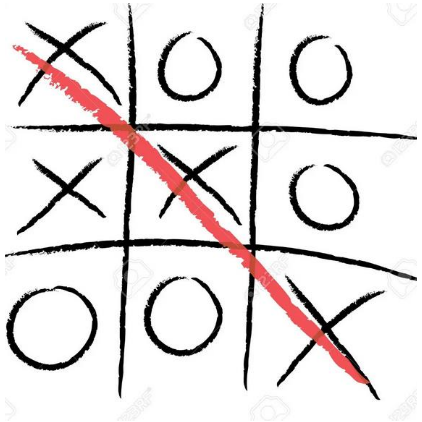
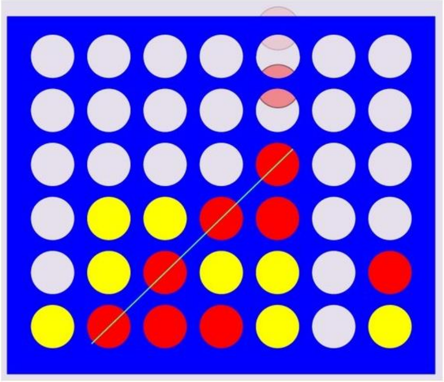
Using Deep Reinforcement learning for creating an adapting agent playing zero-sum games

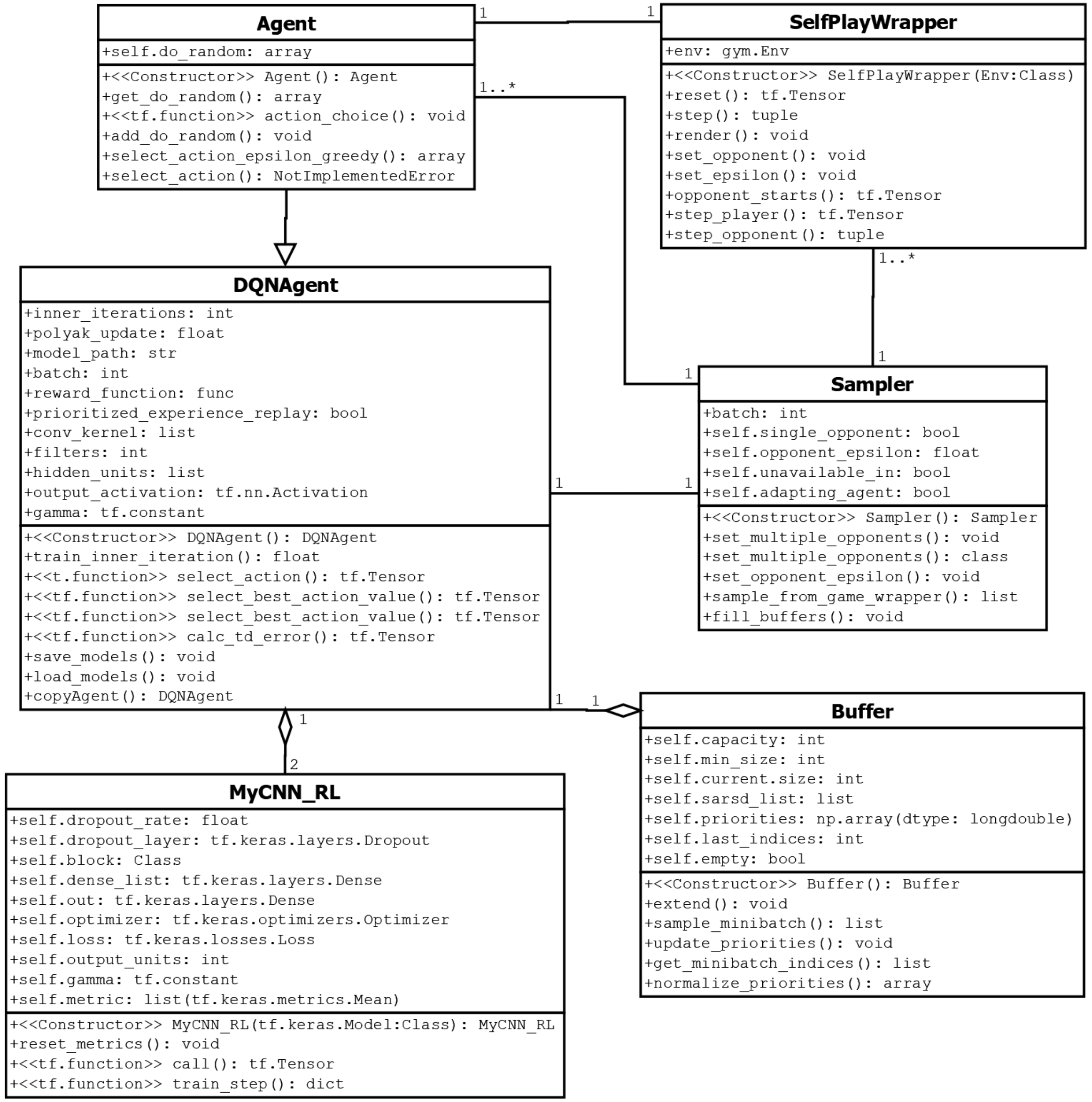




Project by:

Eosandra Grund

Fabian Kirsch



Project by:

Eosandra Grund

Fabian Kirsch

UNIVERSITÄT OSNABRÜCK

```python
class SelfPLayWrapper(Env):

    """ A Wrapper for Env similar to keras-gym Connect Four (adapted to a newer python version)
    also works for similar other envs

    Attributes:
        env (gym.Env): the gym env to wrap arount, has to be a 2 player game
        opponent (Agent): has to be an agent
        epsilon (float): the epsilon value for the epsilon greedy policy
        first_reward: will be used to save the reward, the agent got until the oponent did its step
        last_wrong: will be used to save whether the agent did a wrong move, to make sure the opponent cannot make a move
    """

    def __init__(self,env_class, opponent = None,epsilon : float = 0):
        super(Env, self).__init__()
        self.env = env_class()
        self.opponent = opponent
        self.epsilon = epsilon
        self.first_reward = None
        self.last_wrong = None
            ...

    def opponent_starts(self):
        """ let the opponent do the first action, works similar to reset"""
        s_0 = self.reset()
        # get the opponent's action
        o_action = self.opponent.select_action_epsilon_greedy(self.epsilon, tf.expand_dims(tf.cast(s_0, dtype = tf.float32), ax
False)[0]
        # do the opponent's action
        s_1,_,_ = self.env.step(o_action, return_wrong = False)

        return tf.cast(s_1, dtype=tf.float32)

     def step(self,a, unavailable_in : bool = False, agent = None):
        # do my step
        if unavailable_in:
            s_0,r_0,d_0, w_0 = self.env.step(a,return_wrong =  True)
        else:
            s_0,r_0,d_0 = self.env.step(a,return_wrong =  False)
            w_0 = False

        if d_0 or w_0:
            return tf.cast(s_0, dtype= tf.float32),r_0,d_0

        # get the opponent's action
        o_action = self.opponent.select_action_epsilon_greedy(self.epsilon,tf.expand_dims(tf.cast(s_0, dtype = tf.float32
False)[0]
        # do the opponent's action
        s_1,r_1,d_1 = self.env.step(o_action,return_wrong =  False)
        # calculate the returns
        if d_1:
            return tf.cast(s_1, dtype= tf.float32),self.env.loss_reward if r_1 == self.env.win_reward else r_1, d_1

        return tf.cast(s_1, dtype= tf.float32),r_0,d_1
```

```python
class DQNAgent(Agent):
    """ Implements a basic DQN Algorithm

    Attributes:
        model (MyCNN_RL): the model to train
        target_model (MyCNN_RL): the target_model
        inner_iterations (int): how many inner iterations to do while training
        polyak_update (float): how big the polyak update should be
        model_path (str): where to load and save the models
        buffer (Buffer): the buffer to save and sample data from
        batch (int): how big minibatches to sample from the buffer
        reward_function (func): the function to transform reward gotten from the environment with
        prioritized_experience_replay (bool): whether to use prioritized_experience_replay
        conv_kernel (list): a list of the conv_kernels to use to create new model and target_model for the copy
        filters (int): how many filters to use for the conv layers when making a copy
        hidden_units (list): list of the hidden_units for dense layers for making a copy
        output_activation (tf.nn.Activation): function to be the output activation of model and target_model (used for copy)
        gamma (tf.constant)float: the discount factor for future rewards
        """

    def __init__(self, env, buffer, batch : int, model_path, polyak_update = 0.9, inner_iterations = 10, reward_function = lambda d,r: r,
                conv_kernel = [3], filters = 128, hidden_units = [64], dropout_rate = 0.5, normalisation : bool = True, prioritized_experience_replay : bool = True,
                gamma : tf.constant = tf.constant(0.99),loss_function = tf.keras.losses.MeanSquaredError(), output_activation = None):

        super().__init__()

        # create an initialize model and target_model
        self.model = MyCNN_RL(conv_kernel = conv_kernel, filters  = filters, hidden_units = hidden_units, output_units = env.action_space.n,
                            output_activation = output_activation, loss = loss_function,
                            dropout_rate = dropout_rate, normalisation = normalisation, gamma = gamma)
        self.target_model = MyCNN_RL(conv_kernel = conv_kernel, filters  = filters, hidden_units = hidden_units, output_units = env.action_space.n,
                                   output_activation = output_activation, loss = loss_function,
                            dropout_rate = dropout_rate, normalisation = normalisation, gamma = gamma)

        # build models
        obs = tf.keras.layers.Input(shape=env.reset().shape)
        env.close()
        self.model(obs)
        self.target_model(obs)
        self.target_model.set_weights(np.array(self.model.get_weights(),dtype = object))

        # save other variables as attributes
        self.inner_iterations = inner_iterations
        self.polyak_update = polyak_update
        self.model_path = model_path
        self.buffer = buffer
        self.batch = batch
        self.reward_function = reward_function
        self.prioritized_experience_replay = prioritized_experience_replay
        self.conv_kernel = conv_kernel
        self.filters = filters
        self.hidden_units = hidden_units
        self.output_activation = output_activation
        self.gamma = gamma
```


Project by:

Eosandra Grund

Fabian Kirsch

UNIVERSITÄT OSNABRÜCK

```python
class Buffer:
    """
    Implemens a replay buffer for our DQNAgent class. Can use prioritized experience replay.

    Attributes:
        capacity (int): the maximum capacity
        min_size (int): the minimum filling size for starting training
        current_size (int): the current size of the buffer
        sarsd_list (list): contains the data
        priorities (np.array): contains the priorities for the sarsd_list elements with the same index
        last_indices (np.array): contains the indices of the last minibatch, for updating the priorities afterwards
        empty (bool): If the buffer is empty
    """

    def __init__(self, capacity, min_size):
        self.capacity = capacity
        self.min_size = min_size
        self.current_size = 0
        self.sarsd_list = []
        self.priorities = np.array([],dtype=np.longdouble)
        self.last_indices = None
        self.empty = True # is False after adding data the first time

    def extend(self, sarsd):
        """ adds new data to memory buffer """
        """
        Prameters:
        _____________
        sarsd (list): list of new data samples to add to the buffer. each sample being in the order
            state, action, reward, new_state, done, available_next_action_bool given as a tuple
        _____________

    def sample_minibatch(self, batch_size):
        """ samples a minibatch from the buffer """
        """
        Prameters:
        _____________
        batch_size(int) = The sample size pulled from sarsd_list
        _____________


        Returns:
        _____________
        sample(list) = A sample pulled from sarsd_list containing [state, action, reward, next_state, done]
        _____________
        """
```

```python
class Sampler:
    """
    Implements an algorithm to sample from a self-play environment using two different agents

    Attributes:
        envs (list): List of all the environments to sample from
        batch (int): how many environments to sample from at the same time
        agent (Agent): the agent to sample for
        opponent (Agent) or list of Agents: The opponent to use or use several
        single_opponent (bool): whether opponent is a single agent or a list
        opponent_epsilon (float): the epsilon value to use for the epsilon greedy policy of the opponent(s)
        unavailable_in (bool): if True, the sampling agent can decide for unavailable actions and gets a penalty as reward back and the env state does not change
        adapting_agent (bool): whether the sampling agent is an adapting agents that wants information about the rewards other than putting it in the buffer.
            Also there is information about the average rewards of the past and the opponents level (decided by the agent) added to the buffer.
    """


    def sample_from_game_wrapper(self,epsilon : float, save = True):
        """
        samples from env wrappers

        Parameters:
            epsilon (float): the epsilon to use for the epsilon greedy policy of the sampling agent
            save (bool): whether the created samples should be saved in the sampling agents buffer
        """

        sarsd = []
        current_envs = self.envs
        agent_turn = np.random.randint(0,2,(self.batch,))
        observations = np.array([env.opponent_starts() if whether else env.reset() for whether,env in zip(agent_turn,current_envs)])

        for e in range(10000):

            # agent turn
            available_actions = [env.available_actions for env in current_envs]
            available_actions_bool = [env.available_actions_mask for env in current_envs]
            actions = self.agent.select_action_epsilon_greedy(epsilon, observations,available_actions, available_actions_bool, unavailable = self.unavailable_in)

            if self.single_opponent:
                o_0 = np.array([env.step_player(actions[i],self.unavailable_in) for i,env in enumerate(current_envs)]) # only state for opponent imput

                # opponent turn
                available_actions = [env.available_actions for env in current_envs]
                available_actions_bool = [env.available_actions_mask for env in current_envs]
                o_actions = self.opponent.select_action_epsilon_greedy(self.opponent_epsilon,o_0,available_actions, available_actions_bool, False)
                results = [env.step_opponent(o_actions[i]) for i,env in enumerate(current_envs)] # new state, reward, done,
            else:
                # here the different opponents in the envs are used
                results = [env.step(actions[i],self.unavailable_in) for i,env in enumerate(current_envs)]

            # if adapting agent give the agent information about the player level (only reward from done envs)
            if self.adapting_agent:
                self.agent.add_game_balance_information([results[i][1] for i in range(len(current_envs)) if results[i][2]])

            # get next available actions, as we also save that in the buffer
            available_actions_bool = [env.available_actions_mask for env in current_envs]
```

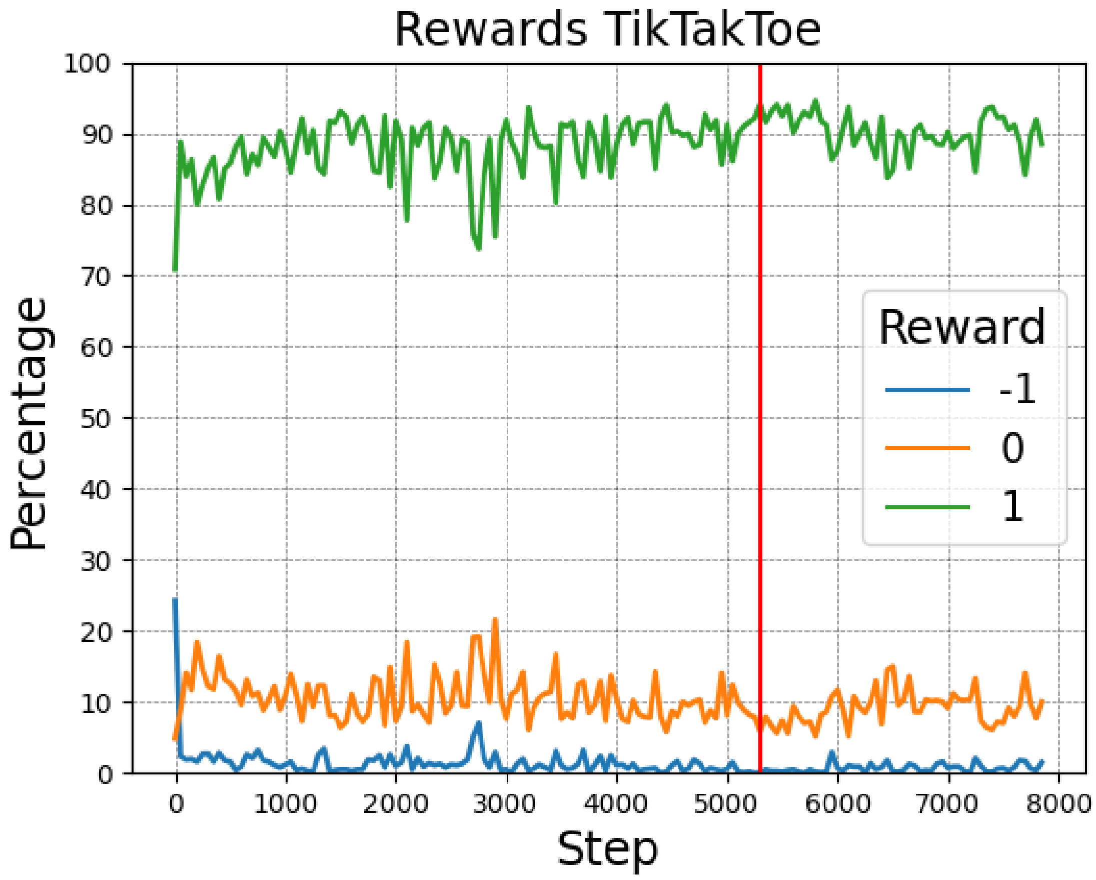
...

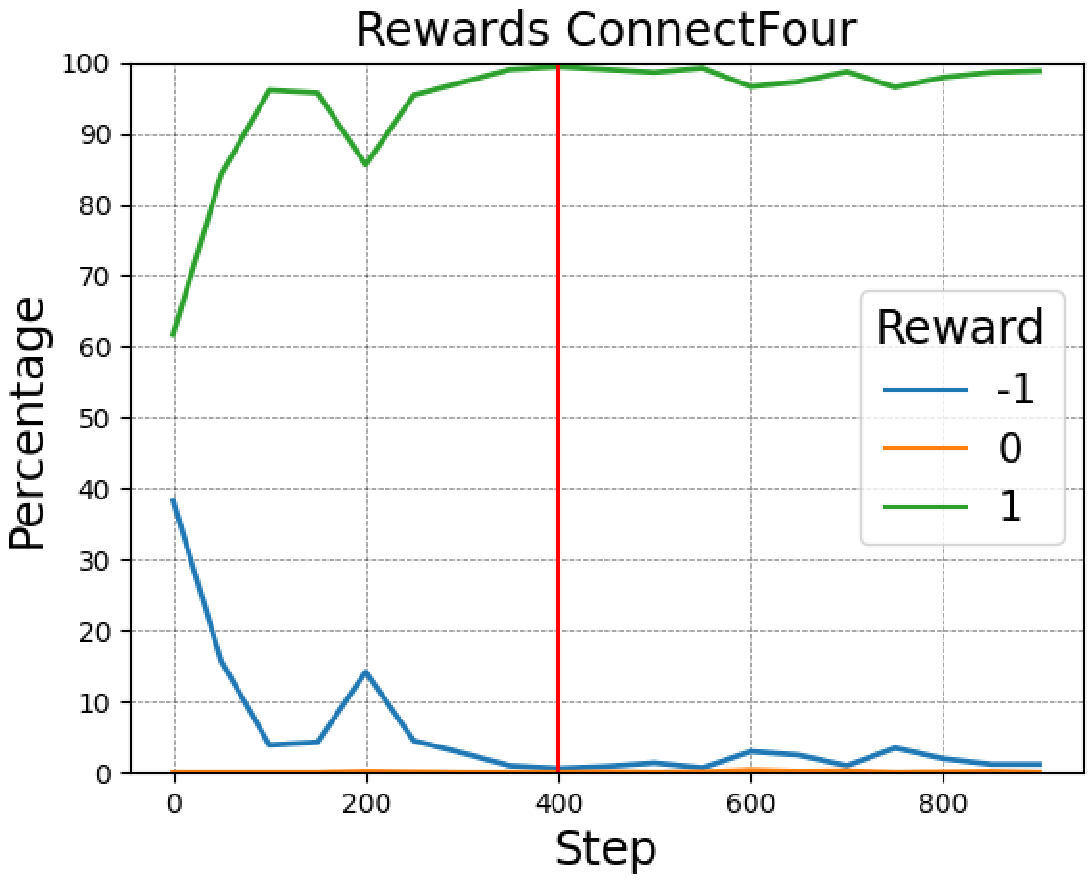
From training.py

```python
def train_self_play_best(agents : list, env_class, batch_size_sampling : int, iterations : int, writers : list, epsilon = 1,
                         epsilon_decay : float = 0.9, epsilon_min : float = 0.01, sampling : int = 1, unavailable_in : bool = False, opponent_epsilon = lambda x: (x/2),
                         d : int = 20, testing_size : int = 100):
    ...
```

UNIVERSITÄT OSNABRÜCK

Best Agents





Project by:

Eosandra Grund

Fabian Kirsch

UNIVERSITÄT OSNABRÜCK

```python
class AdaptingAgent(Agent):
    """
    contains a best agent, whose model outputs the expected future reward for every state-action pair.
    This agent then chooses to use the action with the future reward closest to 0.

    Attributes:
        best_agent (DQNAgent): The agent to use for getting the expected future reward
        self.model: the model of the best_agent
        self.target_model: the target_model of the best_agent
        self.game_balance (numpy.array / list): 1D containing the last reward values
        self.max_balance_length (tf.constant, tf.float32): how many rewards from the past will be saved max
        self.opponent_level (tf.constant): the level of the opponent given by the model (only used in AdaptingDQNAg
    """
def action_choice(self, probs, available_actions_bool = None):
    """
    returns best action, in this case, that makes the future reward closed to 0

    Parameters:
        probs (tf.Tensor): (batch, model output size) the model output to choose an action from
        available_actions_bool (tf.Tensor): a mask showing which actions are available
    """
    # choose action that makes future game_balance closest to zero
    return tf.argmin(tf.math.abs(probs),axis=-1)
```

Equation 1:

$$\text{best}Q_{sa} = \min \sqrt{Q_{s\vec{a}}^2}$$

```python
class AdaptingAgent5(AdaptingAgent3):
    """ normalizes the expected future reward and then chooses the value closest to - game_balance instead of just 0 """

    def __init__(self, best_agent : DQNAgent, calculation_value : tf.constant = tf.constant(0.3), game_balance_max : int = 500):
        super().__init__(best_agent,tf.constant(0.3), game_balance_max)
        self.calculation_value = calculation_value

    def action_choice(self, probs, available_actions_bool = None):
        """
        returns best action, in this case, that makes the future reward closed to minus the game balance, but scales the values around -1 and 1 first
        this function removes available actions if they are given.

        Parameters:
            probs (tf.Tensor): (batch, model output size) the model output to choose an action from
            available_actions_bool (tf.Tensor): a mask showing which actions are available
        """

        scaled_around_value = tf.subtract(tf.divide(probs,tf.reduce_max(tf.math.abs(probs))), -self.get_game_balance(tensor=True) * self.calculation_value)

        if available_actions_bool != None:
            scaled_around_value = tf.where(available_actions_bool, scaled_around_value, tf.constant(10.))
        adapting_action =  tf.argmin(tf.math.abs(scaled_around_value),axis=-1)

        return adapting_action
```

Equation 5:

$$B_r = \frac{\sum \vec{r}_i}{m}, \quad i \in [-m, \ldots, -1]$$

Equation 7:

$$\text{best}Q_{sa} = \min \sqrt{\left(\frac{Q_{s\vec{a}}}{\max \sqrt{Q_{s\vec{a}}^2}} - cp \times B_r \right)^2}$$

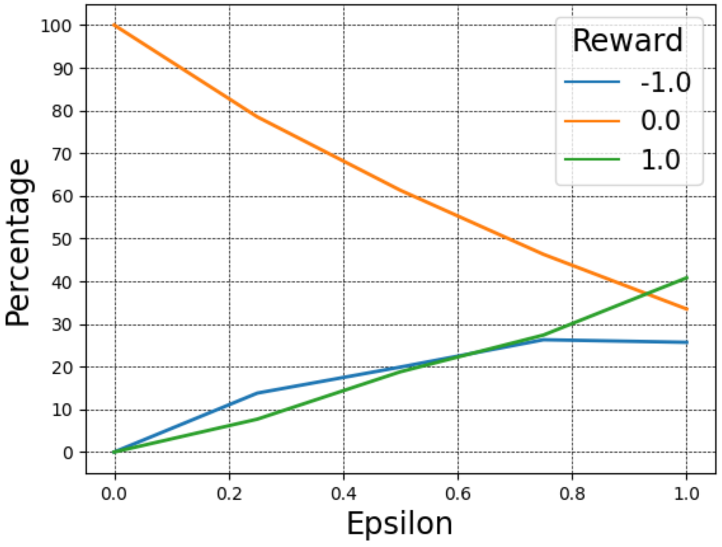
Project by:

Eosandra Grund

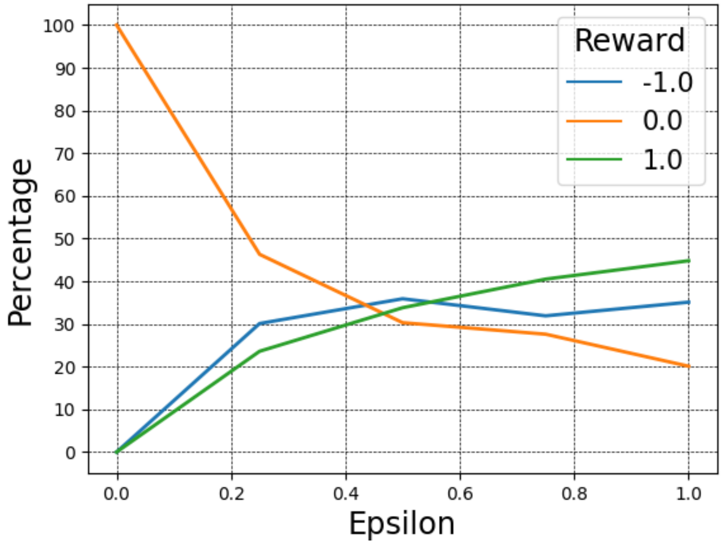
Fabian Kirsch

Tic-Tac-Toe

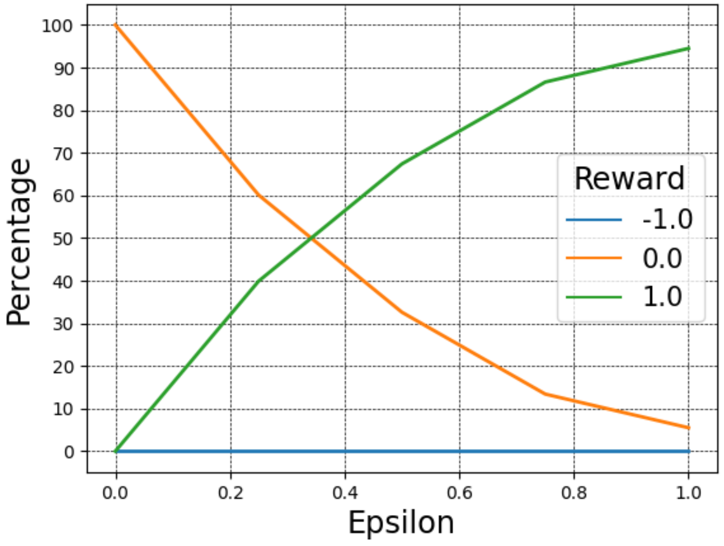
Adapting Agent with Eq1



Adapting Agent with Eq7,cp=2



High-Performance DQN



Project by:

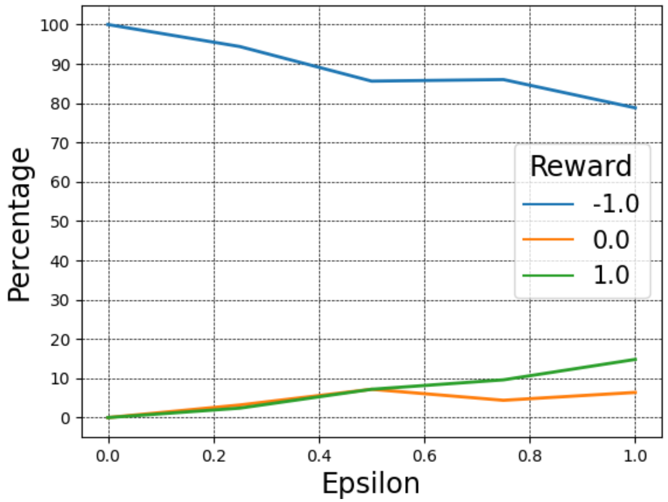
Eosandra Grund

Fabian Kirsch

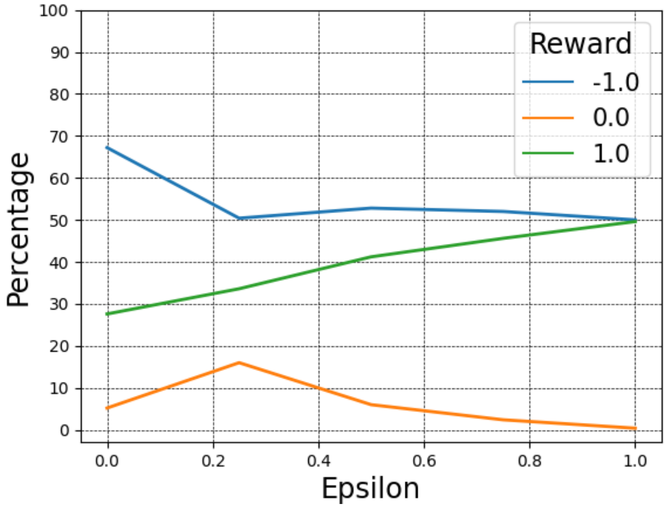
UNIVERSITÄT OSNABRÜCK

Connect Four

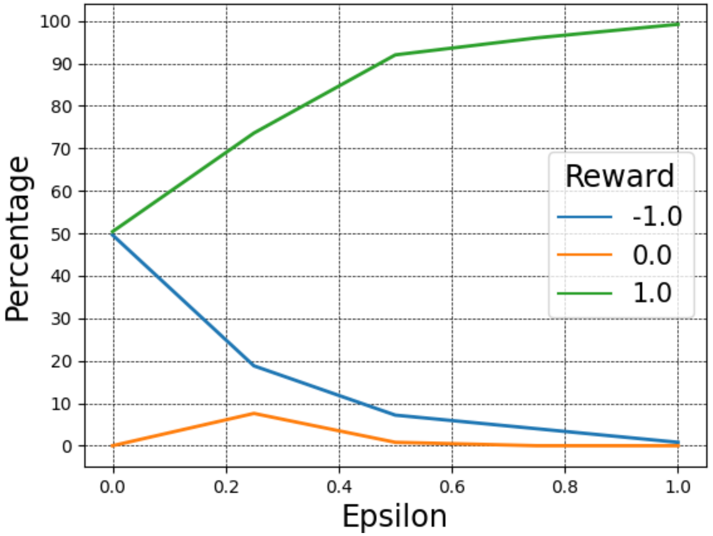
Adapting Agent with Eq1



Adapting Agent with Eq7,cp=3



High-Performance DQN



Project by:

Eosandra Grund

Fabian Kirsch

UNIVERSITÄT OSNABRÜCK

Improvements:

Project by:

Eosandra Grund

Fabian Kirsch