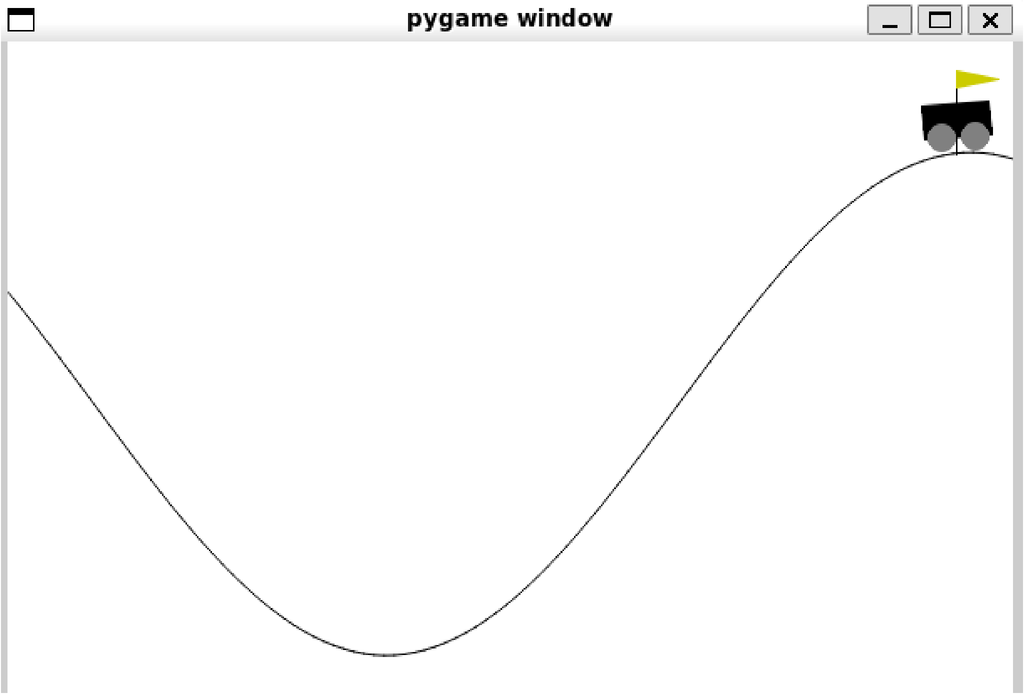
UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Lecture 9
Value Function Approximation
DQN and DDQN

jmanero@faculty.ie.edu

MBD-EN2024ELECTIVOS-MBDMCSBT_37E89_467614

Shaping Rewards – MOUNTAIN-CART



This environment is part of the Classic Control environments which contains general information about the environment.

Action Space	`Discrete(3)`
Observation Space	`Box([-1.2 -0.07], [0.6 0.07], (2,), float32)`
import	`gymnasium.make("MountainCar-v0")`

2 Problems discretization and Shaping rewards

Discretization

- How many bins are ok?
- Hint: use this discretization strategy for the CARTPOLE Assignment !

Shaped Rewards

- -1 each timestep
- This reward does not allow good learning
- There is truncation and termination (careful)
- Can we define a shaped reward?

Objective

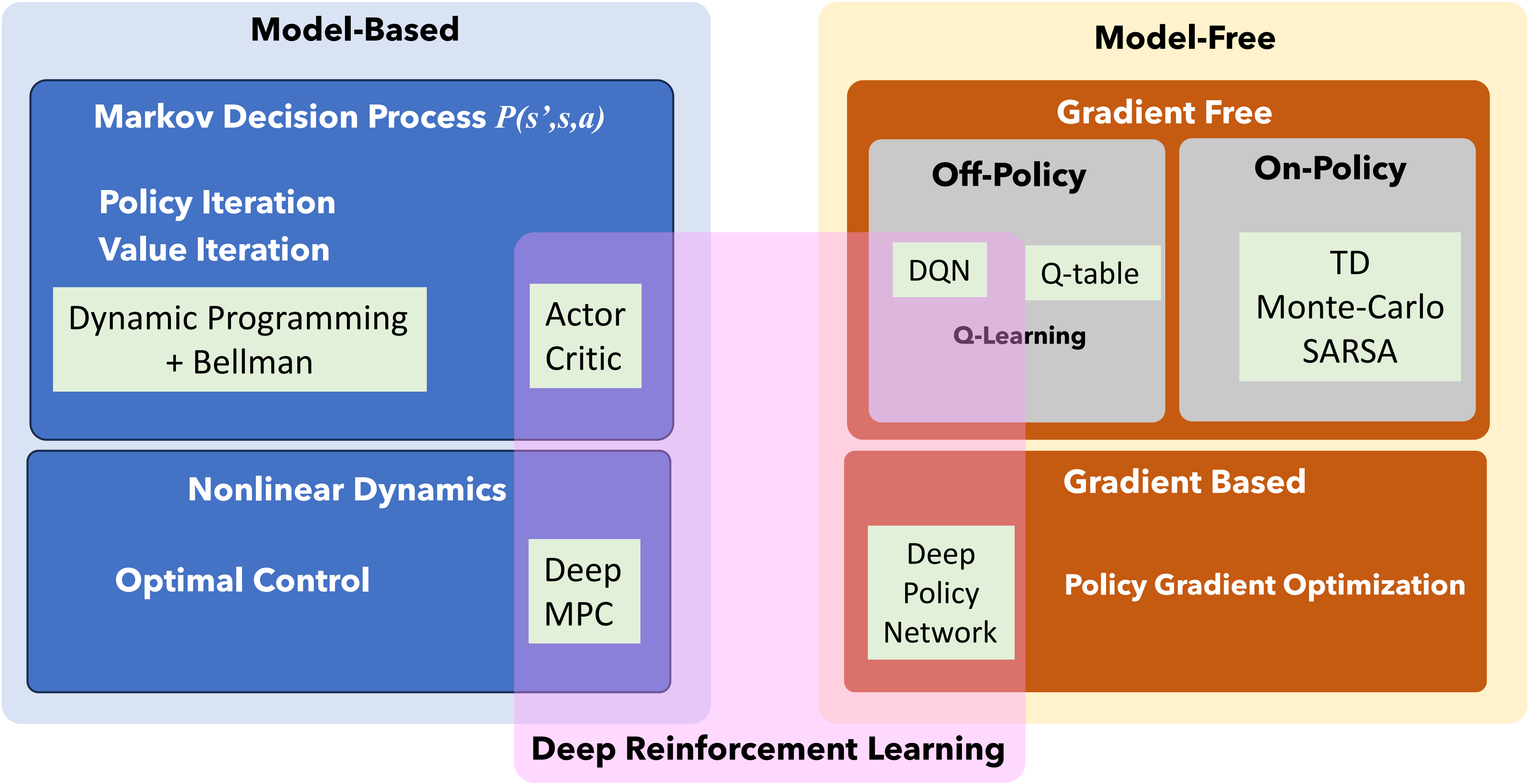
- **By changing BINS and Shaping reward obtain the BEST POSIBLE LEARNING (FAST-SHORT)**
- Use ChatGPT, internet, your ideas, whatever
- Don’t modify the code or the method. Just focus on shaped rewards

https://gymnasium.farama.org/environments/classic_control/mountain_car/
https://github.com/castorgit/RL_course/blob/main/021_Q_learning_MOUNTAIN_CAR.ipynb

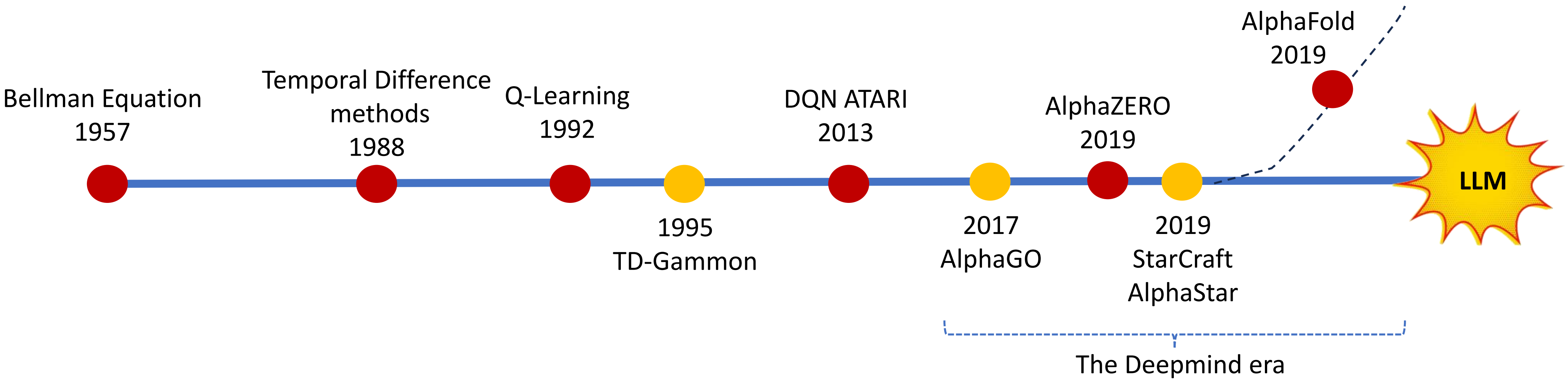
Challenge
For the Assignment CARTPOLE Discretization

```python
#Support Funtions



# Function to create bins
def create_bins(interval, num):
    return np.linspace(interval[0], interval[1], num + 1)

# Updated intervals and bin sizes for discretization
intervals = [(-2.4, 2.4), (-3.0, 3.0), (-0.5, 0.5), (-2.0, 2.0)]
nbins = [12, 12, 24, 24]  # Increased bins for finer state representation
bins = [create_bins(intervals[i], nbins[i]) for i in range(4)]

# Function to discretize state variables into bins
def discretize_bins(x):
    return tuple(np.clip(np.digitize(x[i], bins[i]) - 1, 0, nbins[i] - 1) for i in range(4))
```

Classification of RL Methods

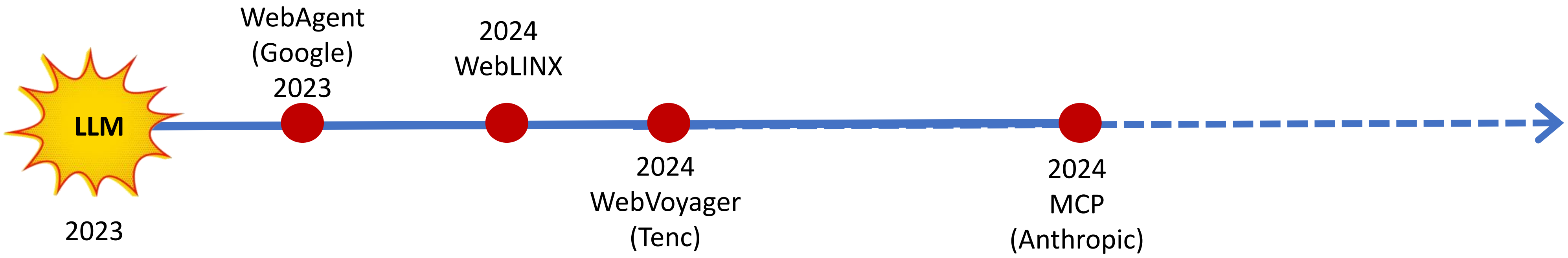


A Brief story of Agents
From Bellman to LLM



(AlphaStar 2019) Grandmaster level in StarCraft II using multi-agent reinforcement Learning
(Silver 2017) Mastering the game of GO without human knowledge
(David Silver 2013) Playing Atari with Deep Reinforcement Learning
(Tesauro 1995) Temporal Difference Learning and TD-Games
(Watkins 1992) Q-learning
(Bellman 1957) Dynamic Programming

After the LLM



(Anthropic 2024) Introducing the MCP
(He 2024) WebVoyager: Building an End-to-End Web Agent with Large Multimodal Models
(Lú 2024) WebLINX: Real-World Website Navigation with Multi-Turn Dialogue
(Gur 2023) A Real-World WebAgent with Planning, Long Context Understanding, and Program Synthesis (DeepMind)

A Brief story of Agents
Frameworks

Langchain
2022
Chaining tasks
Memory management
Context Retention

2023
AUTOGPT
Automates Tasks
Uses refining feedback loop
More emphasis on autonomous decision making

AutoGen
2023
Multiagent
Integrates Human in the loop

2023
Crew.ai
Versatile tools
Allows some human oversight
Collaborative

Langraph
2024
Collaborative agents
Graph Based
Stateful

2024
LlamaIndex
Event Driven
State management and cycles
Debugging tools

TapeAgents
2024
Allows Complex agent
Prompt tunning
Distillation

Contents

The ATARI Games Paper

The original Paper

https://www.youtube.com/watch?v=V1eYniJ0Rnk

arXiv:1312.5602v1 [cs.LG] 19 Dec 2013

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou
Daan Wierstra Martin Riedmiller
DeepMind Technologies
{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

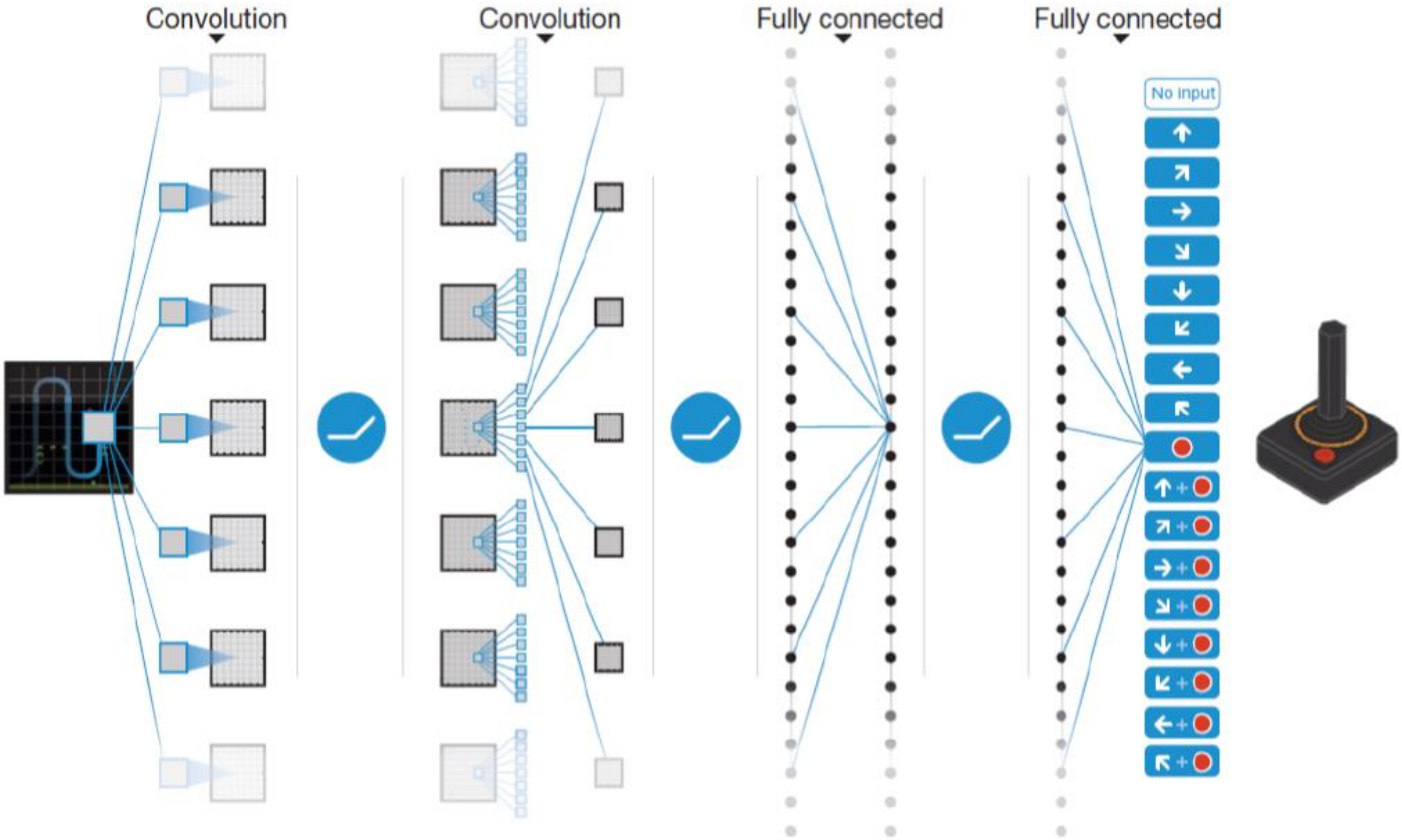
However reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand-labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution.

This paper demonstrates that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL environments. The network is trained with a variant of the Q-learning [26] algorithm, with stochastic gradient descent to update the weights. To alleviate the problems of correlated data and non-stationary distributions, we use

1

https://arxiv.org/pdf/1312.5602

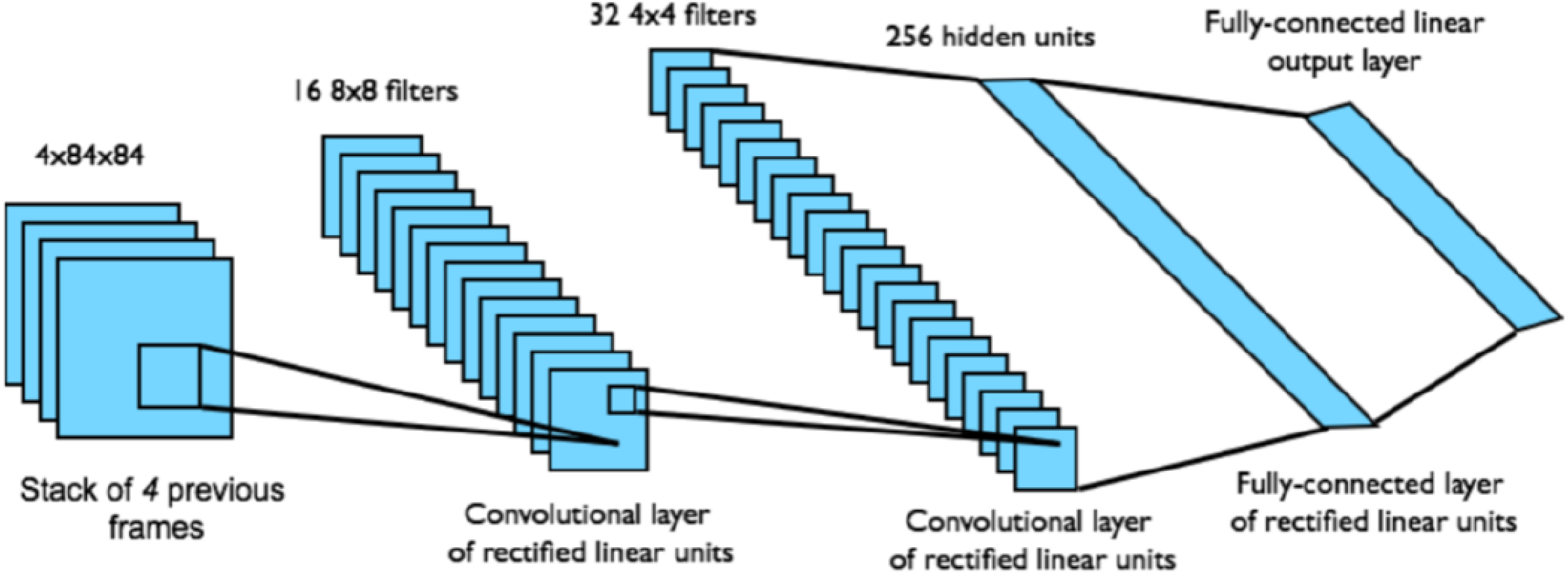
DQN
Playing atari Games



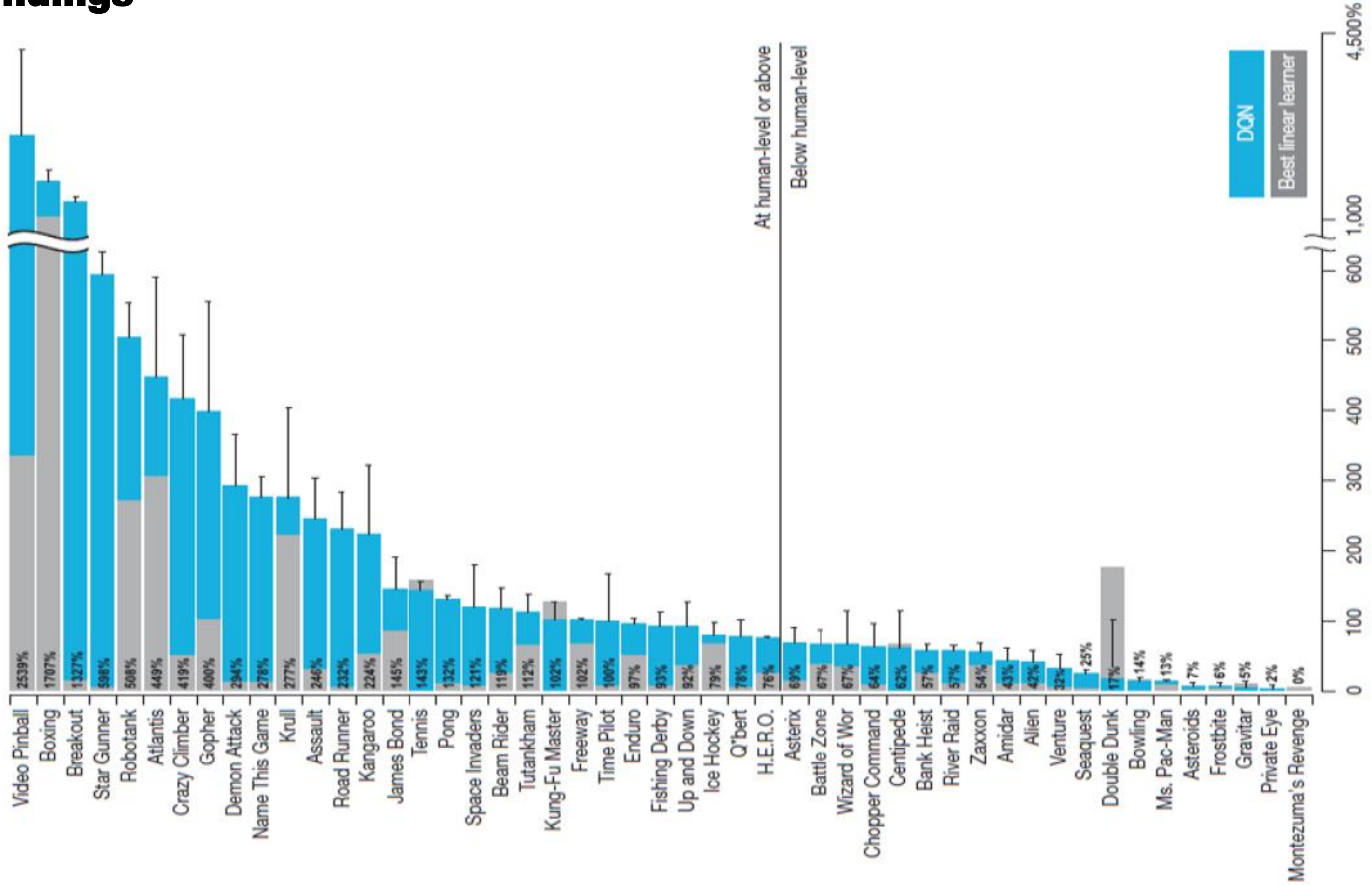
Atari Games

- End-to-end learning of values Q(s; a) from pixels:
  - **State:** Input state s is stack of raw pixels from last 4 frames
  - **Actions:** Output is $Q(s, a)$ value for each of 18 joystick/button positions
  - **Reward:** Reward is direct change in score for that step
- Network architecture and hyper-parameters **fixed across all games**, No tuning!
- Clipping reward -1,0,1 to avoid problem of different magnitudes of score in each game

Atari Games


32 4x4 filters
256 hidden units
Fully-connected linear output layer
16 8x8 filters
4x84x84
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Paper Findings



@faculty.ie.edu

Neural Networks as Function Approximators

What to do when the problem is too big

- What happens when the state space is very large?
  - Backgammon: 10^{20}
  - Go: 10^{170}
  - Helicopter: continuous state space
  - Autonomous vehicle: continuous state space
- How can we scale-up the model-free methods ?

The solution is function approximation

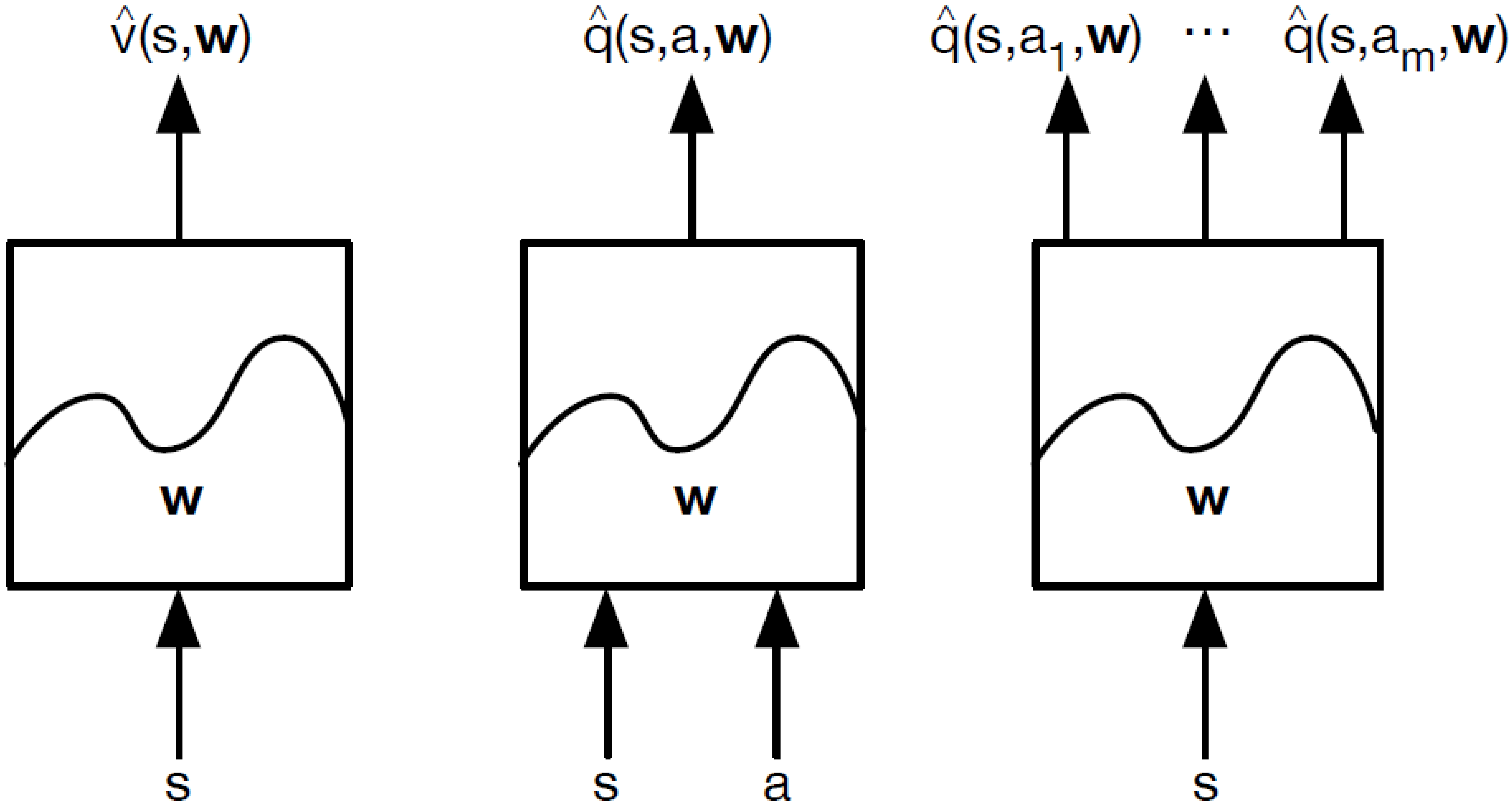
- So far we have represented value function by a *lookup table*
  - Every state s has an entry $V(s)$
  - Or every state-action pair s, a has an entry $Q(s, a)$
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with *function approximation*

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

$$\text{or } \hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

  - *Generalise* from seen states to unseen states
  - *Update* parameter $\mathbf{w}$ using MC or TD learning

Types of Function approximation



Any regressor can be a function approximator

There are many function approximators, e.g.

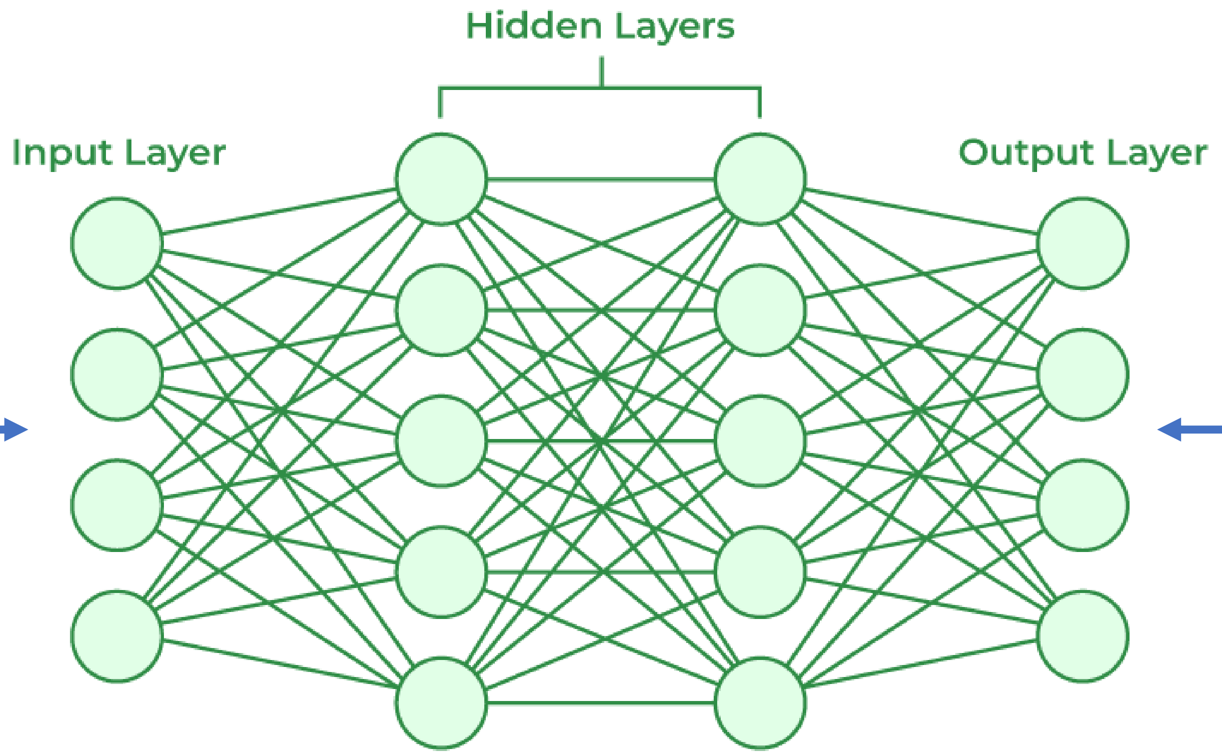
- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Neural Networks as Function approximators

How to approximate the function / Q-Table

Observation

env.observation_space.shape[0]



Action

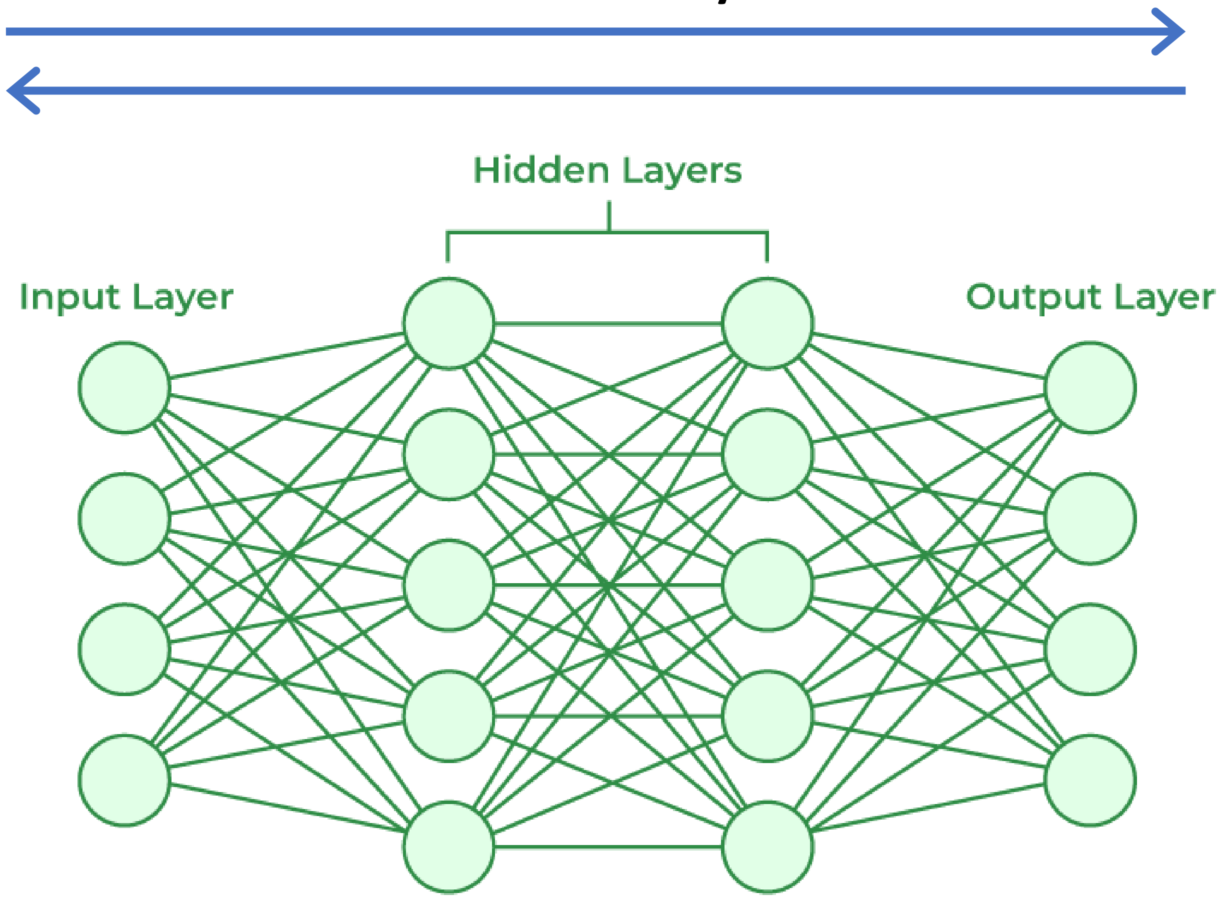
env.action_space.n

Architecture of the network (KERAS)

```
inputs = Input(shape=(state_size,))
hidden1 = Dense(24, activation="relu")(inputs)
hidden2 = Dense(24, activation="relu")(hidden1)
outputs = Dense(action_size,activation="linear")(hidden2)
model   = Model(inputs=inputs, outputs=outputs)

model.compile(optimizer=Adam(learning_rate=learning_rate), loss="mse")


model.fit
model.predict
```

FIT integrating forward and backward pass in one

Model Fit – Fully Automatized



KERAS – integrating Forward and Backward Pass in MODEL.FIT

model.fit: A high-level API for training models.

How It Works:

- You define a model, specify the loss, optimizer, and metrics, then call the FIT method
- Keras handles the entire training loop, including data iteration, forward pass, backward pass, and optimization.

Advantages:

- **Ease of use**: Minimal code required to train a model.
- **Built-in features**: Supports callbacks (e.g., early stopping, model checkpoints), metrics, and data preprocessing.
- **Optimized implementation**: TensorFlow has highly optimized training loops, especially for standard use cases.

Disadvantages:

- **Less control**: Harder to modify or customize the training loop for non-standard tasks.
- **Debugging**: Limited access to intermediate values unless additional effort is made.

KERAS – Disintegrating Forward and Backward Pass using MODEL.TAPE

```
class ANN_model(Model):
       def __init__(self, hidden_size,
num_classes):
       super(ANN_model, self).__init__()
              self.dense1 = Dense(hidden_size)
              self.relu = ReLU()
              self.dense2 = Dense(num_classes)
              self.softmax = Softmax()
       def call(self, inputs):
              x = self.dense1(inputs)
              x = self.relu(x)
              x = self.dense2(x)
              return self.softmax(x)
```

```
model = ANN_model(hidden_size=hidden_size, num_classes=num_classes)
# Loss function and optimizer
loss_fn = SparseCategoricalCrossentropy(from_logits=True)
optimizer = Adam()
```

KERAS – Disintegrating Forward and Backward Pass using MODEL.TAPE

```
 # Iterate over the batches of the dataset.

for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):
      with tf.GradientTape() as tape:
            logits = model(x_batch_train, training=True)
            loss_value = loss_fn(y_batch_train, logits)
      grads = tape.gradient(loss_value, model.trainable_weights)
      optimizer.apply_gradients(zip(grads, model.trainable_weights))

# Update training metric.

      train_acc_metric.update_state(y_batch_train, logits)
```

KERAS – integrating Forward and Backward Pass in MODEL.FIT

tf.GradientTape: A lower-level API for manually implementing backpropagation.

How It Works:

- You define a forward pass of the network within the GradientTape context.
- TensorFlow/KERAS automatically records operations to compute gradients with respect to trainable variables.
- You manually compute the loss and apply gradients using an optimizer.

Advantages:

- **Fine-grained control**: You can customize every aspect of the training process, including gradient computation, loss scaling, and optimization logic.
- **Flexibility**: Useful for research or tasks requiring non-standard training loops, such as multi-task learning, adversarial training, or reinforcement learning.
- **Debugging and Experimentation**: Easier to inspect intermediate values, gradients, and loss calculations.

Disadvantages:

- **Complexity**: Requires more lines of code and careful handling of various steps (e.g., resetting gradients).

On every batch, in backpropagation

1. **Computing the Gradients based on the loss function (Backpropagation)**

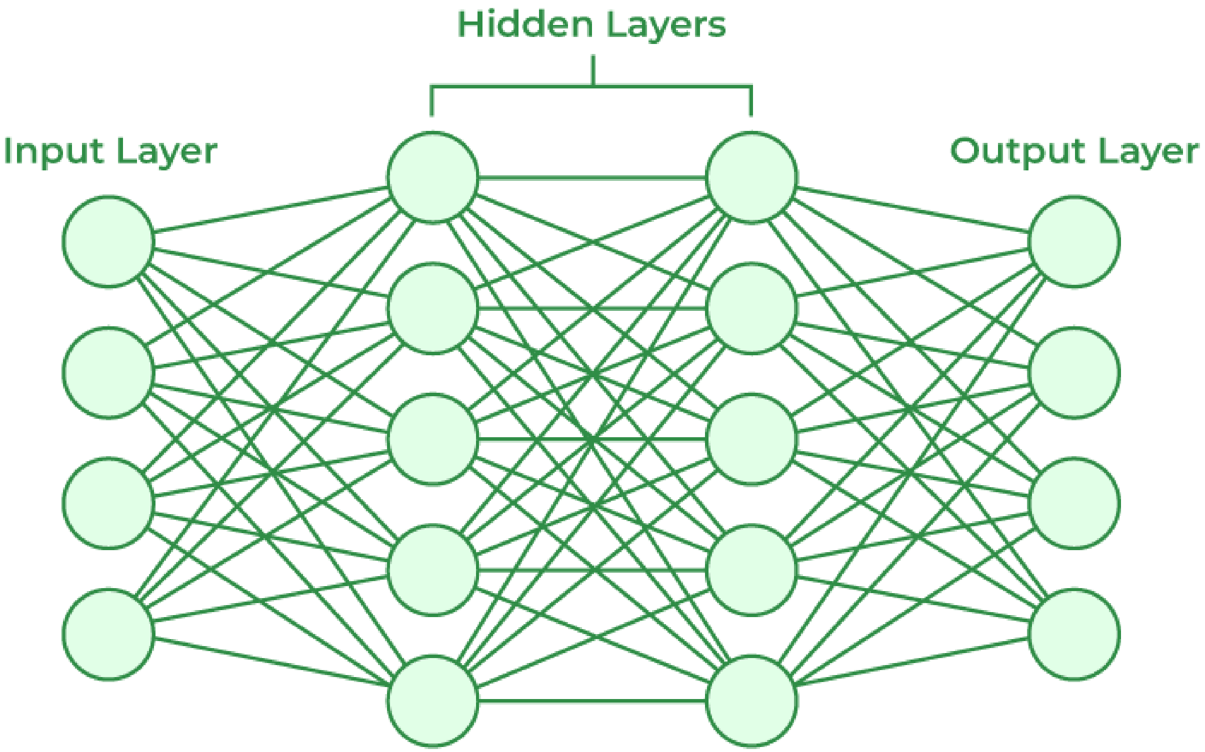
2. **The optimizer updates the weights (Gradient Descent)**

$$\text{weights} \leftarrow \text{weights} - \text{learning\_rate} \times \text{gradient}$$

FIT integrating forward and backward pass in one

Model Tape – Forward

Model Tape – Manual calculation of gradients



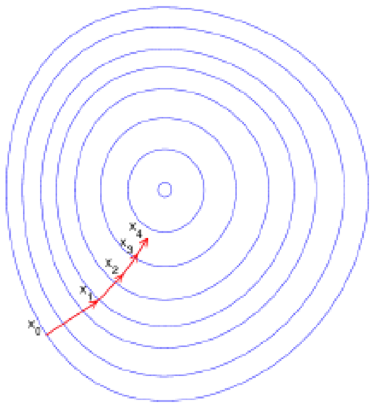
Deep Q-Networks (DQN)

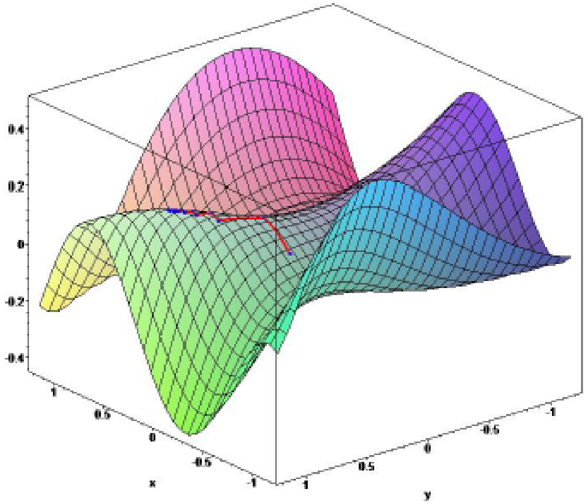
Incremental methods – Gradient Descent

Finding the minima using Gradient Descent

- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$
- Define the *gradient* of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$





- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of -ve gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter

Batch Methods

- Gradient descent is simple and appealing
- But it is *not* sample efficient
- Batch methods seek to find the best fitting value function
- Given the agent's experience ("training data")

Experience Replay in Deep Q-Networks (DQN)

DQN uses experience replay and fixed Q-targets

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters w_i^- [No Title]
- Optimise MSE between Q-network and Q-learning targets

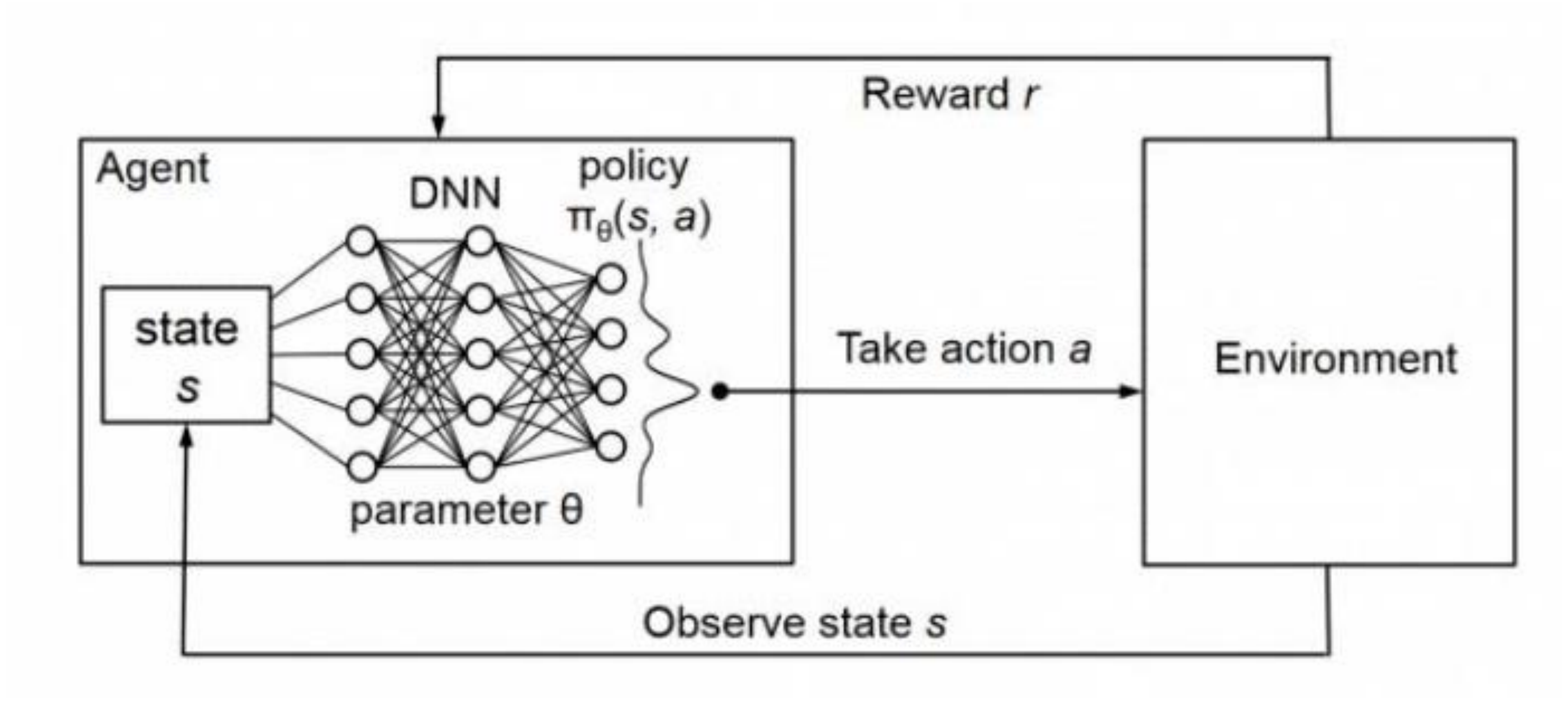
$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

- Using variant of stochastic gradient descent

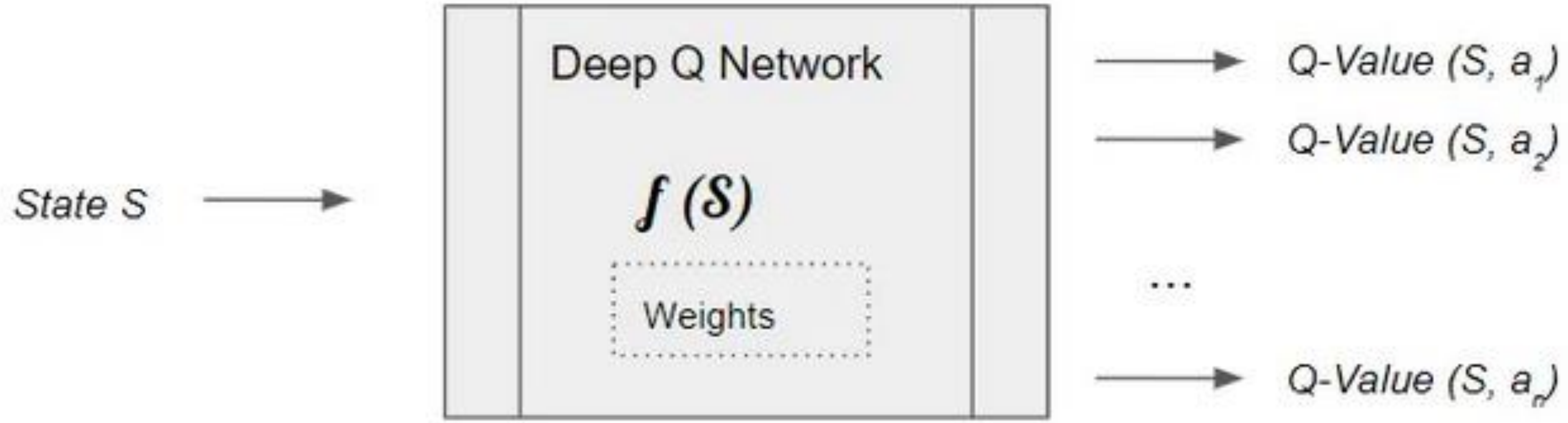
Q-Learning with Neural Networks

- Q-learning converges to optimal $Q^*(s, a)$ using tabular representation
- In value function approximation Q-learning minimizes MSE loss by stochastic gradient descent using a target Q estimate instead of true Q
- But Q-learning with VFA can diverge
- Two of the issues causing problems:
    - Correlations between samples
    - Non-stationary targets
- Deep Q-learning (DQN) addresses these challenges by using
    - Experience replay
    - Fixed Q-targets

Function approximation with a Neural Network



The general idea



Experience Replay

- To help remove correlations, store dataset (called a **replay buffer**) $\mathcal{D}$ from prior experience

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\ldots$
$s_t, a_t, r_{t+1}, s_{t+1}$

$\rightarrow \quad s, a, r, s'$

- To perform experience replay, repeat the following:
  - $(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
  - Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w})$
  - Use stochastic gradient descent to update the network weights

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$

Experience Replay

DQN uses experience replay and fixed Q-targets

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

- Using variant of stochastic gradient descent

Wrap up

- DQN uses experience replay and fixed Q-targets
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters $\boldsymbol{w}^-$
- Optimizes MSE between Q-network and Q-learning targets
- Uses stochastic gradient descent

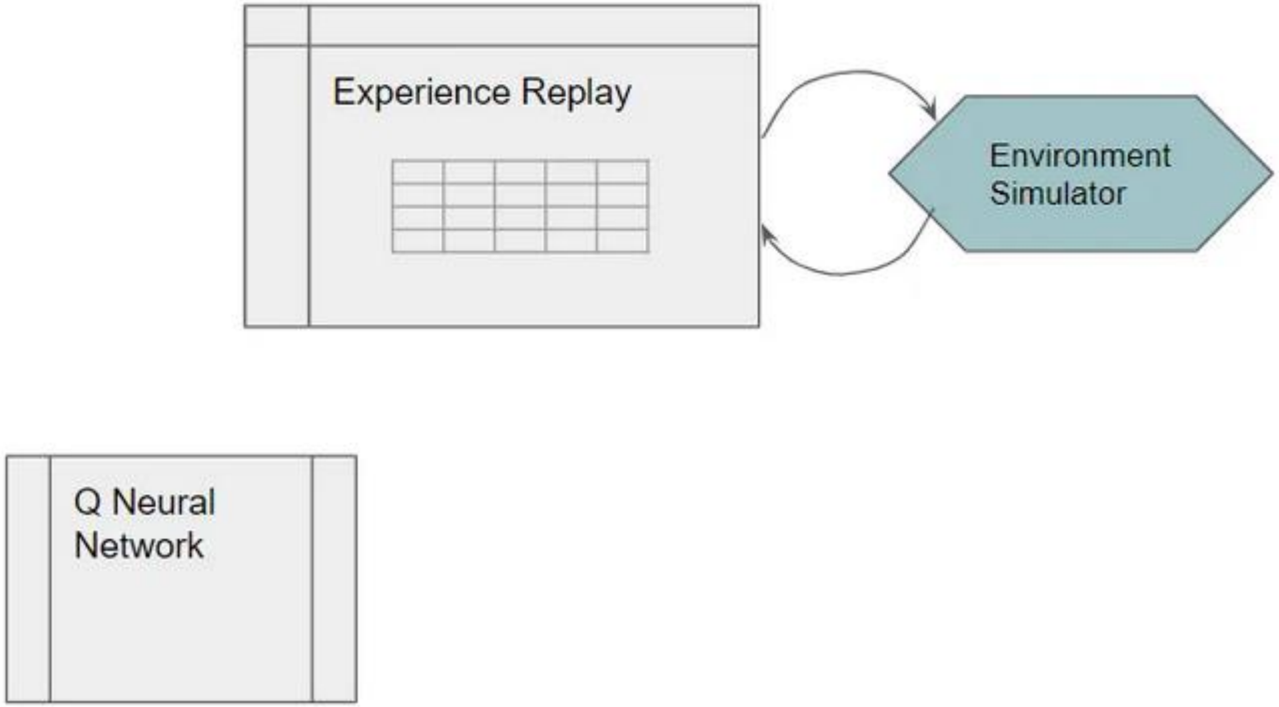
DQN pseudocode

DQN - Deep Q Network (Mnih, et al. 2015)

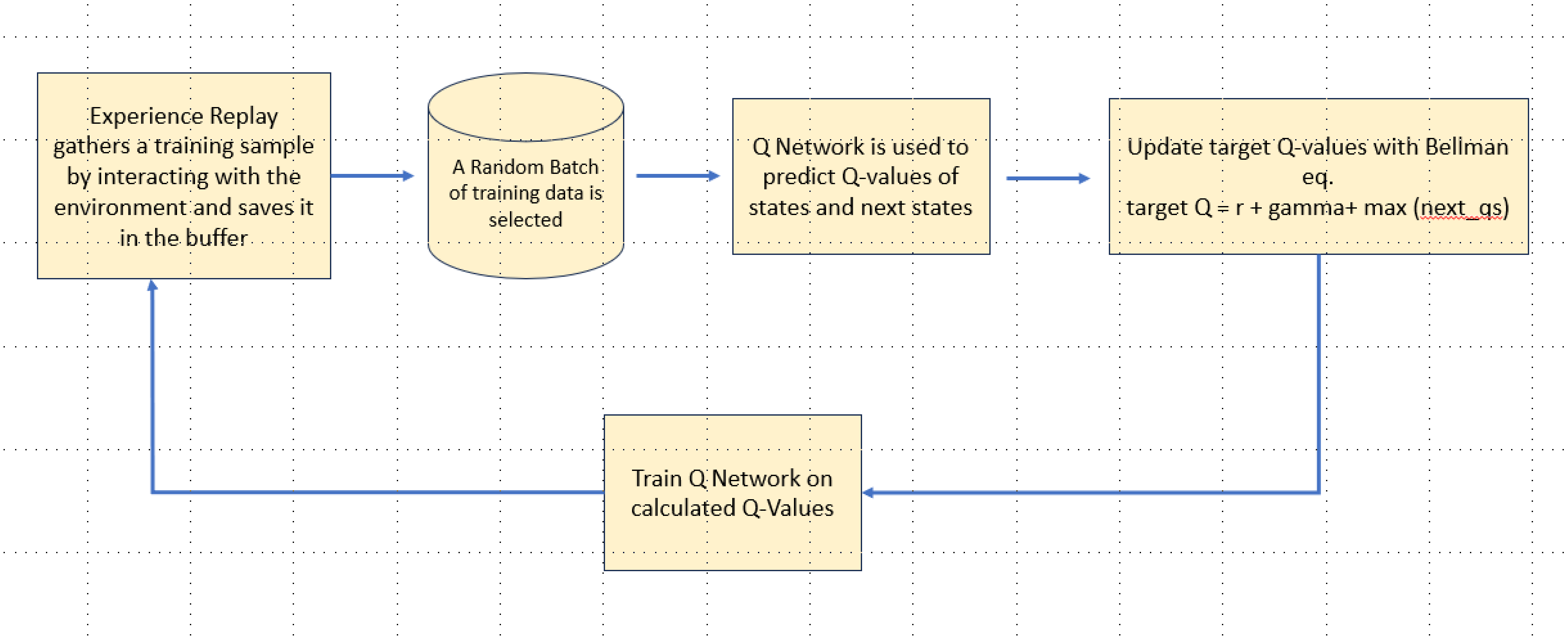
Initialize replay memory R with capacity N
Initialize Q-Network with random weights
Initialize target network Q_target with weights θ target $= \theta$
Set learning_rate α, γ, ϵ
For each episode:
    Initialize s
    While s is not terminal:
        Sample action A with ϵ-greedy policy
        Take action A observe R and next state S'
        Store transition (S, A, R, S', done) in replay memory D
        If replay memory D has sufficient samples:
            Sample a minibatch of transitions from D
            Compute target:
            If $done_j$:
                $y_j = r_j$
            else:
                $y_j = r_j + y * max(Q : target(s_j^{'}, a^{'}, \theta_{target}))$
            Perform Gradient descent step on loss:
                Loss $= y_j - Q(s_j, a_j, \theta))^2$
        Every C steps, update target network
            $\theta_{target} - \theta$
        Update state $s = s^{'}$

Note there are several hyperparameters and algorithm choices. One needs to choose the neural network architecture, the learning rate, and how often to update the target network. Often a fixed size replay buffer is used for experience replay, which introduces a parameter to control the size, and the need to decide how to populate it.

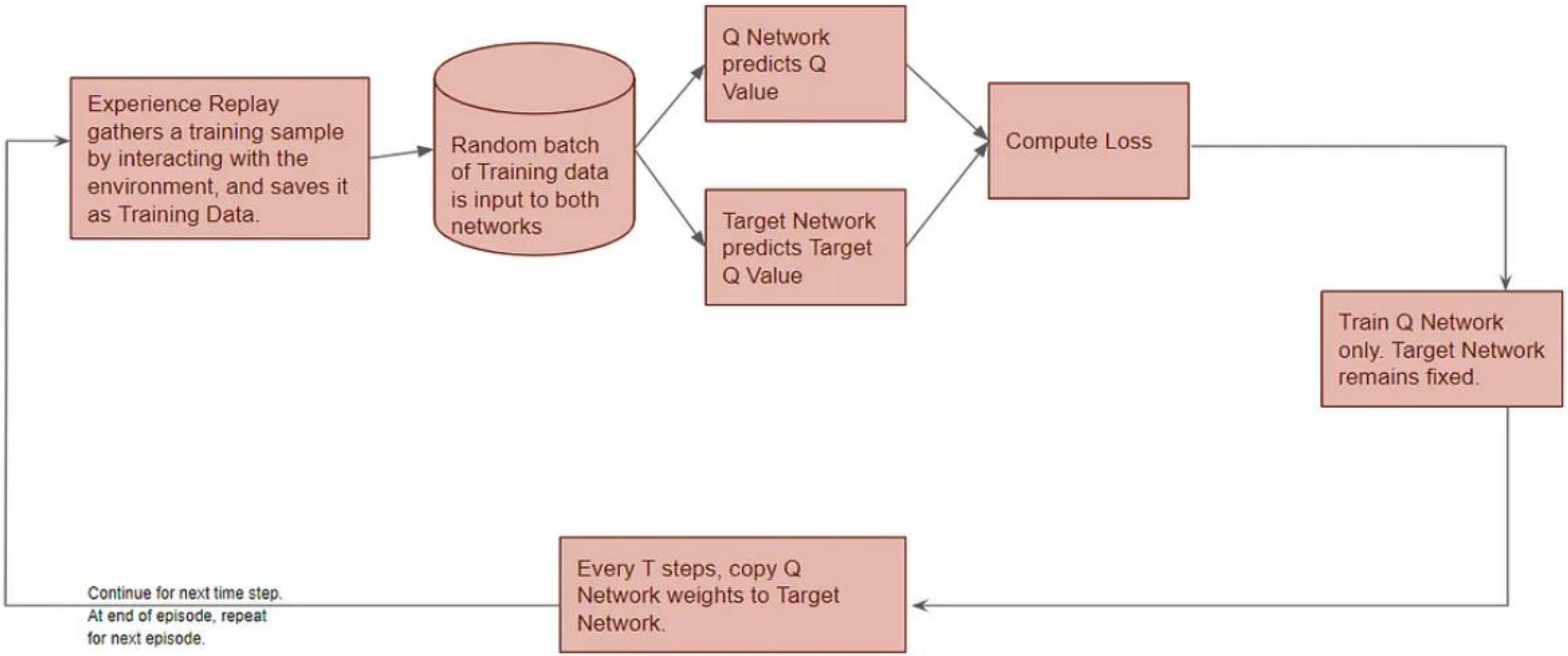
The Components



General Workflow

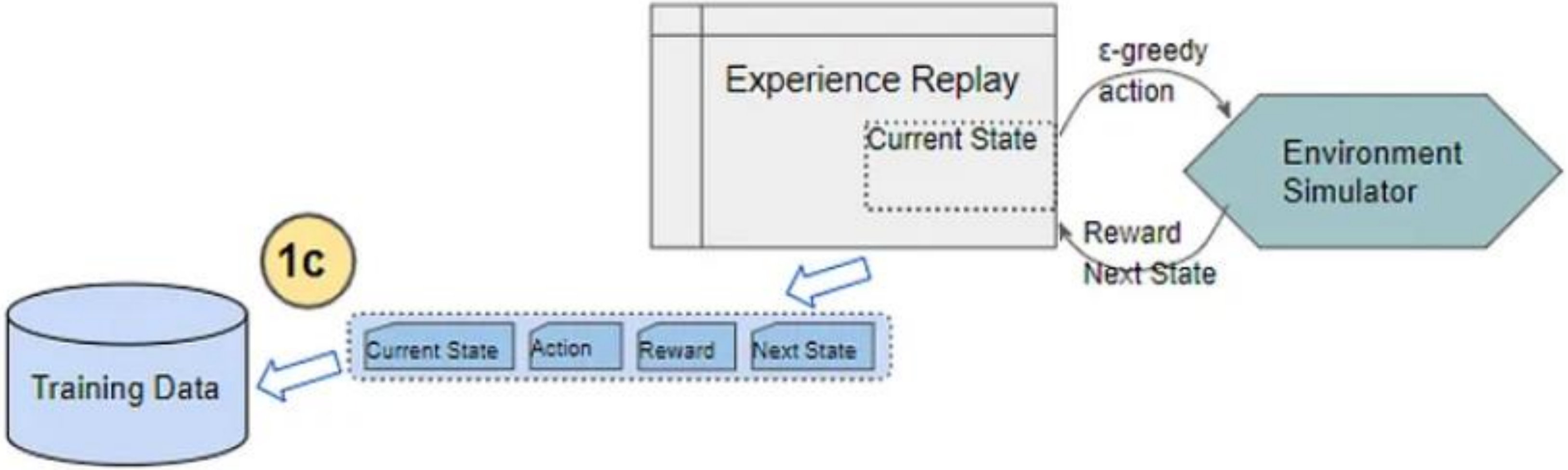


Using 2 networks instead of 1



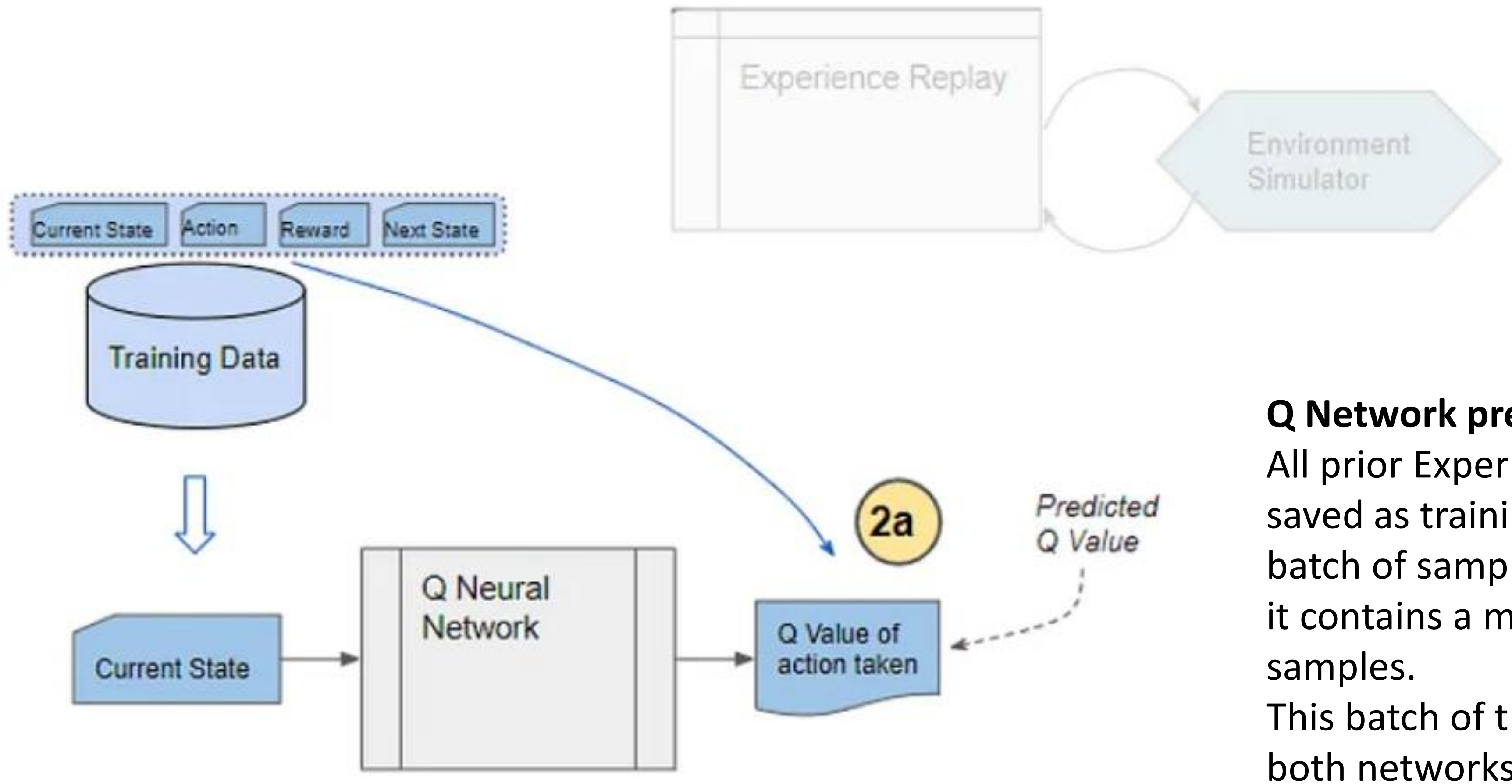
1st step select an ε-greedy action

Experience Replay selects an ε-greedy action from the current state, executes it in the environment, and gets back a reward and the next state.



It saves this observation as a sample of training data.

Q Network predicts Q-value

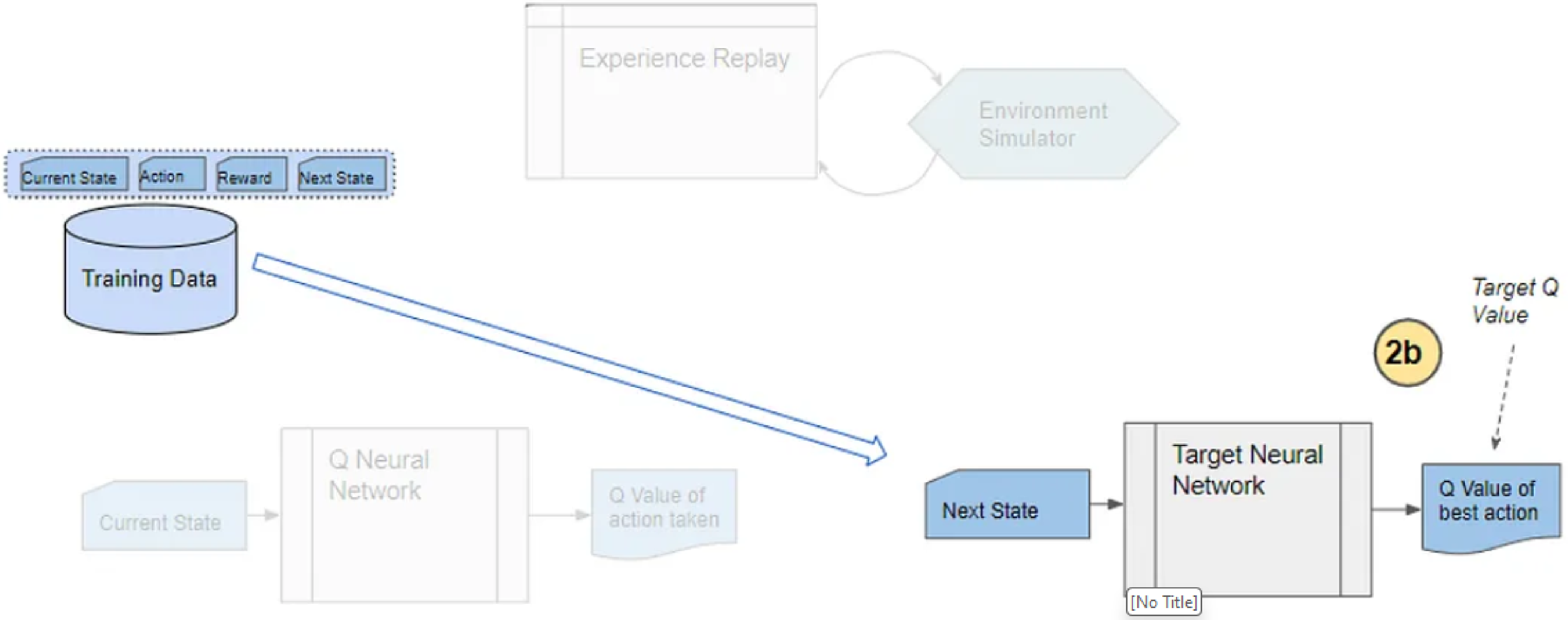


Q Network predicts Q-value

All prior Experience Replay observations are saved as training data. We now take a random batch of samples from this training data, so that it contains a mix of older and more recent samples.

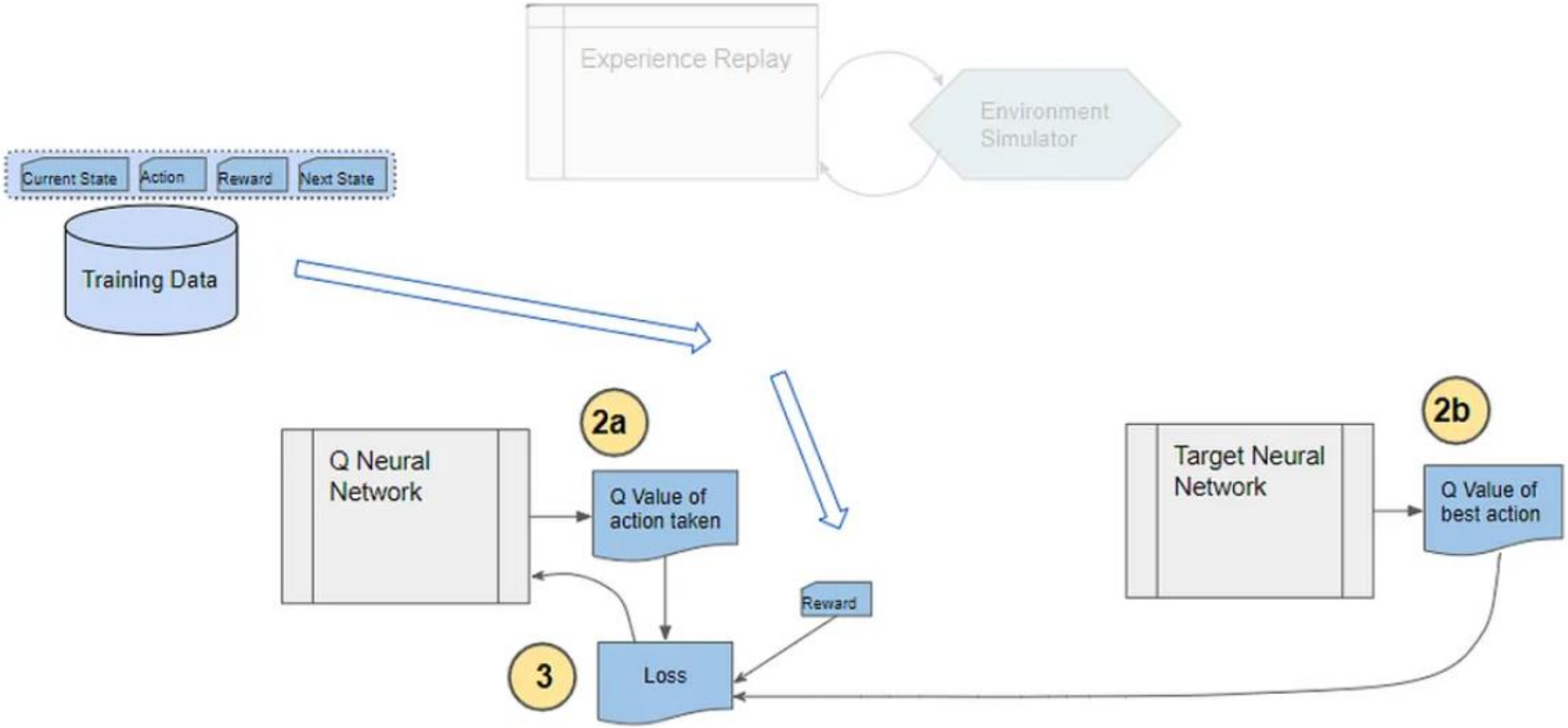
This batch of training data is then inputted to both networks. The Q network takes the current state and action from each data sample and predicts the Q value for that particular action. This is the ‘Predicted Q Value’.

Target Network predicts Target Q-value



Target Network predicts Target Q-value
The Target network takes the next state from each data sample and predicts the best Q value out of all actions that can be taken from that state. This is the 'Target Q Value'.

Target Network predicts Target Q-value



Compute Loss and Train Q Network

The Predicted Q Value, Target Q Value, and the observed reward from the data sample is used to compute the Loss to train the Q Network. The Target Network is not trained.

Summary of DQN Algorithm

- DQN uses experience replay and fixed Q-targets
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters $\boldsymbol{w}^-$
- Optimizes MSE between Q-network and Q-learning targets
- Uses stochastic gradient descent

Conclusions

- DQN is more reliable on some tasks than others. Test your implementation on reliable tasks like Pong and Breakout: if it does achieve good scores, something is wrong.
- Large replay buffers improve robustness of DQN, and memory efficiency is key.
- SGD can be slow .. rely on RMSprop (or any new optimizer)
- Convolutional models are more ecient then MLPs
- DQN uses action repeat set to 4 (because fps too high - speeds training time)
- DQN receives 4 frames of the game at a time (grayscale)
- ϵ is anealled from 1 to .1

DDQN

What is the difference between DDQN and DQN

DQN

```
# Uses TARGET network for both selection AND evaluation
target_q_values = target_model.predict(next_states) # TARGET network
max_target_q = np.max(target_q_values, axis=1) # Find max from TARGET
targets = rewards + gamma * max_target_q * (1 - dones)
```

DDQN

```
# Uses MAIN network for selection, TARGET network for evaluation
next_q_values_main = main_model.predict(next_states)       # MAIN network (selection)
best_actions = np.argmax(next_q_values_main, axis=1)       # Best action from MAIN

next_q_values_target = target_model.predict(next_states)    # TARGET network (evaluation)
selected_q_values = next_q_values_target[..., best_actions] # Evaluate with TARGET

targets = rewards + gamma * selected_q_values * (1 - dones)
```

The full code

```
def experience_replay_DQN(batch_size, model, epsilon):
    if len(replay_buffer) < batch_size:
        return
    states, actions, rewards, next_states, dones = sample_experiences(batch_size)

    q_values = DQN.predict(states, verbose=0)
    next_qs = target_model.predict(next_states, verbose=0)
    for i in range(batch_size):
        target = rewards[i] + (1 - dones[i]) * gamma * np.max(next_qs[i])
        q_values[i][actions[i]] = target
    model.fit(states, q_values, epochs=1, verbose=0)

def experience_replay_DDQN(batch_size, model, epsilon):
    if len(replay_buffer) < batch_size:
        return
    states, actions, rewards, next_states, dones = sample_experiences(batch_size)

    q_values = DQN.predict(states, verbose=0)

    # DDQN: Use main network for action selection
    next_qs_main = DQN.predict(next_states, verbose=0)        # Main network
    best_actions = np.argmax(next_qs_main, axis=1)            # Select best actions

    # DDQN: Use target network for action evaluation
    next_qs_target = target_model.predict(next_states, verbose=0)  # Target network

    for i in range(batch_size):
        # Evaluate the selected action using target network
        target = rewards[i] + (1 - dones[i]) * gamma * next_qs_target[i][best_actions[i]]
        q_values[i][actions[i]] = target

    model.fit(states, q_values, epochs=1, verbose=0)

```

What is the difference between DDQN and DQN

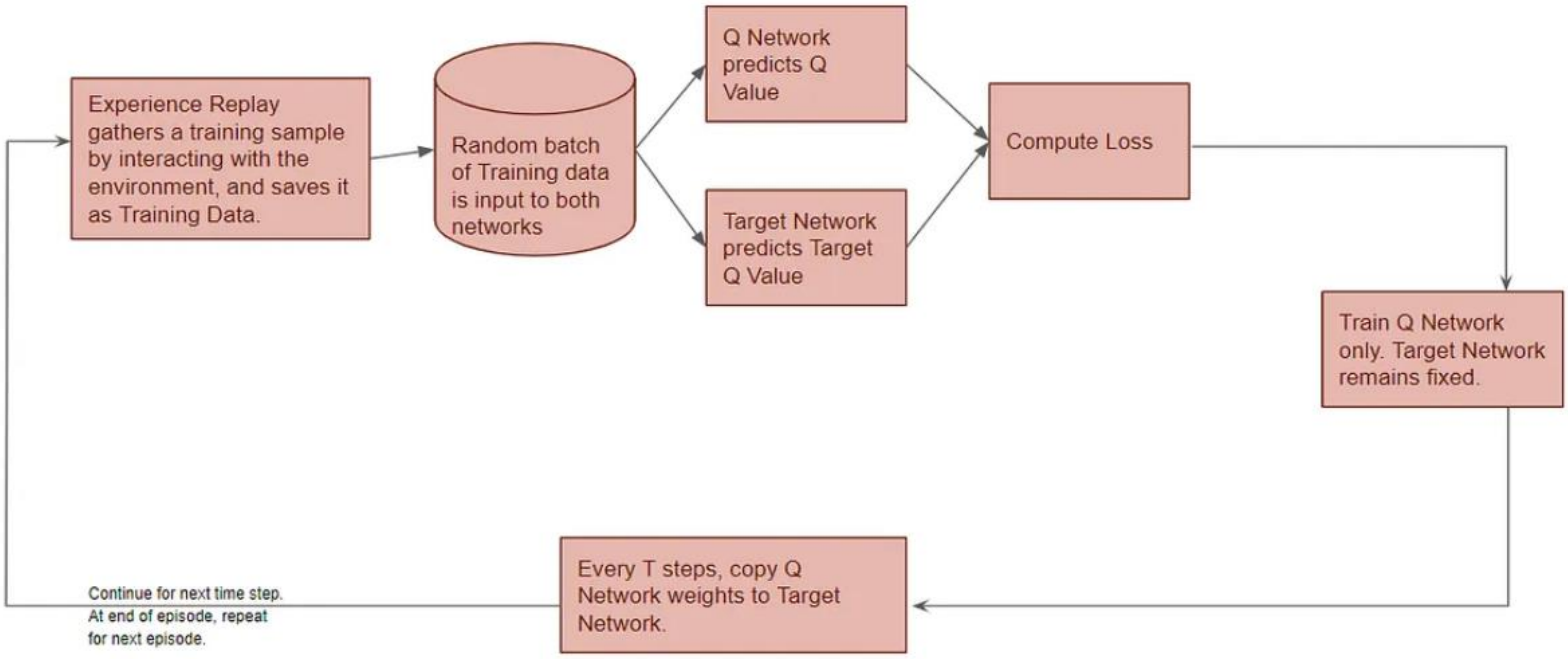
- **DQN**: 2 networks, target network does selection + evaluation
- **DDQN**: 2 networks, main network does selection, target network does evaluation

Both algorithms use the exact same 2-network architecture - they just use the networks differently during the target calculation step.

Double DQN (DDQN)

- **Problem Addressed**: Overestimation bias in Q-values in DQN.
- **Key Innovations**:
  - Separates the action selection and Q-value evaluation using two networks:
    - Online network selects the action.
    - Target network evaluates the Q-value.
- **Result**: More accurate value estimation, improving stability

Double DQN (DDQN)



Merging the weights of the two Networks

Hard update

$$\theta' \leftarrow \theta$$

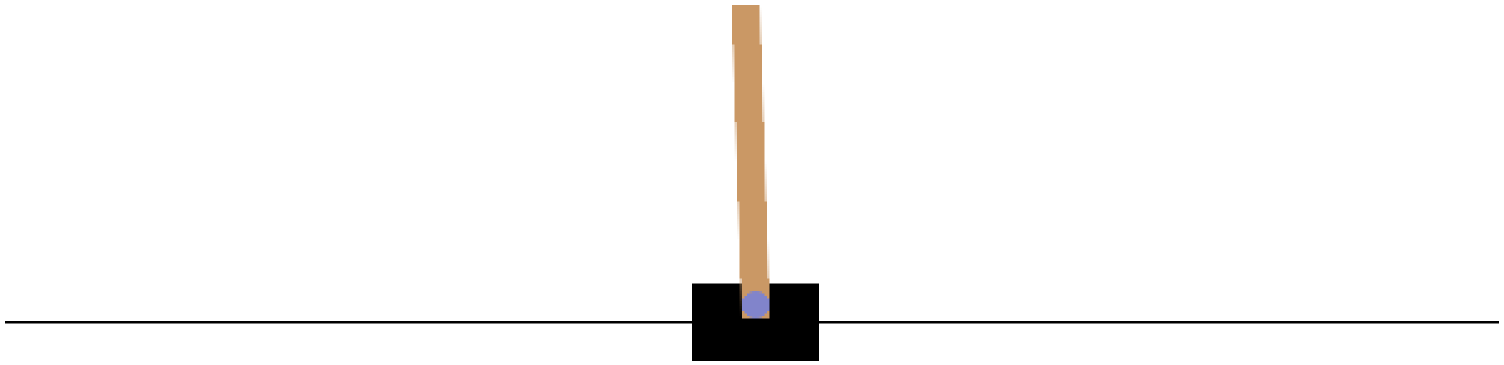
Soft-averaging – Polyak Averaging

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

- Hard update after many episodes
- Soft update shorter periods
- As usual, depends on the application
- Try them both!

The Program structure

What we want to learn



Program Structure
Structure the program in 6 major Modules

Initializations

Hyperparameters

Network – Support Functions (Replay Buffer)
Train with replay buffer

Learning Loop

Visualize learning

Test 10 times
Generate video

Hyperparameters

Environment setup

In [4]:
```
env = gym.make("CartPole-v1")
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
tf.random.set_seed(221)                              # For reproducibility
```

Hyperparameters

In [5]:
```
max_steps = 1200
max_episodes = 1200
ROLLING_WINDOW = 40

batch_size = 64
gamma = 0.99
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.99
learning_rate = 0.0005
MEMORY_SIZE = 100000
num_episodes = 1000
solved_threshold = 200

tau = 0.05
retrain_steps_soft = 15          # We copy weights every retrain_steps
retrain_steps_hard = 500         # Soft update, it is high for long retrain periods. Small for short retrain

```

Neural Network

Neural Network definition

```python
# Build the neural network model
def build_model(state_size, action_size):
    inputs = Input(shape=(state_size,))
    x = Dense(16, activation="relu")(inputs)
    x = Dense(64, activation="relu")(x)
    x = Dense(16, activation="relu")(x)
    outputs = Dense(action_size, activation="linear")(x)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer=Adam(learning_rate=learning_rate), loss="mse")
    return model
```

Polyak optimization

Support Functions

```python
# Soft update function for target network
""" Soft Update using Polyak optimization """
def soft_update(model, target_model, tau):
    target_weights = target_model.get_weights()
    model_weights = model.get_weights()
    new_weights = [
        tau * mw + (1 - tau) * tw for mw, tw in zip(model_weights, target_weights)
    ]
    target_model.set_weights(new_weights)
```

Replay in and out

Replay Function - The core of the DDQN Algorithm

```python
# Replay buffer
replay_buffer = deque(maxlen=buffer_capacity)

# Add experience to replay buffer
def store_experience(state, action, reward, next_state, done):
    replay_buffer.append((state, action, reward, next_state, done))

# Sample experiences from the replay buffer
def sample_experiences(batch_size):
    indices = np.random.choice(len(replay_buffer), batch_size, replace=False)
    batch = [replay_buffer[i] for i in indices]
    states, actions, rewards, next_states, dones = zip(*batch)
    return (
        np.vstack(states),
        np.array(actions),
        np.array(rewards),
        np.vstack(next_states),
        np.array(dones, dtype=np.float32)
    )
```

The core of the program – Replay and train

```
# Double DQN target calculation
def experience_replay_with_ddqn(model, target_model, batch_size, gamma, tau, step):
    if len(replay_buffer) < batch_size:
        return

    states, actions, rewards, next_states, dones = sample_experiences(batch_size)

    # Predict Q-values for next states using both networks
    next_q_values = model.predict(next_states, verbose=0)
    best_actions = np.argmax(next_q_values, axis=1)
    target_q_values = target_model.predict(next_states, verbose=0)

    # Update Q-values using Double DQN formula
    targets = rewards + gamma * target_q_values[np.arange(batch_size), best_actions] * (1 - dones)

    # Update main Q-network
    q_values = model.predict(states, verbose=0)
    q_values[np.arange(batch_size), actions] = targets
    model.fit(states, q_values, epochs=1, verbose=0)

    # Apply soft update to target network
    if step % retrain_steps == 0:
        soft_update(model, target_model, tau)
```

Learning loop

```
# Training loop
epsilon = epsilon_start
episode_rewards = []
rolling_avg_rewards = []

start_time = time.time()

for episode in range(num_episodes):
    state, _ = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0
    done = False
    terminated = False
    truncated = False
    step = 0
    for e in range(episodes):                       # Should be While True, however we limit number of eps
        step = step + 1
        # Epsilon-greedy policy
        if np.random.rand() <= epsilon:
            action = np.random.randint(action_size)  # Explore
        else:
            action_vals = model.predict(state, verbose=0)
            action = np.argmax(action_vals[0])  # Exploit

        # Perform action
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated
        next_state = np.reshape(next_state, [1, state_size])
        total_reward += reward

        # Store experience
        store_experience(state, action, reward, next_state, done)

        # Update state
        state = next_state

        # Train using experience replay
        experience_replay_with_ddqn(model, target_model, batch_size, gamma, tau, step)

        if done:
            break

    # Decay epsilon
    epsilon = max(epsilon_min, epsilon * epsilon_decay)

    # Record reward
    episode_rewards.append(total_reward)
    rolling_avg = np.mean(episode_rewards[-ROLLING_WINDOW:])
    rolling_avg_rewards.append(rolling_avg)

    # Print progress
    print(f"Episode: {episode+1:3}/{num_episodes}, Reward: {total_reward:+8.2f}, "
          f"Epsilon: {epsilon:.2f}, Rolling Avg: {rolling_avg:5.2f}, Steps: {step:3}, Terminated: {done} ")

    # Check if environment is solved
    if rolling_avg >= solved_threshold:
        print(f"Environment solved in {episode+1} episodes!")
        model.save("lunarlander_ddqn_model1.keras")
        break

end_time = time.time()
```

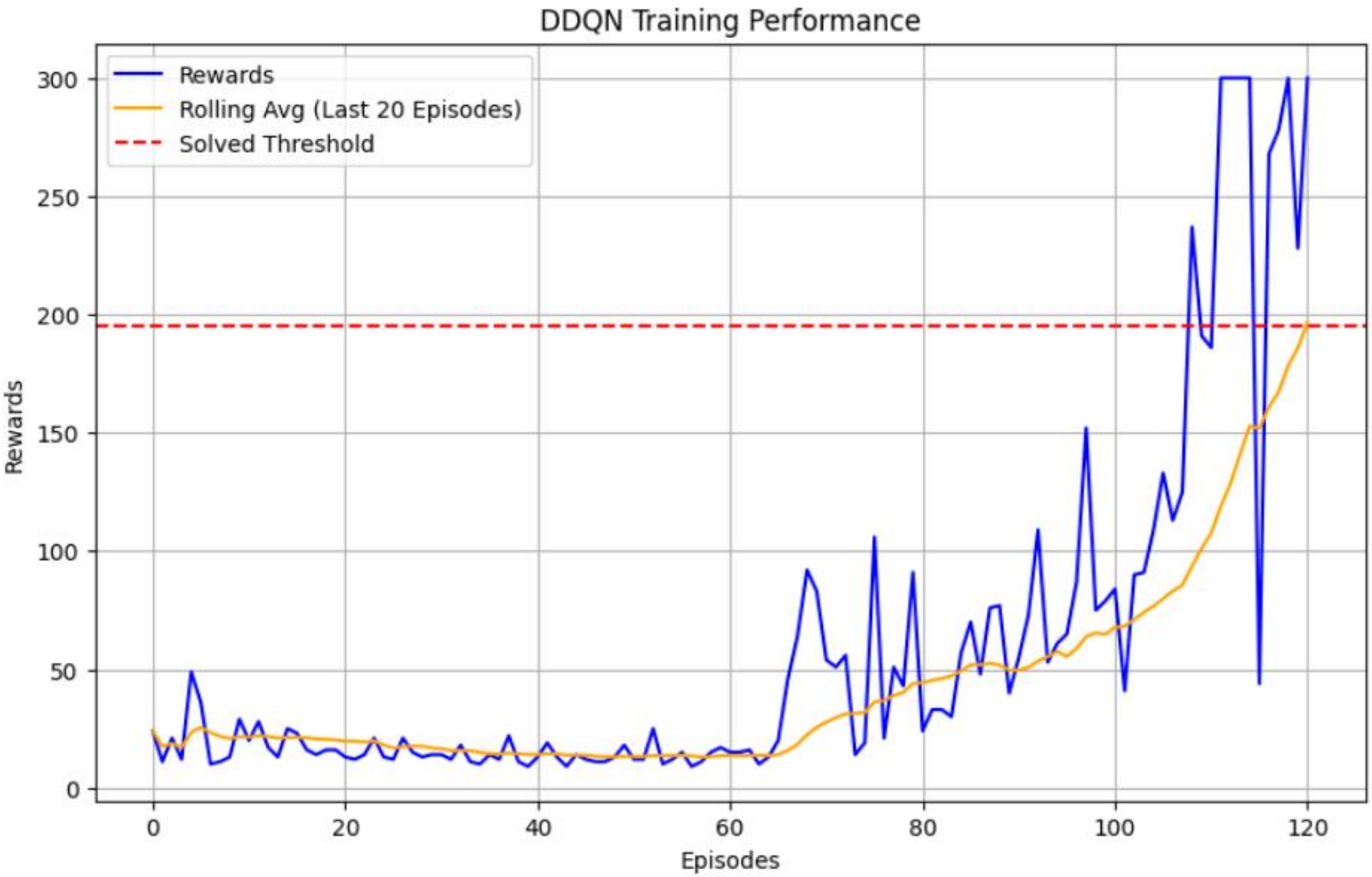
View Result

Result Visualization

In [11]:

```
# Plot rewards with rolling average
plt.figure(figsize=(10, 6))
plt.plot(episode_rewards, label='Rewards', color='blue')
plt.plot(rolling_avg_rewards, label='Rolling Avg (Last '+str(ROLLING_WINDOW)+' Episodes)', color='orange')
plt.axhline(y=solved_threshold, color='red', linestyle='--', label='Solved Threshold')
plt.title('DDQN Training Performance')
plt.xlabel('Episodes')
plt.ylabel('Rewards')
plt.legend()
plt.grid()
plt.show()
```

This is what we want to See



We test it

Testing 10 episodes with the DDQN trained networks

In [12]:

```python
# Testing for 10 episodes
start_time = time.time()
max_steps = 500

for e_test in range(10):  # Run 10 test episodes
    state, _ = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0

    steps = 0
    for s in range(max_steps):                            # we limit because sometimes it goes ad-aeternum
        # Use the trained model for testing
        action_vals = model.predict(state, verbose=0)  # Predict action values
        action = np.argmax(action_vals[0])  # Choose the action with the highest Q-value

        next_state, reward, done, _, _ = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        total_reward += reward
        state = next_state
        steps = steps + 1

        if done or (steps == max_steps):
            print(f"Test Episode: {e_test + 1}/10, Reward: {total_reward:.2f}, Steps: {steps:3}")
            break

end_time = time.time()
testing_duration = (end_time - start_time) / 60  # Convert to minutes
print(f"Testing completed in {testing_duration:.2f} minutes")
```

```
Test Episode: 1/10, Reward: 424.00, Steps: 424
```

Video Render

```
# Test the trained agent with video rendering
# This code is useful if you are using colab otherwise use render_mode='human'
env = gym.make(("CartPole-v1"), render_mode='rgb_array')  # Enable RGB rendering
frames = []  # Store frames for visualization

# Render a single test episode
state, _ = env.reset()
state = np.reshape(state, [1, state_size])
tot_rewards = 0

while True:
    # Use the trained model for action
    action_vals = model.predict(state, verbose=0)  # Predict action values
    action = np.argmax(action_vals[0])            # Choose the action with the highest Q-value

    next_state, reward, done, truncated, _ = env.step(action)
    frames.append(env.render())                   # Save frame for rendering later
    next_state = np.reshape(next_state, [1, state_size])
    tot_rewards += reward
    state = next_state

    if done or truncated:
        print(f"Rendered Test Episode Reward: {tot_rewards:.2f}")
        break

env.close()

# Save the rendered episode as a GIF
def save_frames_as_gif(frames, path='./', filename='CARTPOLE_DDQN.gif'):
    images = [Image.fromarray(frame) for frame in frames]
    gif_path = os.path.join(path, filename)
    images[0].save(gif_path, save_all=True, append_images=images[1:], duration=50, loop=0)
    print(f"Saved GIF to: {gif_path}")

save_frames_as_gif(frames, filename='CARTPOLE_DDQN.gif')

```

```
Rendered Test Episode Reward: 421.00
Saved GIF to: ./CARTPOLE_DDQN.gif
```

END

Session 9