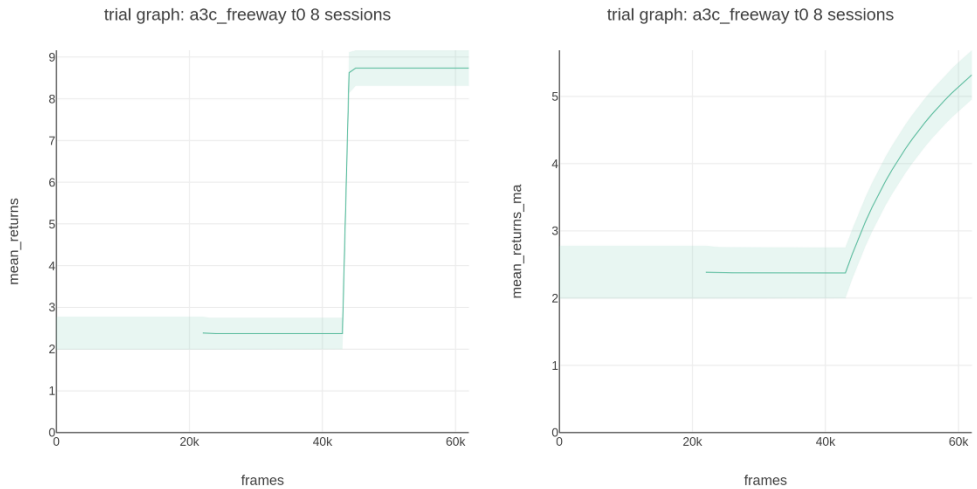


Figure 29. A3C 8 sessions graphs.

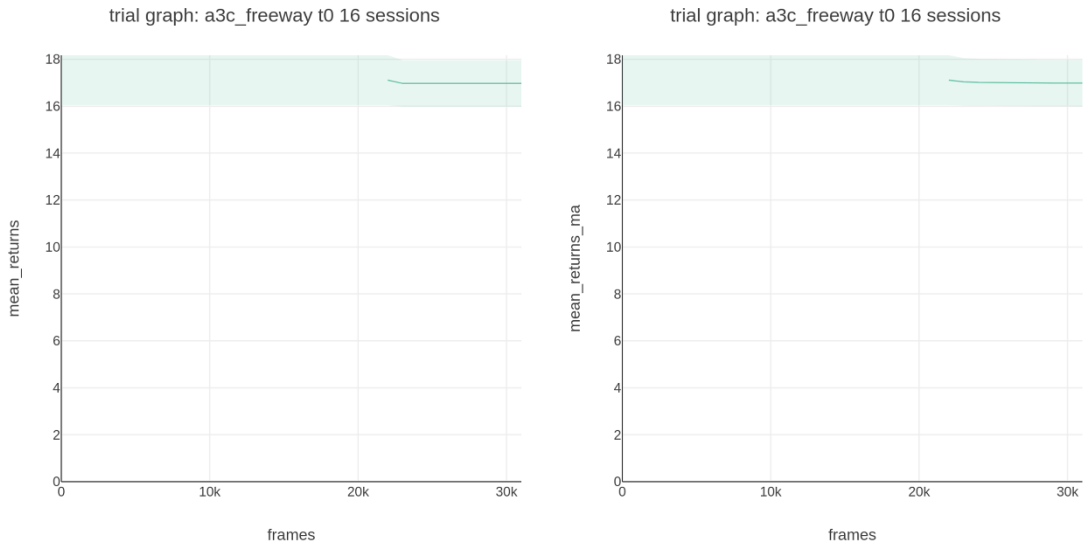


Figure 30. A3C 16 sessions graphs.

```
{
  "a3c_freeway": {
    "agent": [{
      "name": "A3C",
      "algorithm": {
        "name": "ActorCritic",
        "action_pdtype": "default",
        "action_policy": "default",
        "explore_var_spec": null,
        "gamma": 0.99,
        "lam": 0.95,
        "num_step_returns": null,
        "entropy_coef_spec": {
          "name": "no_decay",
          "start_val": 0.01,
          "end_val": 0.01,
          "start_step": 0,
          "end_step": 0
        },
        "val_loss_coef": 0.5,
        "training_frequency": 32
      },
      "memory": {
        "name": "OnPolicyBatchReplay"
      },
      "net": {
        "type": "ConvNet",
        "shared": true,
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [32, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [512],
        "hid_layers_activation": "relu",
        "init_fn": "orthogonal_",
        "normalize": true,
        "batch_norm": false,
        "clip_grad_val": 0.5,
        "use_same_optim": false,
        "loss_spec": {
          "name": "MSELoss"
        },
        "actor_optim_spec": {
          "name": "GlobalAdam",
          "lr": 0.007
        },
        "critic_optim_spec": {
          "name": "GlobalAdam",
          "lr": 0.007
        },
        "lr_scheduler_spec": null,
        "gpu": false
      }
    }],
    "env": [{
      "name": "Freeway-v0",
      "frame_op": "concat",
      "frame_op_len": 4,
      "reward_scale": "sign",
      "num_envs": 8,
      "max_t": null,
      "max_frame": 500000
    }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": "synced",
      "log_frequency": 1000,
      "eval_frequency": 1000,
      "max_session": 8,
      "max_trial": 1
    }
  }
}
```

Figure 31. A3C config file.

4. Conclusions

For the purpose of understanding the results obtained and comparing the different algorithms, I will present three metrics.

Strength refers to the average performance of the agent over the entire duration of the trial. This metric is usually calculated as the mean of the returns (cumulative rewards) obtained by the agent across all episodes or evaluation points during the trial. It provides an overall view of the agent's average performance, and it is useful for understanding general learning effectiveness.

Max Strength represents the highest performance achieved by the agent at any point during the trial. This metric is determined by identifying the maximum return (cumulative reward) obtained in any single episode or evaluation point. It provides information on the best possible performance the agent achieved, and it is useful for understanding the highest potential of the agent.

Final Strength indicates the performance of the agent at the end of the trial. This metric is typically the mean of the returns (cumulative rewards) over the last few episodes or evaluations, providing a measure of the agent's performance after completing its training. It reflects how well the agent performs after it has had the opportunity to learn from the entire training process.

Algorithm	Strength	Max Strength	Final Strength
REINFORCE	1.9053	2.3750	2.1666
SARSA	4.3648	5.5781	5.3750
DQN	11.8797	20.9218	19.7968
DQN w/ target nets	17.5363	21.5468	21.4687
DDQN	14.7540	16.8125	16.1875
DQN PER	19.4317	22.3437	22.1562
A2C n-step returns	0.0013	0.0156	0.0000
A2C GAE	10.6579	17.1875	16.8906
PPO	16.2700	24.2968	23.4531
A3C 8 sessions	5.3196	8.7343	8.7343
A3C 16 sessions	16.9825	17.1618	16.9687

In conclusion, considering the presented metrics, the best overall score during the whole training process was achieved by DQN PER. Also, the best result and the best result after training was achieved by PPO. However, these results do not prove that one algorithm is better than another. The agents were trained on Google Colab and the results might be influenced by the hardware limitations of this platform.

5. Reference List

[1] Gymnasium Documentation. https://gymnasium.farama.org/environments/atari/freeway/

[2] Atari Age. https://atariage.com/manual_html_page.php?SoftwareLabelID=192

[3] Internet Archive. https://archive.org/details/FreewayAtari2600900DPIScans

[4] Reinforcement Learning Course, Prof. Dr. Habil. Eng. Călin-Adrian POPA, 2025

[5] SLM Lab. https://slm-lab.gitbook.io/slm-lab