Deep Reinforcement Learning:

DQN, Double DQN, Dueling DQN, & Prioritized Experience Replay

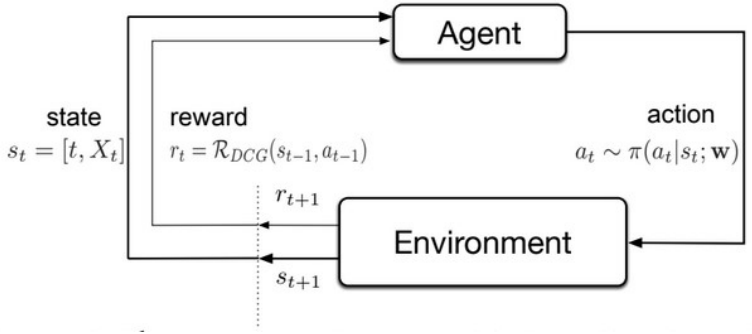
James Chapman
CIS 730 Artificial Intelligence– Term Project
Kansas State University
jachapman@ksu.edu

Overview

- Background - Reinforcement Learning
- Deep Reinforcement Learning
- Experiment Design
  - Gymnasium
  - Training - Pong & Breakout
- Testing & Evaluation
- Results
- Discussion

Background - Reinforcement Learning (1 of 4)

- "Classic" Reinforcement Learning
  - Machine learning
  - Control theory



Sutton, R. S., Barto, A. G. (2018). Reinforcement Learning: An Introduction. The MIT Press.

- Markov Decision Process
  - Mathematical Framework for Dynamics
  - Probabilistic state-transition
  - s, a, P, r

$$P(s'|s,a) = \Pr(s_t = s'|s_{t-1} = s, a_{t-1} = a)$$

Background - Reinforcement Learning (2 of 4)

- State-Value Function
  - Prediction of the cumulative reward, given policy π

$$V^{\pi}(s) = \mathbb{E}[\, R_t \mid S_t = s \,]$$

- Bellman equations
  - Recursive definitions
  - Long-term values
  - Discounted γ

$$V^{\pi}(s) = \sum_{s'} P(s'|s,a)[R + \gamma V^{\pi}(s')]$$

Background - Reinforcement Learning (3 of 4)

- State-Action-Value Function —> Q-Learning
  - Prediction of the cumulative reward, given policy π
  - Discounted γ
  - Bellman equation

$$Q^{\pi}(s,a) = \sum_{s',r} P(s',r|s,a)\left[r + \gamma \sum_{a'} \pi(a'|s') Q^{\pi}(s',a')\right]$$

  - Optimal Q^* equation

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

Background - Reinforcement Learning (4 of 4)

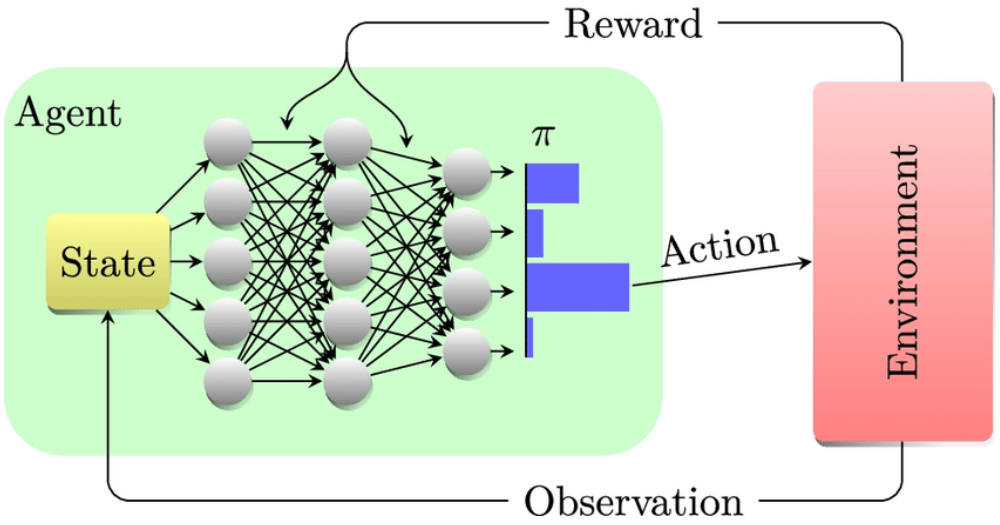
- Monte Carlo methods
  - Sparse rewards

- Temporal difference learning
  - Bootstrapping
  - $Q^{\pi-}$: fixed & separate policy

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

Deep Reinforcement Learning

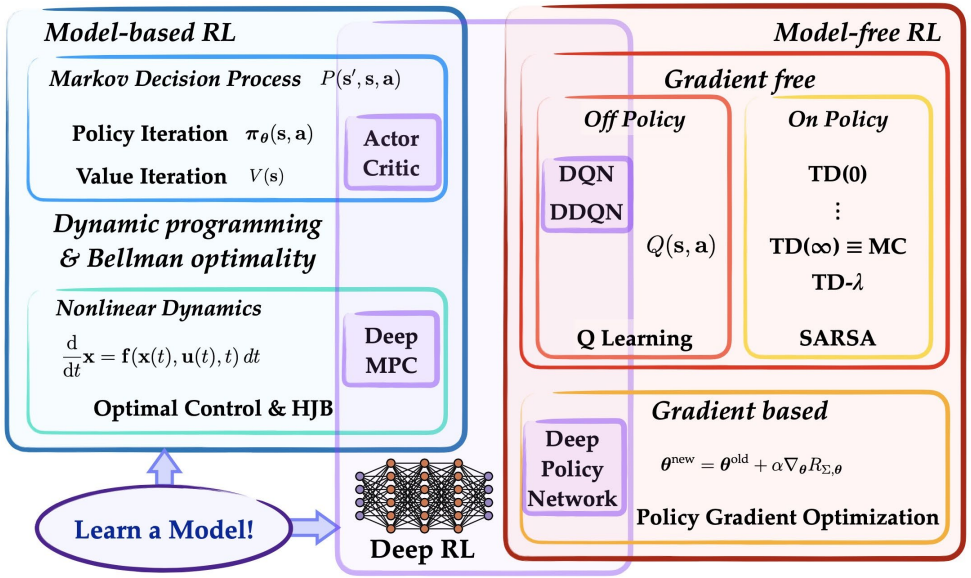
- Neural networks are universal function approximators!
  - Learn Q- function
  - Q(State-Action)



Dalla Pozza, N., Buffoni, L., Martina, S., & Caruso, F. (2021). Quantum reinforcement learning: the maze problem. Quantum Machine Intelligence, 4. doi.org/10.48550/arXiv.2108.04490

Deep Reinforcement Learning

- Deep Q-Networks (DQN)
- Double DQN
- Dueling DDQN
- Prioritized Experience Replay (PER)



Brunton, S. L., & J. Nathan Kutz. (2019). Data-Driven Science and Engineering. Cambridge University Press.

Deep Q-Networks (DQN) (1 of 2)

- High-dimensional sensory/feature space
- Deep Q-Networks (DQN) - Mnih, et al. (2013)
  - Temporal Difference Loss Function
  - In terms of neural network weights, θ
  - θ^- weights of fixed and separate network

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

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. doi.org/10.48550/arXiv.1312.5602

Deep Q-Networks (DQN) (2 of 2)

- Off-policy
  - action with highest Q-value
  - e greedy
- Experience Replay Buffer
  - Correlation (i.i.d)
  - Batch update

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. doi.org/10.48550/arXiv.1312.5602

Double DQN

- Double DQN - Hasselt, et al. (2016)
  - DQN overestimates
  - Change action selection

DQN

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

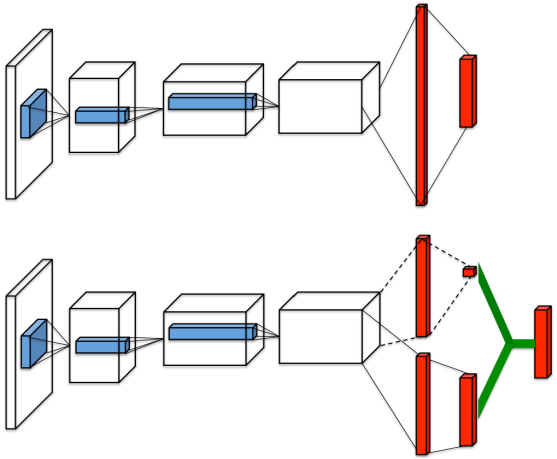
Double DQN

$$L(\theta) = E\left[\left(r + \gamma\, Q\left(s', \operatorname*{argmax}_{a'} Q\left(s', a'; \theta\right); \theta^{-}\right) - Q(s, a; \theta)\right)^2\right]$$

Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016. doi.org/10.48550/arXiv.1509.06461

Dueling DQN

- Dueling DQN - Wang, et al. (2016)
- New Q-value
- Split fully connected layer
  - "Value" of state
  - "Advantage" of actions



$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a))$$

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. 2016. Dueling network architectures for deep reinforcement learning. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16). JMLR.org, 1995–2003. doi.org/10.48550/arXiv.1511.06581

Prioritized Experience Replay (1 of 2)

- "Standard"experience replay
  - samples randomly
- Prioritized Experience Replay - Schaul et al. (2016)
  - Some samples provide more information (surprise)
  - " Prioritize " by temporal difference (TD)
    - Probability $P(i)$ of being sampled

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015). Prioritized Experience Replay Published at ICLR 2016 doi.org/10.48550/arXiv.1511.05952

Prioritized Experience Replay (2 of 2)

- Hyperparameters
  - α (α=0, standard experience replay)
  - β (anneals to 1)
- Weights w_i
- Updated loss function

$$P(i) = \frac{P(i)^\alpha}{\sum_k P(k)^\alpha}$$

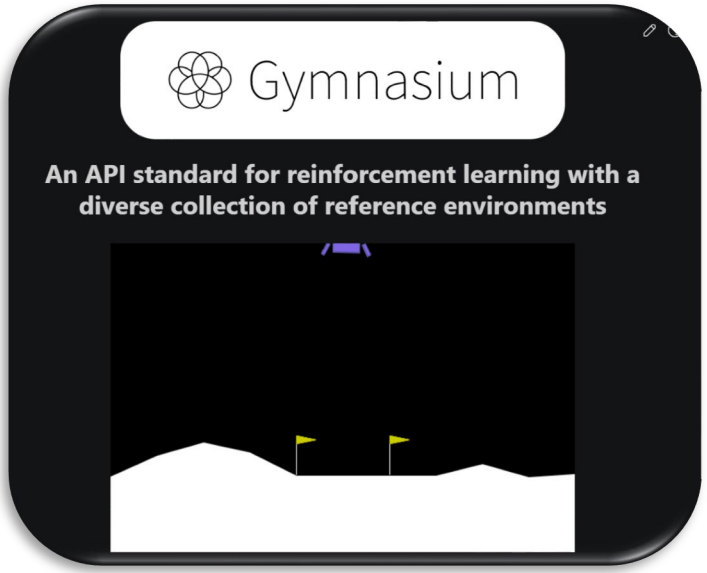
$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

$$\text{Loss} = \sum_i w_i \cdot \left(y_i - Q(s_i, a_i; \theta)\right)^2$$

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015). Prioritized Experience Replay Published at ICLR 2016 doi.org/10.48550/arXiv.1511.05952

Experiment Design

- OpenAI – Gym
- Farmara - Gymnasium
  - Arcade Learning Environment (ALE)
    - Atari emulator
  - 210 x 160 pixel RGB images @ 60 Hz
  - Up to 18 actions



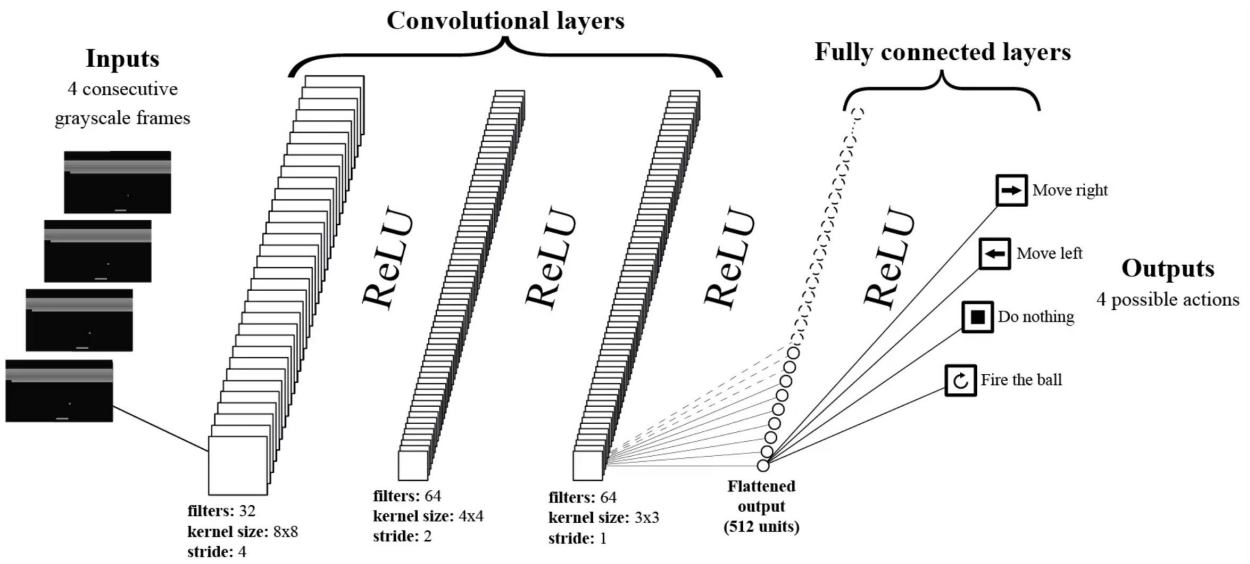
- gymnasium.Env.**reset()**
- gymnasium.Env.**step(action)**

  return game score & lives, etc.

Towers, M. et al. (2023, March 25). Gymnasium API. Zenodo. https://doi.org/10.5281/zenodo.8127026

Experiment Design

- Deep Q-Networks (DQN) - Mnih, et al. (2013)

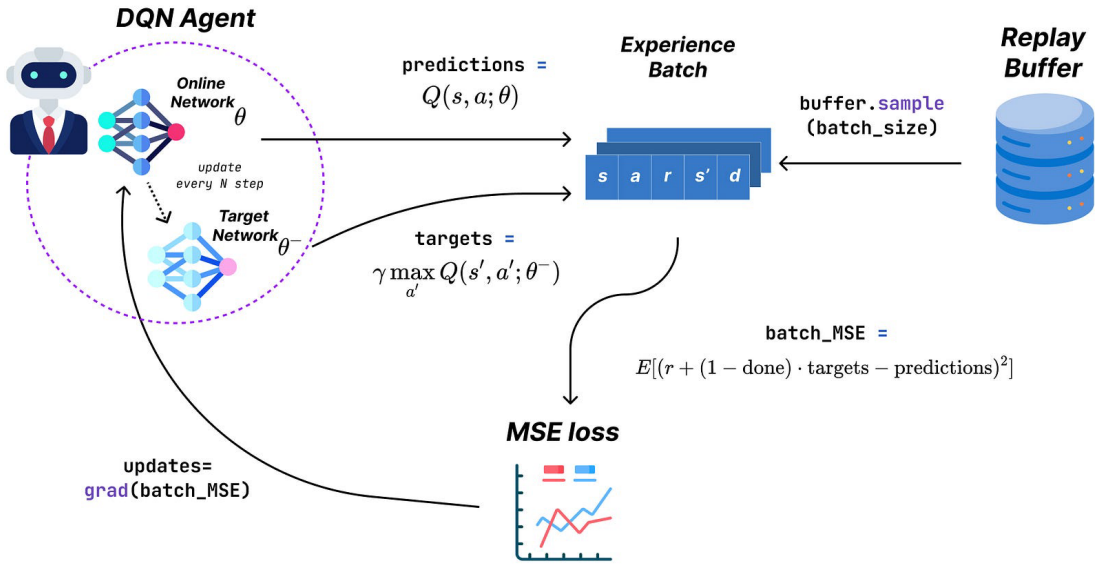


Experiment Design

- Beginning of this project
- "Deep Reinforcement Learning Explained" by Jordi Torres
  - Practical tutorial with starter code
  - OpenAI Gym Pong
  - PROBLEMATIC – Gymnasium

https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/tree/master

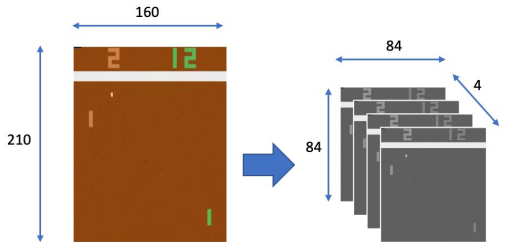
- TRAINING
  - Wrappers
  - Agent
  - Replay Buffer
  - Q-Neural Network



https://towardsdatascience.com/a-gentle-introduction-to-deep-reinforcement-learning-in-jax-c1e45a179b92

- WRAPPERS
  - Processing environment

```
def PongWrappers(env):
    print('Creating environment------------')
    env = MaxAndSkipEnv(env, 4) # From stable_baselines3
    env = FireResetEnv(env)  # From stable_baselines3
    env = NoopResetEnv(env, noop_max=noop_max)  # From stable_baselines3

    print(env.observation_space)
    env = ProcessFrame84(env)
    print(env.observation_space)
    env = ImageToPyTorch(env)
    print(env.observation_space)
    env = BufferWrapper(env, 4)
    print(env.observation_space)
    env = ScaledFloatFrame(env)
    print(env.observation_space)
    print('--------------------------------')
    return env
```




```
Creating environment------------
Box(0, 255, (210, 160, 3), uint8)
Box(0, 255, (84, 84, 1), uint8)
Box(0.0, 1.0, (1, 84, 84), float32)
Box(0.0, 1.0, (4, 84, 84), float32)
Box(0.0, 1.0, (4, 84, 84), float32)
--------------------------------
```

- AGENT
  - Controlling actions & interactions with the environment

env.reset()

env.step(action)

```python
class Agent:
    def __init__(self, env, exp_buffer, action_repeat):
        self.env = env
        self.exp_buffer = exp_buffer
        self.action_repeat = action_repeat
        self._reset()

    def _reset(self):
        self.state, info = env.reset()
        self.lives = 0#self.env.ale.lives()
        self.total_reward = 0.0
        self.action_repeat_count = 0
        self.action_repeat_action = env.action_space.sample()

    def play_step(self, net, epsilon, device):
        # Choose Action
        if self.action_repeat_count < self.action_repeat:
            action = self.action_repeat_action
        elif np.random.random() < epsilon:
            action = env.action_space.sample()
        else:
            state_a = np.array([self.state], copy=False)
            state_v = torch.tensor(state_a).to(device)
            with torch.inference_mode():
                q_vals_v = net(state_v)
                _, act_v = torch.max(q_vals_v, dim=1)
            action = int(act_v.item())

        # Take Action
        new_state, reward, terminated, truncated, info = self.env.step(action) # ?clipped_reward
        self.action_repeat_action = action
        self.action_repeat_count += 1
        is_done = terminated or truncated
        self.total_reward += reward # ?np.clip(reward, -1, 1)?np.sign(float(reward))

        # IMPORTANT
        # Gymnasium doesn't have built in method to check if Life was lost
        # This changes the done_mask in the Q function
        if self.lives != self.env.ale.lives():
            self.lives = self.env.ale.lives()
            terminated = True
```

- EXPERIENCE REPLAY
  - (PER implementation is long)

```
Experience = collections.namedtuple('Experience', field_names=['state', 'action', 'reward', 'done', 'new_state'])

class ExperienceReplay:
    def __init__(self, capacity):
        self.buffer = collections.deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        indices = np.random.choice(len(self.buffer), batch_size, replace=False)
        states, actions, rewards, dones, next_states = zip(*[self.buffer[idx] for idx in indices])
        return np.array(states), np.array(actions), np.array(rewards, dtype=np.float32), \
               np.array(dones, dtype=np.uint8), np.array(next_states)
```

- NEURAL NETWORK
  - Q-function

Returns Q-values for each action

```
class ConvNet(nn.Module):
    def __init__(self, input_shape, n_actions, is_dueling):
        super(ConvNet, self).__init__()
        self.input_shape = input_shape
        self.n_actions = n_actions
        self.is_dueling = is_dueling

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
        self.conv_out_size = self._get_conv_out(input_shape)#(3136)
        self.fc = nn.Sequential(
            nn.Linear(self.conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )
        self.dueling_value = nn.Sequential(
            nn.Linear(self.conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )

    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        if is_dueling:
            ############## Dueling Networks ################
            conv_out = self.conv(x).view(x.size()[0], -1)
            advantage_out = self.fc(conv_out)
            value_out = self.dueling_value(conv_out)
            return value_out + advantage_out - advantage_out.mean()
            ################################################
        else:
            conv_out = self.conv(x).view(x.size()[0], -1)
            return self.fc(conv_out)
```

- TRAINING LOOP (for reference)

```
: while True:
    frame_idx+= 1
    game_frame_idx+= 1
    if frame_idx> num_frames:
        break

    epsilon = max(epsilon-eps_decay, eps_min) #Linear decay
    beta =  min(beta + beta_step, 1.0)
    reward = agent.play_step(net, epsilon, device=device)

    if reward is not None:
        total_rewards.append(reward)
        mean_rewards.append(np.mean(total_rewards[-100:]))
        std_rewards.append(np.std(total_rewards[-100:]))
        epsilons.append(epsilon)
        frame_idxs.append(frame_idx)
        game_frame_idx = 0 # reset the number of frames of the game

        if frame_idx > warmup_size:
            plot_rewards()

    if len(buffer) < warmup_size:
        continue

    if frame_idx % sync_target_frames == 0:
        target_net.load_state_dict(net.state_dict())

    if frame_idx > start_testing_idx and frame_idx % test_interval == 0:
        interval_SAVE_NAME = NETWORK_SAVE_PATH + '-{}.dat'.format(frame_idx)
        torch.save(net.state_dict(), interval_SAVE_NAME)

    if game_frame_idx % frame_learning_skip != 0:
        continue

    if is_prioritized_replay:
        ######### Prioritized Experience Replay ########
        batch_dict = buffer.sample(beta)

        states = batch_dict["obs"]
        actions = batch_dict["acts"]
        rewards = batch_dict["rews"]
        dones = batch_dict["done"]
        next_states = batch_dict["next_obs"]
        weights = torch.FloatTensor(
            batch_dict["weights"].reshape(-1, 1)
        ).to(device)
        indices = batch_dict["indices"]
        ################################################
    else :
        ######## Simple Experience Replay #############
        batch = buffer.sample(batch_size)
        states, actions, rewards, dones, next_states = batch
        ################################################
```

```
states_v = torch.tensor(states).to(device)
next_states_v = torch.tensor(next_states).to(device)
actions_v = torch.tensor(actions).to(device)
rewards_v = torch.tensor(rewards).to(device)
done_mask = torch.ByteTensor(dones).to(device)

if is_double:
    ################## Double DQN ##################
    with torch.no_grad():
        maxActions = net(next_states_v).max(1, keepdim=True)[1]
        next_state_values = target_net(next_states_v).gather( 1, maxActions).squeeze(-1)
    ################################################
else :
    ################### Simple DQN #############
    with torch.no_grad():
        maxActions = target_net(next_states_v).max(1, keepdim=True)[1]
        next_state_values = target_net(next_states_v).gather( 1, maxActions).squeeze(-1)
    ################################################

state_action_values = net(states_v).gather(1, actions_v.unsqueeze(-1).long()).squeeze(-1)
expected_state_action_values = next_state_values * gamma * (1 - done_mask) + rewards_v

if is_prioritized_replay:
    ######### Prioritized Experience Replay ########
    elementwise_loss = F.smooth_l1_loss(state_action_values, expected_state_action_values, reduction="none")
    loss_t = torch.mean(weights * elementwise_loss)

    td_error  = state_action_values - expected_state_action_values
    loss_for_prior = np.abs(td_error.cpu().detach().numpy())
    new_priorities = loss_for_prior + prior_eps
    buffer.update_priorities(indices, new_priorities)
    ################################################
else:
    ######### Simple Experience Replay #############
    loss_t = nn.MSELoss()(state_action_values, expected_state_action_values)
    ################################################

optimizer.zero_grad()
loss_t.backward()
optimizer.step()
```

Experiment Design

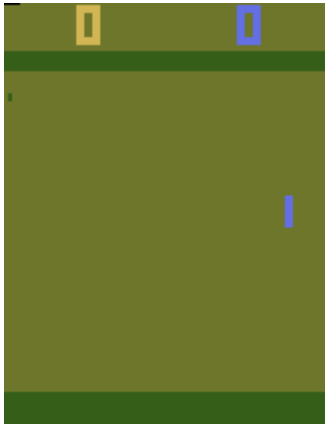
- HYPERPARAMETERS
  - 5 million frames
  - 500 frames - Sync target network
  - 1 million frames ε decay (linear)
    - 1.0 – 0. 01
  - Experience replay buffer
    - 100,000 experiences
  - Back propagation – every 4 steps
  - Action repeat - 0 steps
  - Gamma – 0.99
  - Batch size – 32
  - Learning rate – 0.0001

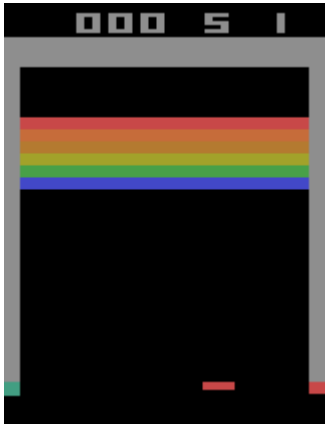
Hyperparameters & Control

```
# ENV_NAME = "PongNoFrameskip-v4"
ENV_NAME = "BreakoutNoFrameskip-v4"
# ENV_NAME = "SpaceInvadersNoFrameskip-v4"
# ENV_NAME = "MsPacmanNoFrameskip-v4"
# ENV_NAME = "QbertNoFrameskip-v4"
# ENV_NAME = "LunarLander-v2"

NETWORK_SAVE_PATH = 'models/Breakout-DUELING'
DATA_SAVE_PATH = 'data/Breakout-DUELING'

is_double = True
is_dueling = True
is_prioritized_replay = True
noop_max = 30
action_repeat = 0
frame_learning_skip = 4

gamma = 0.99
batch_size = 32
learning_rate = 1e-4

num_frames = 5e6
replay_size = 100000
warmup_size = 10000
sync_target_frames = 500 #1000 #DOUBLE, PER, DQN
start_testing_idx = 1e6
test_interval = 250000

eps_start=1.0
eps_min=0.01
eps_decay=9.9e-7 #(1-0.01)/1,000,000

alpha=0.2 # 0.6
beta = 0.6 # 0.4
prior_eps = 1e-6
beta_step = (1.0 - beta) / 1e6 # num_frames
```

Experiment Design

- PongNoFrameskip-v4
- BreakoutNoFrampskip-v4
  - DQN
  - Double DQN
  - PER
  - Dueling DQN w/ PER
- Test every 250,000 frames
  - save neural network parameters
  - Testing.ipynb





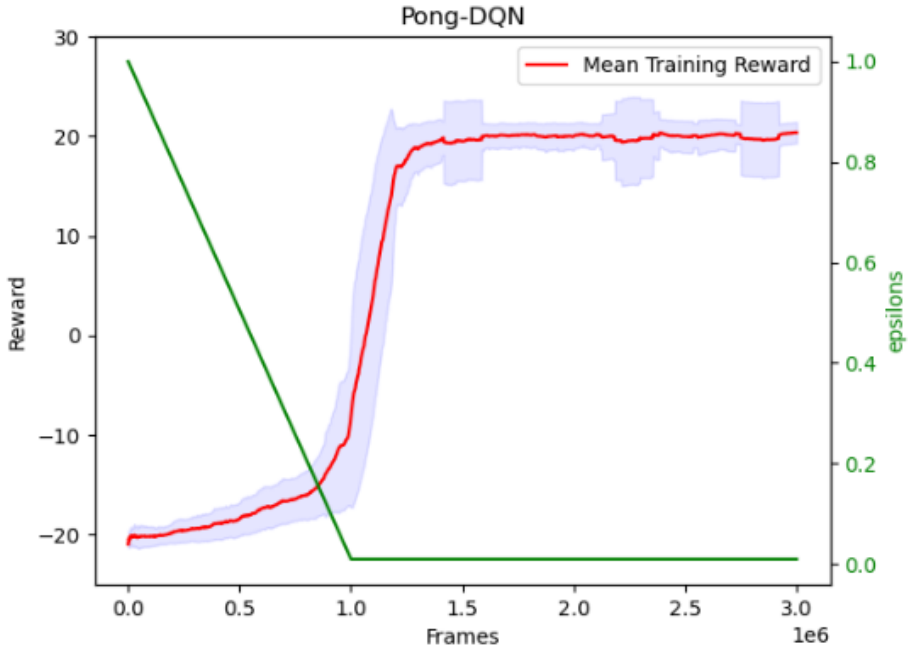
Experiment Design - TESTING

- For each model & interval
  - 150,000 frames
  - Introduce stochasticity
    - Generalization
    - e – 0.005
    - no-op max – 30
    - Mean & STD
      - Interval with highest mean value reported

Results

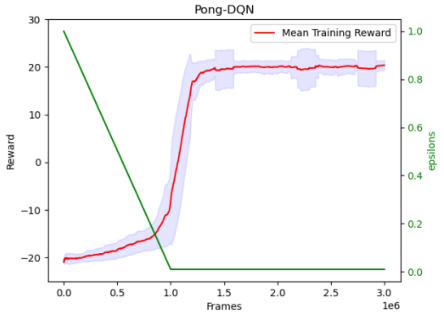
- Pong - Training Curves
  - Easy

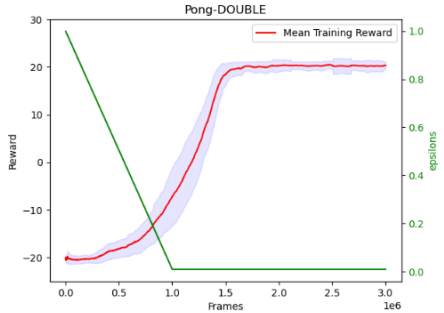
    (starter code)

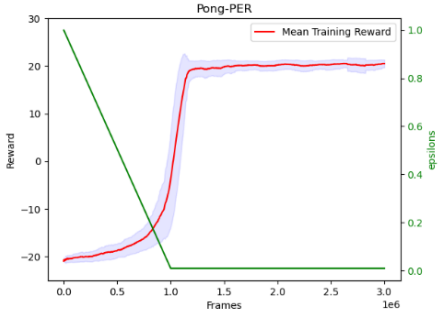


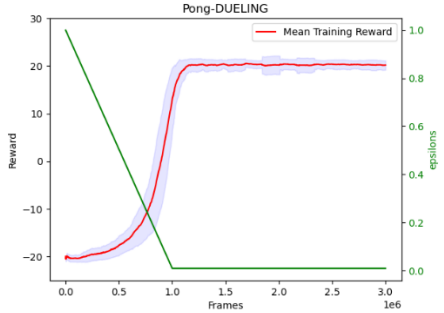
Results

- Pong - Training Curves





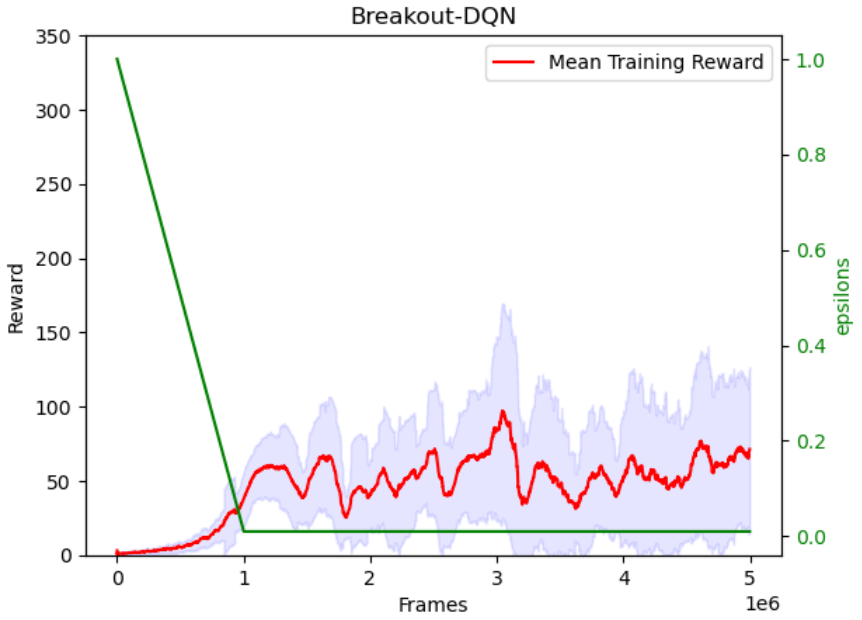




Results

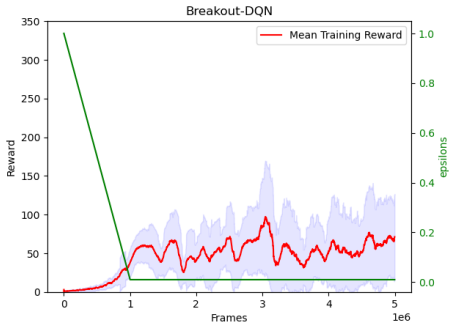
- Breakout – DQN Training Curve
  - Trial and error
  - Very sensitive
  - Exploration vs Exploitation

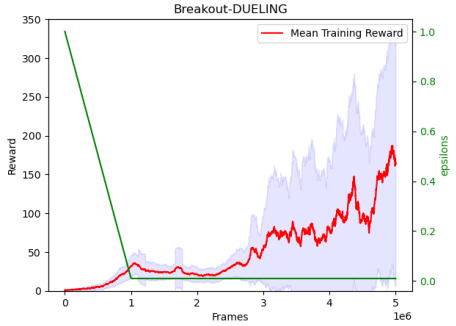
"The devil is in the details"



Results

- Breakout – Training Curves





Results

Model	Breakout	Pong
DQN	112.4 +/- 57.6	20.8 +/- 0.5
DOUBLE	96.5 +/- 43.5	20.8 +/- 0.5
PER	43.5 +/- 13.4	20.8 +/- 0.5
DUELING w/ PER	**207.3 +/- 124.0**	**20.9 +/- 0.4**
Random [1]	1.2	- 20.4
Human [1]	31	- 3
DQN [1]	168	20
Double DQN [2]	**375.0**	**21.0**
Double DQN w/ PER [3]	371.6	18.9
Dueling DDQN w/ PER [4]	366.0	20.9
State-of-the-art [5]	864	21

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. doi.org/10.48550/arXiv.1312.5602

[2] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016. doi.org/10.48550/arXiv.1509.06461

[3] Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015). Prioritized Experience Replay Published at ICLR 2016 doi.org/10.48550/arXiv.1511.05952

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. 2016. Dueling network architectures for deep reinforcement learning. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16). JMLR.org, 1995–2003. doi.org/10.48550/arXiv.1511.06581

[5] Papers with code - atari 2600 pong benchmark (Atari Games). The latest in Machine Learning. (n.d.). https://paperswithcode.com/sota/atari-games-on-atari-2600-pong

Discussion (1 of 3)

- High deviation
- Implementation details
  - Frame skipping
  - Action repeat
  - Reward clipping
  - Network sync rate
  - PER – rank/prop
  - Hyperparameter scheduling

Model	Breakout	Pong
DQN	112.4 +/- 57.6	20.8 +/- 0.5
DOUBLE	96.5 +/- 43.5	20.8 +/- 0.5
PER	43.5 +/- 13.4	20.8 +/- 0.5
DUELING w/ PER	**207.3 +/- 124.0**	**20.9 +/- 0.4**
Random [1]	1.2	- 20.4
Human [1]	31	- 3
DQN [1]	168	20
Double DQN [2]	**375.0**	**21.0**
Double DQN w/ PER [3]	371.6	18.9
Dueling DDQN w/ PER [4]	366.0	20.9
State-of-the-art [5]	864	21

Discussion (2 of 3)

- Training longer
  - (up to 200 million frames)
  - Parallel environments
- High RAM/GPU needs
  - Google Colab Pro +
  - Problematic

Model	Breakout	Pong
DQN	112.4 +/- 57.6	20.8 +/- 0.5
DOUBLE	96.5 +/- 43.5	20.8 +/- 0.5
PER	43.5 +/- 13.4	20.8 +/- 0.5
DUELING w/ PER	**207.3 +/- 124.0**	**20.9 +/- 0.4**
Random [1]	1.2	- 20.4
Human [1]	31	- 3
DQN [1]	168	20
Double DQN [2]	**375.0**	**21.0**
Double DQN w/ PER [3]	371.6	18.9
Dueling DDQN w/ PER [4]	366.0	20.9
State-of-the-art [5]	864	21

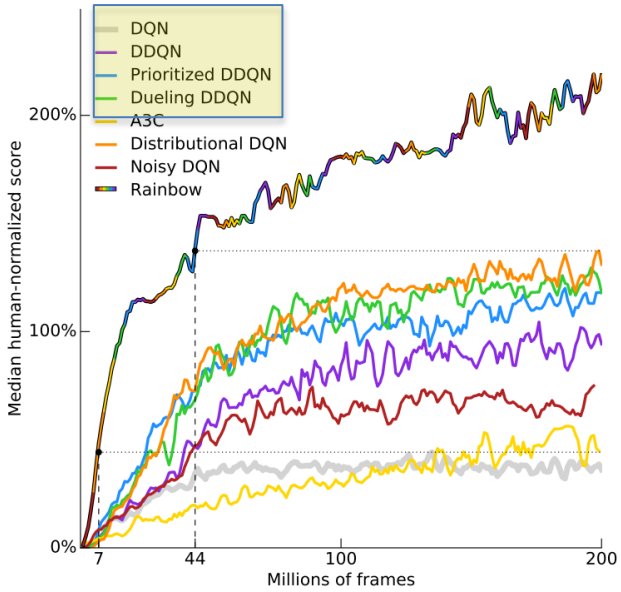
Discussion (3 of 3)

- Open AI transition
  - DQN (old) less support
- Professional implementations
  - CleanRL
  - Stable Baselines
  - Ray RL Lib
  - Tianshou
  - Dopamine

Model	Breakout	Pong
DQN	112.4 +/- 57.6	20.8 +/- 0.5
DOUBLE	96.5 +/- 43.5	20.8 +/- 0.5
PER	43.5 +/- 13.4	20.8 +/- 0.5
DUELING w/ PER	**207.3 +/- 124.0**	**20.9 +/- 0.4**
Random [1]	1.2	- 20.4
Human [1]	31	- 3
DQN [1]	168	20
Double DQN [2]	**375.0**	**21.0**
Double DQN w/ PER [3]	371.6	18.9
Dueling DDQN w/ PER [4]	366.0	20.9
State-of-the-art [5]	864	21

Future work

- More DQN variations
  - Rainbow
- Hyperparameter tuning
  - Optuna
- Random Seeds
  - Validation
- t-SNE



Hessel, M., Modayil, J., Hasselt, H.V., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M.G., & Silver, D. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning. AAAI Conference on Artificial Intelligence. doi.org/10.48550/arXiv.1710.02298

Future work

- Agent57
  - All 57 Atari games
- Generalized Data Distribution Iteration
  - Current State-of-the-art

Badia, Adrià & Piot, Bilal & Kapturowski, Steven & Sprechmann, Pablo & Vitvitskyi, Alex & Guo, Daniel & Blundell, Charles. (2020). Agent57: Outperforming the Atari Human Benchmark. doi.org/10.48550/arXiv.2003.13350

Fan, J., & Xiao, C. (2022). Generalized Data Distribution Iteration. International Conference on Machine Learning. doi.org/10.48550/arXiv.2206.03192

Deep Reinforcement Learning:

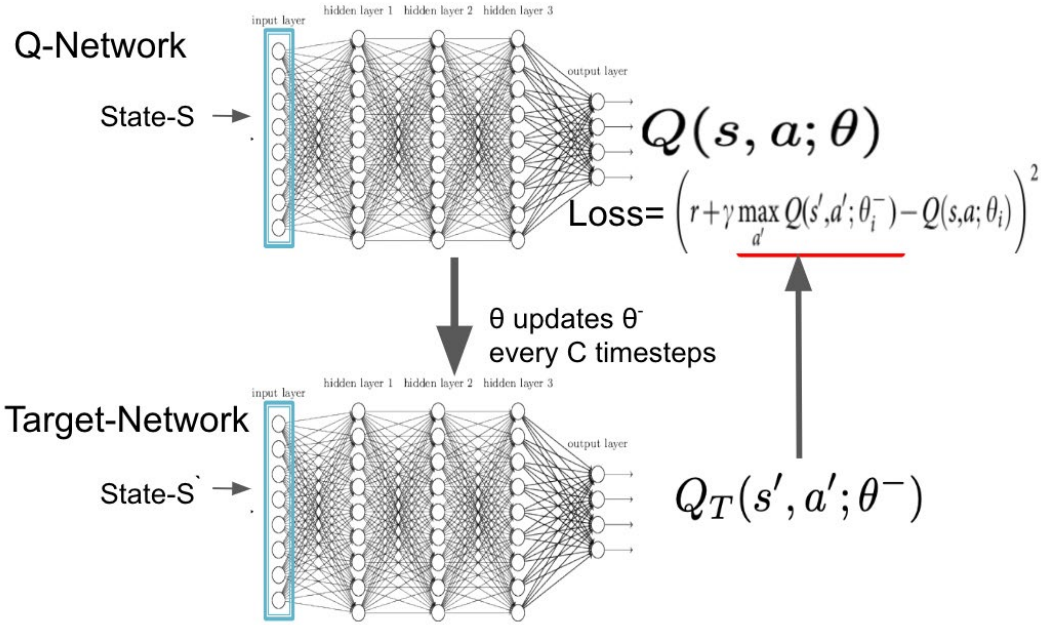
DQN, Double DQN, Dueling DQN, & Prioritize Experience Replay

James Chapman
CIS 730 Artificial Intelligence– Term Project
Kansas State University
jachapman@ksu.edu

Project Repository & Code
https://github.com/JamesChapmanNV/Deep_Reinforcement_Learning

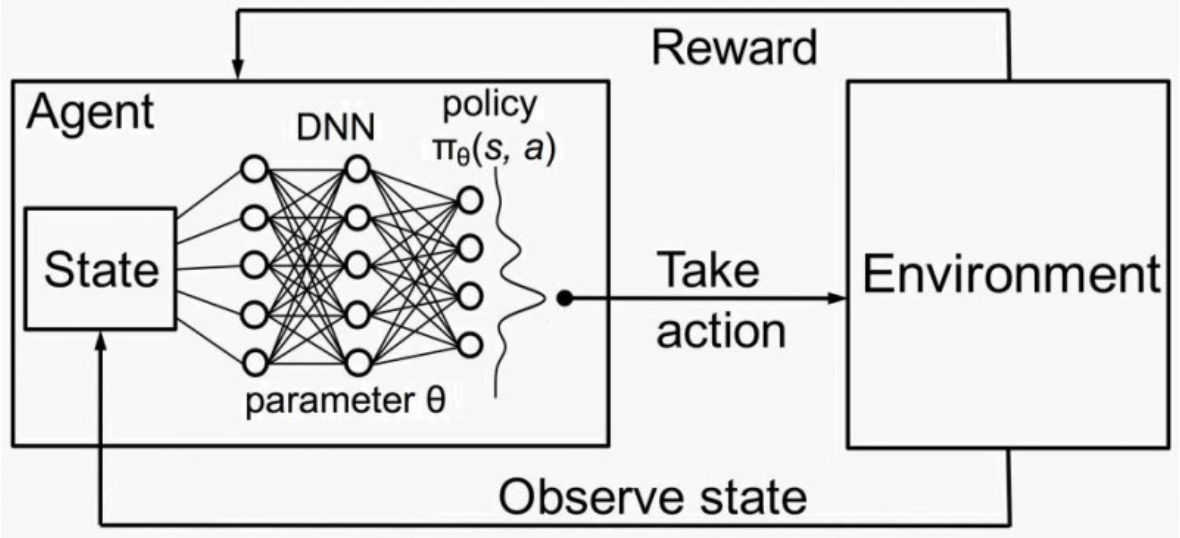
Project Repository & Code

https://github.com/JamesChapmanNV/Deep_Reinforcement_Learning



Project Repository & Code

https://github.com/JamesChapmanNV/Deep_Reinforcement_Learning



Initialization

```
base_env = gym.make(ENV_NAME)
env = PongWrappers(base_env)

net = ConvNet(env.observation_space.shape, env.action_space.n,is_dueling).to(device)
target_net = ConvNet(env.observation_space.shape, env.action_space.n,is_dueling).to(device)
optimizer = optim.Adam(net.parameters(), lr=learning_rate)

if is_prioritized_replay:
    buffer = PrioritizedExperienceReplay(env.observation_space.shape, replay_size, batch_size, alpha)
else:
    buffer = ExperienceReplay(replay_size)
agent = Agent(env, buffer, action_repeat)

frame_idx = 0
game_frame_idx = 0
epsilon = eps_start

total_rewards = []
mean_rewards = []
std_rewards = []
frame_idxs = []
epsilons = []
```

```
Creating environment------------
Box(0, 255, (210, 160, 3), uint8)
Box(0, 255, (84, 84, 1), uint8)
Box(0.0, 1.0, (1, 84, 84), float32)
Box(0.0, 1.0, (4, 84, 84), float32)
Box(0.0, 1.0, (4, 84, 84), float32)
-------------------------------
```

Experiment Design

- Farmara's Gymnasium
  -
```
        # Save to Buffer
        if is_prioritized_replay:
            ######### Prioritized Experience Replay ########
            self.exp_buffer.add(self.state, action, reward, new_state, terminated)
            ################################################
        else:
            ######### Simple Experience Replay #############
            exp = Experience(self.state, action, reward, terminated, new_state)
            self.exp_buffer.append(exp)
            ################################################

        # This has not Triggered yet
        #if truncated:
            #print(terminated, truncated, self.env.ale.lives(), info)

        self.state = new_state
        done_reward = None
        if is_done:
            done_reward = self.total_reward
            self._reset()
        return done_reward
```