SIT320 Advanced Algorithms – Task 13
Te' Claire 220320033

Table of Contents

GitHub Link

https://github.com/leakydishes/advanced_algorithms/tree/main/13_design_patterns

Overview

A. *Continue working on tic-tac-toe problem from (distinction) Module 12, and attempt following extensions:*

(1) Integrate the idea of deep representation learning in your code. For this you can use a deep convolutional neural network to represent the state of the board and integrate it in your implementation of Q-Learning algorithm.

(2) Implement a policy gradient algorithm. You can use simple REINFORCE algorithm, do your own research, and decide which algorithm you will be implementing.

(3) Compare the performance of your policy gradient algorithm with deep Q-Learning algorithm.

Summary

Research for this task mainly included direct references to *Reinforcement Learning: An Introduction by Sutton R.S and Barto, A. G (1998)* [1] and Floris Laporte's article *Reinforcement from the Ground Up* (2020) [2].

In the Q-Learning algorithm (Implemented in Module 12) I created a tic-tac-toe bot (agent) which used reinforcement learning. Reinforcement Learning (RL) addresses '…sequential decision-making problems that are typically under uncertainty' or a '…learning paradigm that learns to optimise sequential decisions' [1].

RL *'State-Action-Reward'* framework

A State: the problems features (information), measured/ unmeasured variables that are known/ unknown.
An Action Space: In each state of system, the decisions that will be taken.
A Reward Signal: Feedback of performance using a scaler signal, to learn which actions are needed in each state.
Constraints: Example, a reward with negative penalty.
Uncertainty: System randomness

A policy (*Table*) informs us of actions to take in which state, $Q(s,a)$ = Q-table.
Deterministic Policy: At each state only learn which action to take.
Exploration: The action with the highest probability
Stochastic Policy: To learn the probability of each action
Exploitation: A sample of the distribution based on the probabilities

This strategy (*policy)* in a dynamic system is where RL attempts to win a policy by taking actions in different states of the system.

$$Q_t r_{t+1}(s,a) = r_t + \gamma \frac{max}{a'}\ Q(s',a')$$

Updates the state-action pair with top node, root of update (small/ filled action node).

Q-Learning Algorithm [1, p. 132]
Step Size $\propto \in ([0,1], small\ \in >\ 0$
Initialise $Q(s,a)$, for all s $\in\ S^+, a\ \in A(s)$, arbitrarily except $Q(terminal,\ \cdot) = 0$
Loop for each episode (game):
Initialise S
Loop for each step of game:
Choose A from S using policy from Q
Take *action A, observe* $R,\ S'$
$Q(S,A) \leftarrow Q(S,A) + \propto [R + \ \gamma\ max_a\ Q(S',a) - Q(S,A)]$
$S\ \leftarrow\ S'$
until S is terminal

A type of RL, Temporal-difference (TD) is the combination of ideas from Monte Carlo Method and Dynamic programming (DP), a type of 'bootstrap', doesn't wait for final outcomes when updating estimates based on its learned estimates [1, p. 119]. Monte Carlo method estimates systems by random sampling, estimate the value of state or state-action pairs in Markov decision processes. This method is used in TD (Q-Learning) to update value estimates based on different predicted and actual rewards. Where the action value function $Q(s,a)$ is learnt and updated iteratively, whereas (as learnt in Module 12) Monte Carlo methods waits until the end of an episode (game) to update value estimates.

Overall TD uses experience (updating an estimate $V\ of\ V_\pi$) to solve prediction problems in nonterminal states S_t while following a policy π. This is advantages over DP, as TD doesn't require a model of the environment, when selecting reward and next-state probability distributions.

Classical RL is different to supervised ML (Machine Learning) [1].

ML Supervised: (Value Function), aims to predict the known (assumptions/ predictions of input data). Learning of the prediction model is via supervision, where the learning objective is accuracy maximisation.

Classical RL: (Policy mapping), aims to produce a policy (strategy) which computes the next best action/ (learning based) decision making, combinations of current state, action, next state, reward. Each are interdependent and is not tabular like ML. '…there is no ground truth or supervision' [1].

However, Q-Learning is an off-policy RL, the agent learns the value of state-action pairs (Q-values) while following a different policy to the one it is currently improving. Sutton and Barto (1998) explain this as 'afterstate value functions' [1, p. 137], as the agent doesn't have an option of selection an action (in conventional state-value functions) and instead evaluates board positions after the agent had made a move. In this problem (tic-tac-toe) this method of RL creates a more '…efficient learning method' [1, p. 137]. Challenges of Q-Learning maintenance of the Q table, enumerating all the possible states and it is impossible to be able to present all the states.

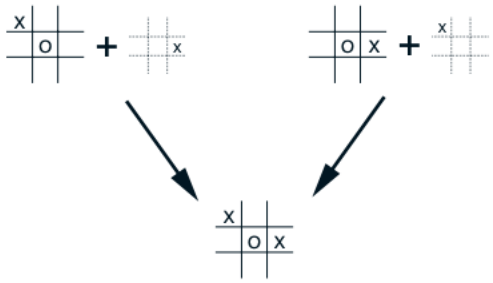


Fig. 1 Tic-tac-toe TD method [1, p. 137]

TD is classified as on-policy or off-policy (Q-Learning) with other methods which are driven by TD errors are '…one-step, tabular and model-free' [1, p. 138]. Whereas, *Deep Neural Network* (DNN) a form of RL, it uses the value function and the policy mapping to provide more choices within a neural architecture.

Q Learning Algorithm Deep Convolutional Neural Network (Board State) Q-Learning Algorithm

(1) Integrate the idea of deep representation learning in your code. For this you can use a deep convolutional neural network to represent the state of the board and integrate it in your implementation of Q-Learning algorithm.

In previous modules (SIT320), it was evident that tic-tac-toe (game) could not be solved with 'classical techniques' [1, p. 8]. For example, Minimax algorithm assumes the player (opponent) will perform moves (actions) in a particular way, this assumption can be invalid/ incorrect. To find the most optimal solution DP can use the probabilities of an opponent's behaviour (each move calculated) or '…learn the model of the opponent's behaviour' [1, p. 9] for sequential decision problems (tic-tac-toe). However, many iterations of episodes (games) are required to estimate/ learn probabilities (using a value function method to evaluate all states).

This is where *Artificial Neural Networks* (ANN) can provide a bot (agent) the ability to generalise from similar states experienced in the past. An ANN is a '…network of interconnected units [nodes] that have some of the properties of neurons, the main component of the nervous system' [1, p. 223]. ANNs use a stochastic gradient method, they can learn value functions (from TD errors), maximise expected rewards or a policy gradient algorithm. Weights are adjusted to improve the networks performance from maximising or minimising (using an objective function).

A type of ANNs is Deep *Convolutional Neural Network* (CNN), '…specialized for processing high-dimensional data arranged in spatial arrays, such as images' [1, p. 227]. Each layer in CNN creates a multitude of feature maps (pattern of activity within an array of units), each unit performs the same operation on data (within a receptive field). Using different locations on the arrays of incoming data to store each unit of a feature map, thus units in the same feature map have the same weights.

This application of RL uses approximation/ estimates based on learning rather than tables, *Sutton and Barto (1998)* [1] examines DQN (deep Q-network), the combination of Q-learning with deep convolutional ANN. DQN doesn't require problem specific features and learnt from an Atari game different policies, using the same input data, network architecture and parameter values with different weights of the ANN set to random between episodes. QDN used a $\in -$greedy policy in a game emulator, returning a reward signal (+1, -1 or 0), with output units estimated by optimal action values (actions A_t at time t) of corresponding state-action pairs [1, p. 437]. Additionally, a version of Q-Learning is used to calculate the gradient using backpropagation (during A_t), updating vector weights w_t with pre-processed image 'stacks' S_t and S_{t+1},

$$w_{t-1} = W_t + \propto \left[R_{t+1} + \ \gamma \frac{\max}{\mathrm{a}} \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)\right] \nabla \hat{q}(S_t, A_t, w_t)$$

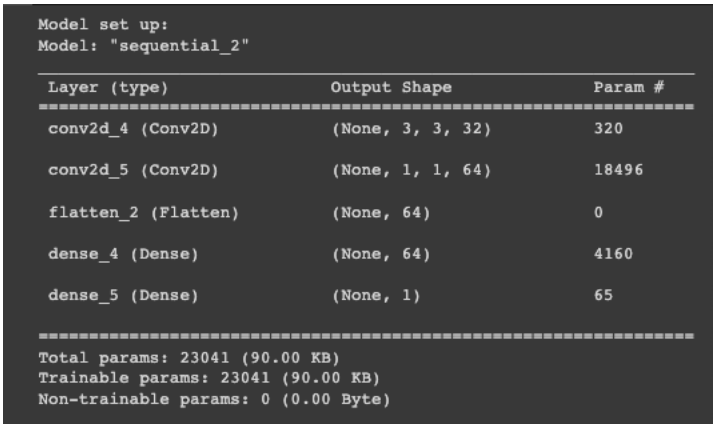
Note, DQN uses subsampling as part of each CNN layer, feature maps containing small sample of possible receptive fields using '…three successive hidden convolutional layers' [1, p. 439].

a. Board Representation: CNN Model

Deep Convolutional Neural Network (CNN), a model that can take the board (game) state as input. One or more hidden layers, an output layer (Q value) for all possible moves in the game state. I used TensorFlow [4] an open-source Machine learning (ML) (Python library from Google) to create the CNN model, I don't have the financial budget to use Google Collaboratory GPU thus I will be using the non-GPU format. I added padding to the layer before the first Conv2D layer in TensorFlow model when using (3x3 game state size) (MaxPooling2D layer) [4] this is to avoid zero or negative outputs of the model shape, this happens when you pass a model that is too small for the layer parameters.

I created a CNN model architecture summary, Fig. 1, architecture using the TensorFlow, this information in the table is the layers of the model and the output shapes of each layer

(number of trainable parameters). This table visualises the CNN model structure. This model has two convolutional layers (input/ output layers) and two middle layers.



```
Model set up:
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 3, 3, 32)          320

 conv2d_5 (Conv2D)           (None, 1, 1, 64)          18496

 flatten_2 (Flatten)         (None, 64)                0

 dense_4 (Dense)             (None, 64)                4160

 dense_5 (Dense)             (None, 1)                 65

=================================================================
Total params: 23041 (90.00 KB)
Trainable params: 23041 (90.00 KB)
Non-trainable params: 0 (0.00 Byte)
```

Fig. 1 Model Architecture Summary CNN (3x3 game state)

To encode the data for the CNN, need to consider either using a single node and feeding it a unique hash value of the board or using an array with each element encoding the value of the piece (specific position as an integer). I have prepared the data for the input layer to be, 'O' = 1 (9bits), 'X' -1 (9bits), and empty space = (0) (9bits), representing a total of 27 bits. The output layer will consist of 9 nodes each representing a position on the board, the values in the output nodes are the Q values for the corresponding moves (the estimate future reward for that given state).

b. Modify Q-Learning Algorithm

My aim of this modification is to train the CNN to find playable spaces to predict the best X and O actions (moves) as more layers are added to the network. CNN can use regression to use less space than tabular format (Q-Learning) and mimic the Q function to predict what the next action (move) might be in the game.

Floris Laporte's article *Reinforcement from the Ground Up* (2020) [2] was used to study Q learning with CNN. I found this topic difficult to implement into the previous tic tac toe game methods, however I still wanted to see how this algorithm effected the game play. The QModel class, Fig. 2 consists of an embedding layer, three fully connected (linear) layers and ReLU activation functions. The forward methods define how the model processes input data with the save and load methods used for the model parameters. With a epsilon green policy as the strategy used, where the agent will choose random actions with a certain probability (epsilon) to explore moves.

```python
class QModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.embedding = torch.nn.Embedding(3, 3)
        self.layer1 = torch.nn.Linear(30, 300)
        self.layer2 = torch.nn.Linear(300, 300)
        self.layer3 = torch.nn.Linear(300, 9)
        self.relu = torch.nn.ReLU()

    def forward(self, states2d, turns):
        if not torch.is_tensor(states2d):
            states2d = torch.from_numpy(states2d)
        if not torch.is_tensor(turns):
            turns = torch.from_numpy(turns)
        assert states2d.dim() == 3 # batch dimension required
        assert turns.dim() == 1 # only dim = batch dim
        x = torch.cat([states2d.flatten(1), turns[:,None]], 1)
        x = self.relu(self.embedding(x)).flatten(1)
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        x = self.layer3(x)
        return x

    def _serialize_tensor(self, tensor):
        if tensor.dim() == 0:
            return float(tensor)
        return [self._serialize_tensor(t) for t in tensor]

    def _deserialize_tensor(self, tensor):
        return torch.tensor(tensor, dtype=torch.get_default_dtype())

    def save(self, filename):
        if not filename.endswith(".json"):
            filename += ".json"
        with open(filename, "w") as file:
            json.dump(
                {k: self._serialize_tensor(t) for k, t in self.state_dict().items()},
                file,
            )

    def load(self, filename):
        if not filename.endswith(".json"):
            filename += ".json"
        with open(filename, "r") as file:
            self.load_state_dict(
                {k: self._deserialize_tensor(t) for k, t in json.load(file).items()}
            )
        return self
```

Fig 2. QModel Class

The TicTacToe class, Fig. 3 play method simulates playing different numbers of games and records transitions (state, action, next state and reward). The play_turn method simulates a players turn and checks the outcome, the visualise state is used to show the output to terminal.

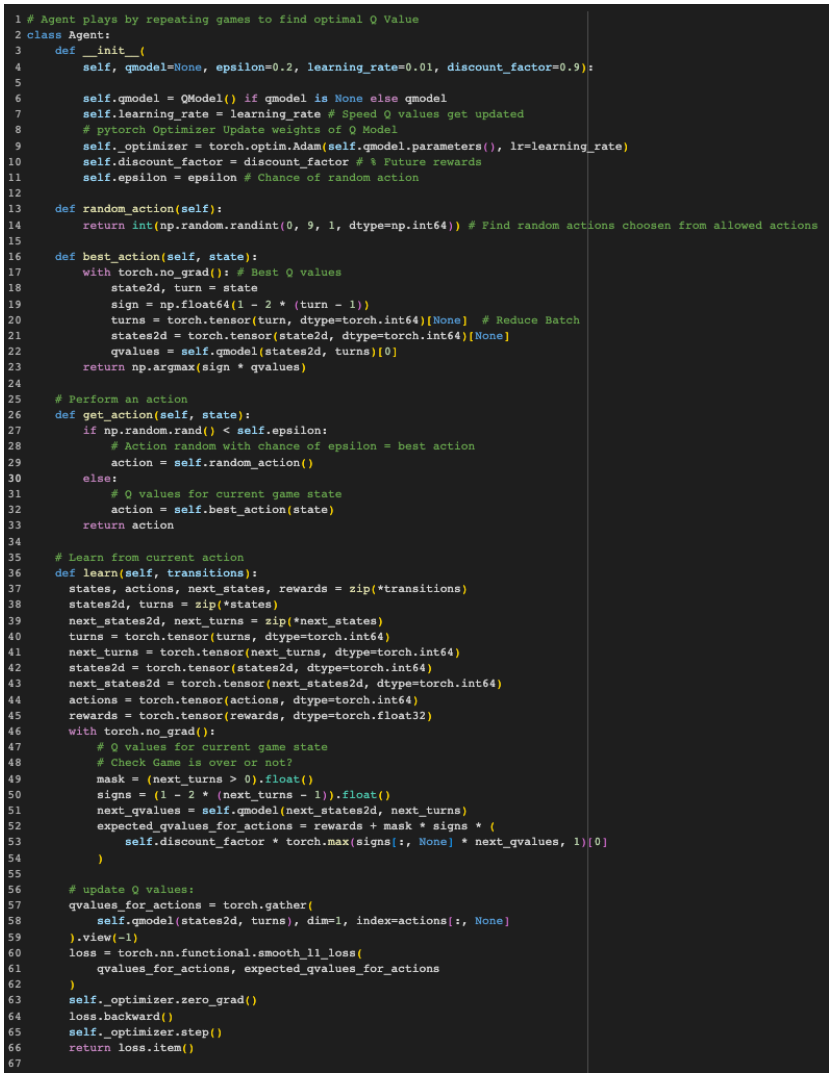
```python
class TicTacToe:
    def __init__(self, player1, player2):
        self.players = {1: player1, 2: player2} # Players against each other
        # Game Outcome (tie, player1 wins, player2 wins)
        self._reward = {0: 0, 1: 1, 2: -1}

    def play(self, num_games=1, visualize=False):
        transitions = []
        for _ in range(num_games):
            turn = 1
            state2d = np.zeros((3,3), dtype=np.int64)
            state = (state2d, turn) # full state of the game
            for i in range(9):
                current_player = self.players[turn]
                action = current_player.get_action(state)
                next_state, reward = self.play_turn(state, action)
                transitions.append(
                    (state, action, next_state, reward)
                )
                if visualize:
                    self.visualize_state(next_state, turn)

                (state2d, turn) = state = next_state

                if turn == 0:
                    break
        return transitions
```

```python
    # Current current player move check win/ loss
    def play_turn(self, state, action):
        state2d, turn = state # Find states
        next_state2d = state2d.copy()
        next_turn = turn % 2 + 1
        ax, ay = action // 3, action % 3  # Action two indices

        # Check space is legal
        if state2d[ax, ay] != 0:  # Check invalid move
            next_state2d.fill(0)
            next_state = (next_state2d, 0)  # next_turn == 0 -> game over
            return next_state, self._reward[next_turn]  # next player wins
        next_state2d[ax, ay] = turn # apply action

        # check if the action resulted in a winner
        mask = next_state2d == turn
        if (
            (mask[0, 0] and mask[1, 1] and mask[2, 2])
            or (mask[0, 2] and mask[1, 1] and mask[2, 0])
            or (mask[0, 0] and mask[0, 1] and mask[0, 2])
            or (mask[1, 0] and mask[1, 1] and mask[1, 2])
            or (mask[2, 0] and mask[2, 1] and mask[2, 2])
            or (mask[0, 0] and mask[1, 0] and mask[2, 0])
            or (mask[0, 1] and mask[1, 1] and mask[2, 1])
            or (mask[0, 2] and mask[1, 2] and mask[2, 2])
        ):
            next_state = (next_state2d, 0)  # next_turn == 0 -> game over
            return next_state, self._reward[turn]  # current player wins

        # if the playing board is full, but no winner found = draw
        if (next_state2d != 0).all():  # final draw
            next_state = (next_state2d, 0)  # next_turn == 0 -> game over
            return next_state, self._reward[0]  # no winner

        # if no move winner = next player's turn.
        next_state = (next_state2d, next_turn)
        return next_state, self._reward[0]  # no winner yet
```

Fig 3. TicTacToe Class

The Agent class, Fig. 4 uses the Q learning approach where the agent learns a Q value function to estimate the expected future rewards for each action. The agent takes random actions (probability determined) by the epsilon parameter which can be tested with more time. The best action method returns the best action based on the current Q values and the

get action decides this next action (random or based on Q values). The learn method updates these Q values using transitions recorded in each game.



```python
# Agent plays by repeating games to find optimal Q Value
class Agent:
    def __init__(
        self, qmodel=None, epsilon=0.2, learning_rate=0.01, discount_factor=0.9):

        self.qmodel = QModel() if qmodel is None else qmodel
        self.learning_rate = learning_rate # Speed Q values get updated
        # pytorch Optimizer Update weights of Q Model
        self._optimizer = torch.optim.Adam(self.qmodel.parameters(), lr=learning_rate)
        self.discount_factor = discount_factor # % Future rewards
        self.epsilon = epsilon # Chance of random action

    def random_action(self):
        return int(np.random.randint(0, 9, 1, dtype=np.int64)) # Find random actions choosen from allowed actions

    def best_action(self, state):
        with torch.no_grad(): # Best Q values
            state2d, turn = state
            sign = np.float64(1 - 2 * (turn - 1))
            turns = torch.tensor(turn, dtype=torch.int64)[None]  # Reduce Batch
            states2d = torch.tensor(state2d, dtype=torch.int64)[None]
            qvalues = self.qmodel(states2d, turns)[0]
        return np.argmax(sign * qvalues)

    # Perform an action
    def get_action(self, state):
        if np.random.rand() < self.epsilon:
            # Action random with chance of epsilon = best action
            action = self.random_action()
        else:
            # Q values for current game state
            action = self.best_action(state)
        return action

    # Learn from current action
    def learn(self, transitions):
      states, actions, next_states, rewards = zip(*transitions)
      states2d, turns = zip(*states)
      next_states2d, next_turns = zip(*next_states)
      turns = torch.tensor(turns, dtype=torch.int64)
      next_turns = torch.tensor(next_turns, dtype=torch.int64)
      states2d = torch.tensor(states2d, dtype=torch.int64)
      next_states2d = torch.tensor(next_states2d, dtype=torch.int64)
      actions = torch.tensor(actions, dtype=torch.int64)
      rewards = torch.tensor(rewards, dtype=torch.float32)
      with torch.no_grad():
          # Q values for current game state
          # Check Game is over or not?
          mask = (next_turns > 0).float()
          signs = (1 - 2 * (next_turns - 1)).float()
          next_qvalues = self.qmodel(next_states2d, next_turns)
          expected_qvalues_for_actions = rewards + mask * signs * (
              self.discount_factor * torch.max(signs[:, None] * next_qvalues, 1)[0]
          )

      # update Q values:
      qvalues_for_actions = torch.gather(
          self.qmodel(states2d, turns), dim=1, index=actions[:, None]
      ).view(-1)
      loss = torch.nn.functional.smooth_l1_loss(
          qvalues_for_actions, expected_qvalues_for_actions
      )
      self._optimizer.zero_grad()
      loss.backward()
      self._optimizer.step()
      return loss.item()

```

Fig 4. Agent Class

Mean Squared Error (MSE) (learning to mimic another function) is used to implement regression [1], [4]. The output of the CNN is used as the input to the loss function with discounted rewards and maximum Q values (next states) being the updated estimate of the Q function,

$$loss = MSE(Q(s,a), Q*(s,a))$$

c. Data Preparation

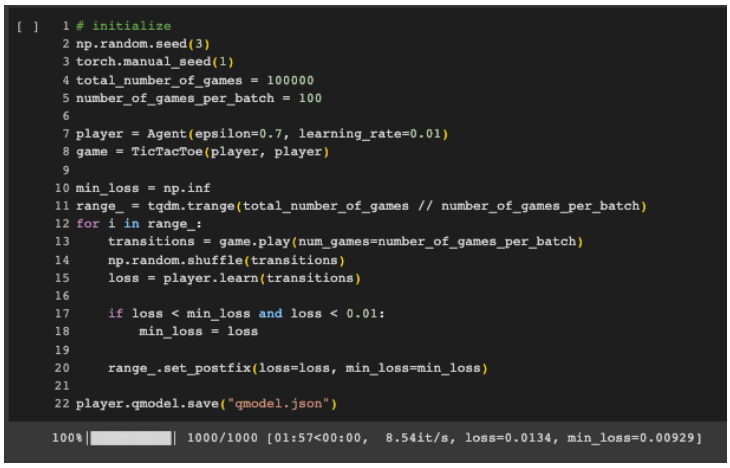
The data preparation for training the CNN model requires the current state (game/ board representation), action, target Q values (reward function + weighted sum of future Q values). The episodes of the game for each state (during gameplay) are run separately and stored in the state (input) and the corresponding A-vale (output).

The data preparation (2020) [2] involves creation and management of state transition during the RL training, these are used for Q learning and are stored as a list of tuples, this makes it easier to sample the transitions to train the agent. Batch training is used to create the Q

values and the data is converted to PyTorch tensors for CNN processing. The algorithm considers invalid moves and assigned rewards to state transitions. Additionally, the models parameters are serialized and saved as a JSON file, this saves time by not having to retrain the model each time by reloading the parameters.

d. Training CNN

The CNN model requires iterative training for collective nodes (using data collected during each episode/ game), according to the current policy, supervised learning to teach the model what Q-values should be for different board states. The aim is for each node to multiply each linear input (one or more) i_x with weights w_x and additional value *bias* b_x, this result is applied to a function f_a, the Activation Function (AF). The bias (a constant parameter) enables a shift in the feature map, giving more flexibility to the model [3], achieving non-linear outputs. The training for this implementation, Fig 5 uses a set hyperparameters than plays a specified number of games in a batch to collect transitions updating the agents Q values. The training selects moves with frequencies proportional to the probabilities outputted by the CNN. The hyperparameter (reward) is the 'winning' outcomes of actions within the time taken t_0 and the weighted sum of values w_x for this result with all future steps S_x in the game. A Gradient Decent Optimiser (GDO) [6], to adjust weights in the CNN is used to maximise this reward. Note, the MSE minimised the difference between predicted and actual Q- values. After the model had been trained, I played a game using the agents best estimate Q values.



```
[ ]  1 # initialize
     2 np.random.seed(3)
     3 torch.manual_seed(1)
     4 total_number_of_games = 100000
     5 number_of_games_per_batch = 100
     6
     7 player = Agent(epsilon=0.7, learning_rate=0.01)
     8 game = TicTacToe(player, player)
     9
    10 min_loss = np.inf
    11 range_ = tqdm.trange(total_number_of_games // number_of_games_per_batch)
    12 for i in range_:
    13     transitions = game.play(num_games=number_of_games_per_batch)
    14     np.random.shuffle(transitions)
    15     loss = player.learn(transitions)
    16
    17     if loss < min_loss and loss < 0.01:
    18         min_loss = loss
    19
    20     range_.set_postfix(loss=loss, min_loss=min_loss)
    21
    22 player.qmodel.save("qmodel.json")

100%|          | 1000/1000 [01:57<00:00,  8.54it/s, loss=0.0134, min_loss=0.00929]
```

Fig 5. Training

e. Evaluate

This implementation, Fig 6 combined Q learning with Deep CNN where the agent could learn complex strategies in the game using experience of prior games, this gave the agent more flexibility by allowing it to generalise its moves from one state to another. I believe these state transitions were efficient in data usage as the training was very quick to compile and the flexible hyperparameters meant I could test different applications (epsilon, discount factor). Hyperparameters require evaluation (CNN, Q-Learning) and the training process. Evaluating the performance of AI player against different game strategies to insure effective learning. The benefit of CNN is weight sharing, reducing the number of trainable network parameters (creating a faster algorithm) and avoids overfitting. The classification layer is

organised and high reliant on the features extracted and can be used in large-scale networks. However, I don't think for larger game state sizes this would be efficient, I struggled to implement a 5x5 board size and with more time I believe I would have proved this hypothesis. When I tried to change the epochs size it appeared to be very hypersensitive and was time consuming to find the best hyperparameters to test. Due to time limits I wasn't able to test the algorithm with other forms of RL, however I found it was able to enhance the RL difficulties into smaller 'supervised' [3] tasks where the maximum Q value produced for the Q learning overestimates the value of state actions [1, p. 137].

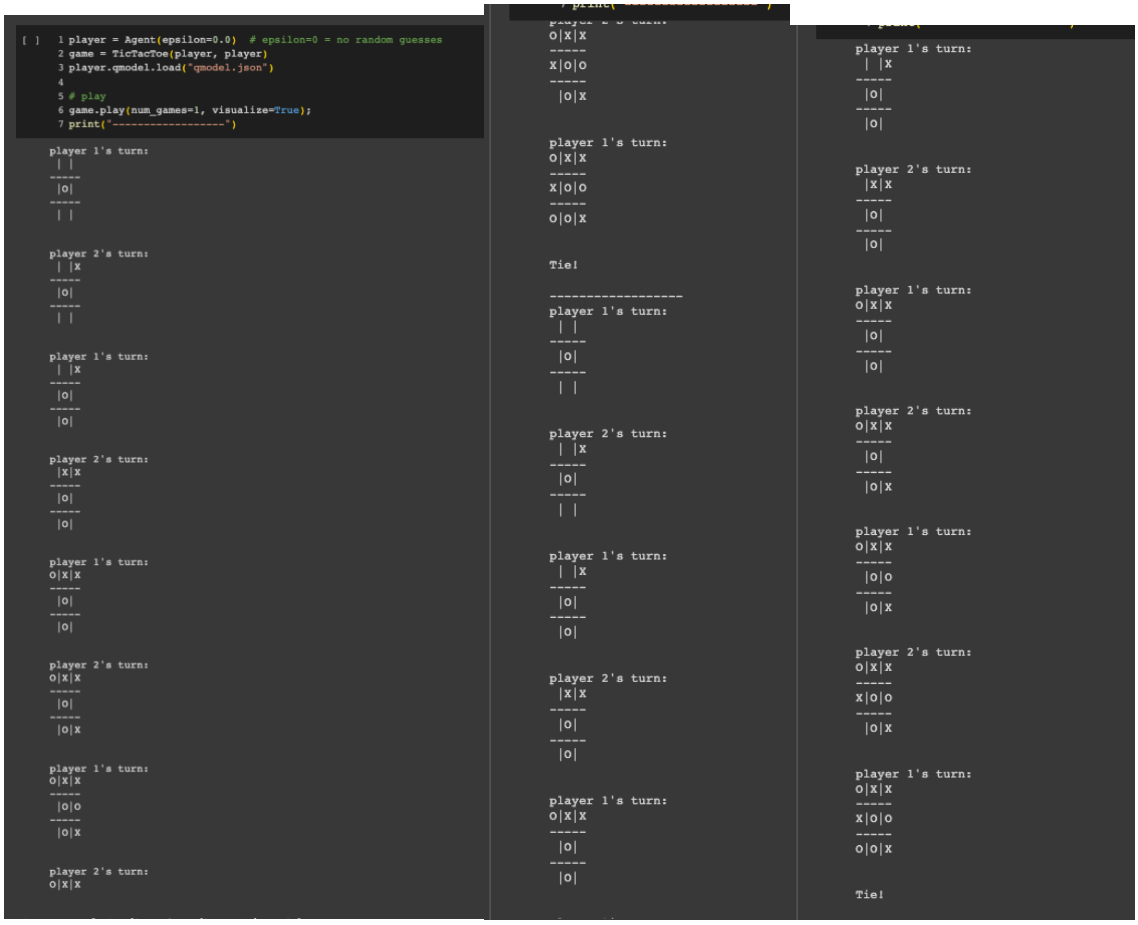


Fig. 6 Loading Data and Testing

Research Policy Gradient Algorithm

The main aim of RL agents is to maximise the expected reward when following a policy π, these are defined using parameters θ (weights, biases of units in the CNN). Where a network of nodes is to be trained on the weights and bias with feedback on process *Gradient Decent* [3]. Nodes are arranged in layers input nodes (first layer), middle layers (hidden layers) and output nodes (last layer), deep refers to multiple hidden layers within the CNN, with backpropagation used with middle layers, Fig. 7.

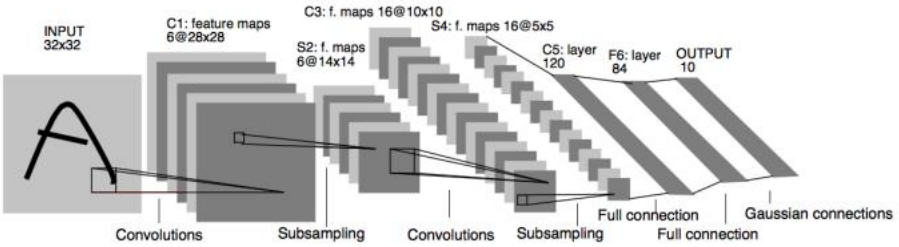


Fig. 7 Deep CNN [1, p. 227]

A policy is a distribution that the agent uses to find optimal actions, we can use a deep neural network to increase the probability of finding the most optimal actions which approximates the agent's policy [10]. From the previous tasks in this unit, I have learnt that policy updates use deep RL algorithms which are either value based, or policy based, this includes DQN discussed in the previous question, Fig. 8.

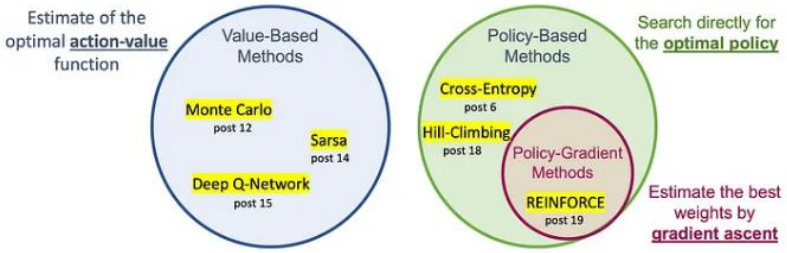


Fig. 8 Value method vs Policy based methods [13]

However, most algorithms use a value function which learns the value of an action to pick the best (optimal) action. A 'parameterized policy' [1] selects actions without depending on a value function, it can utilise this to learn a policy parameter. In RL the aim is to maximise the agent's performance (finding the most optimal actions) over a duration of time. Where the probability is the action a taken at a time t given the environment state s with parameter θ, thus a gradient of performance measured θ_t. Example, agent uses a function J_1 (weights and θ of the deep CNN), the updated rule for new θ is where it equals, old θ and additional learning resulting in new $\theta \cdot$ the gradient of that performance metric. Which is gradient accent [10]. This gradient will the proportional to the amount of time spent in each state including the sum of actions pairs and the gradient of that policy (gradient accent).

A CNNs network observes the environment as input and outputs actions selecting according to a SoftMax activation function [11] Fig. 9, alternatively the CNN would be a simple linear regression model only capable of returning '0s'. It generates a game (episode) and keeps track of that states, actions, rewards, and the agent's memory. It than revisits these states at the end of each episode (checking the states, actions, and rewards) to calculate the discounted future returns at each time step. The returns are than used as weights and the agents' actions as labels to than perform backpropagation to than update the weights of the Deep CNN. The agent repeats for several rotations until the most optimal actions are found.

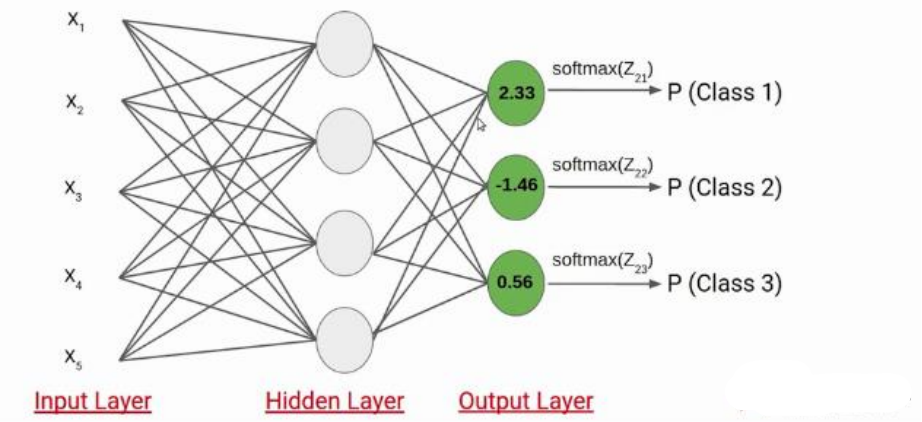


Fig. 9 SoftMax Activation Function applied to each neuron in CNN [11]

A Policy Gradient algorithm is an iterative approach aimed to '…reach the optimal policy that maximises the expected return' [10], with the parameter θ. This algorithm uses RL and adjusts a policy directly to find the best way to maximise the expected reward informed by the parameter policy [13], selecting actions that lead to the highest expected total reward.

$$\pi\ (a|s, \theta)$$

All MDPs '…have at least one optimal policy' [13], within the optimal policies (deterministic) there is a parameter θ which maximises J. The performance function J informs us how well the policy is functioning, picking actions that maximise the total expected reward. To achieve this the performance function $J(\theta)$ is maximised by the expected reward of the RL following the parametrised policy, where the trajectory $\tau = r(\tau)$ [13]. Trajectory (state-action-rewards sequences) were there are no restrictions on length.

$$J(\theta) = \mathbb{E}_{\pi}[r(\tau)]$$

This results in needing to maximise the J function, by adjusting on the parameters of θ to maximise the total expected reward. This process is called gradient ascent, where iteratively updating θ will increase the expected reward,

$$\theta_{t+1} = \theta_t + \propto \nabla J(\theta_t)$$

However, this is challenging as the actions selected and state distribution is dependent on this, additionally as environments are unpredictable this makes it difficult to plan policy updates in advance. The policy gradient theorem takes the gradient of the performance function J be reforming the equation to take a derivative of the function J, excluding the derivative of that state distribution. Thus, it allows us to compute the gradient of the performance J, $\pi_\theta(a|s)$,

$$\nabla \mathbb{E}_{\pi\theta}[r(\tau)] = \mathbb{E}_{\pi\theta}[r(\tau) \nabla log \pi_\theta(\tau)]$$

The parametrized policy doesn't directly depend on a reward, instead it depends on variances within each trajectory $r(\tau)$. Thus, past rewards don't contribute and $r(\tau)$ can be replaced with a discounted return G_t [13]. This results in a Policy Gradient algorithm *REINFORCE,* where we look at the overall performance in the agent's behaviour to guide the

policy improvement in terms of cumulative rewards. The RL agent uses the environment starting state to goal state (Monte Carlo Policy Gradient algorithm) [12], Fig 10 and unlike the TL or DP 'bootstrap' methods [1, p. 119].

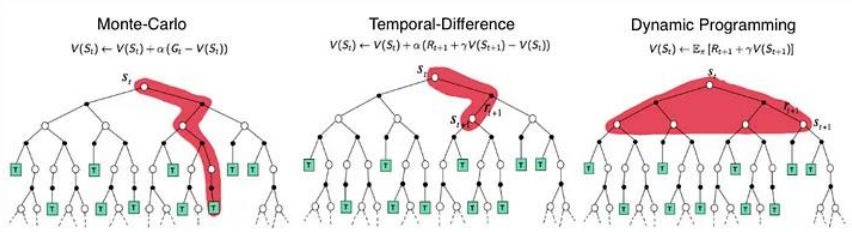


Fig. 10 Difference in Policy Gradient, Temporal Difference and Dynamic Programming [12]

To optimise the policy methods such as Maximisation Likelihood Estimate (MLE) can be used to iteratively adjust the policies parameters to select actions that will lead to higher rewards. As such policy gradient methods are a type or RL that optimise parameter policy that focus on the agent's behaviour not just the immediate rewards.

(2) Implement a policy gradient algorithm.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Fig. 11 Monte Carlo Policy Gradient Control [1, p. 328]

Policy gradient algorithms such as Monte Carlo, Fig 11 have significant advantages, they can learn to take actions for specific probabilities and efficient exploration by approaching deterministic policies 'asymptotically' [1, p. 337]. Additionally continuous action spaces are easily handled by policy gradient algorithms where action value methods can lake (ε-greedy). An algorithms performance can also be measured using the policy gradient theorem which does not involve state distribution (policy planning for future actions). And by 'adding a state-value function as a baseline reduce REINFORCE's variance without introducing bias' [1, p. 337]. This reduces bootstrapping methods which introduce bias (TD, DP), however these action value methods can reduce variance.

Compare The Convergence Performance of Algorithms

(3) Compare the performance of policy gradient algorithm with deep Q-Learning algorithm.

RL changes the trajectory results in for a multitude of rewards thus Monte Carlo Policy Gradient has 'high variance but zero bias' [14]. Whereas TD and DP (DQN) algorithms when used as a step (one action is used with a small change) results in a low variance. This can affect 'model convergence' as Policy Gradients are weak to variance (and mass producing

samples can impact efficiency), especially in on policy methods where behaviour policy and target policy are the same. Whereas off policy methods can improve exploration/ target policy without creating mass amounts of new samples, the more we know about a model's environment (dynamic) the less train and error we need to experiment with to find the most optimal policy, Fig. 12.

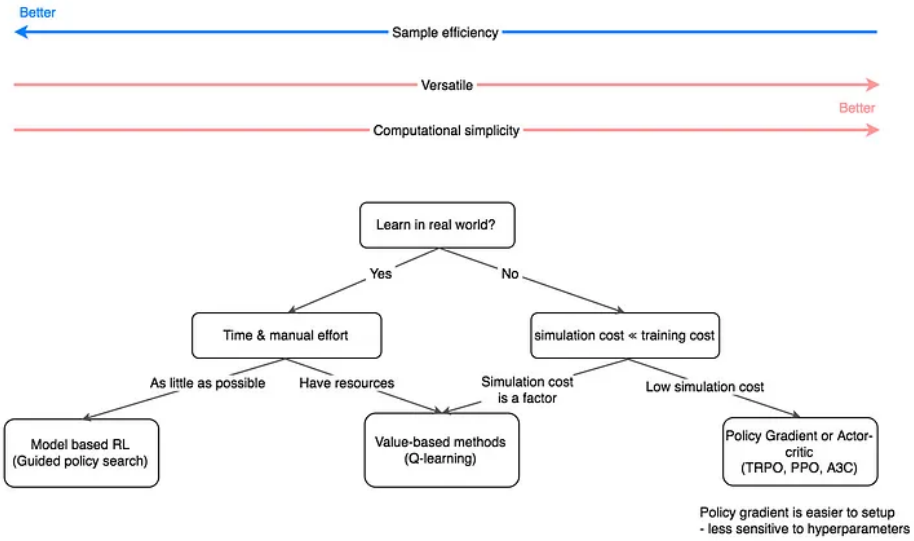


Fig 12. Sample efficiency (factor to choose the best algorithm) [14]

Many RL methods make assumptions (continuity) assuming that the state space or the control space is continuous (DQN) and Q learning with Deep CNN when in a continuous control space has to many complex steps [14]. This is due to the searching required of the entire control space to find the maximum Q value for the next action, this is computationally very difficult. Whereas policy gradient algorithms can support continuous control, as it 'optimises the policy directly' [14] by implementing constraints using policy parameters θ within the objective function J. However, the choice isn't necessary one or the other, we can have the best of both outputs by adding value learning to a policy gradient or adding a policy gradient to a RL [14].

References

[1] A. G. B. R. S. Sutton, Reinforcement Learning: An Introduction, Cambridge, Massachusetts: The MIT Press, 1998.

[2] D. S. L. V. N. P. T. Mummert, "What is reinforcement learning?," IBM Developer , 14 9 2022. [Online]. Available: https://developer.ibm.com/learningpaths/get-started-automated-ai-for-decision-making-api/what-is-automated-ai-for-decision-making/.. [Accessed 5 10 2023].

[3] TensorFlow, "Essential documentation," n.d. [Online]. Available: https://www.tensorflow.org/guide. [Accessed 5 10 2023].

[4] TensorFlow, "tf.keras.layers.MaxPooling2D," n.d. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPooling2D. [Accessed 5 10 2023].

[5] TensorFlow, "tf.keras.metrics.mean_squared_error," TensorFlow, n.d. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/metrics/mean_squared_error. [Accessed 5 10 2023].

[6] V. Jade, "https://towardsdatascience.com/texture-vs-shape-the-bias-in-cnns-5ee423edf8db," Medium, 30 5 2023. [Online]. Available: Texture vs Shape: The Bias in CNNs. [Accessed 5 10 2023].

[7] TensorFlow, “tf.compat.v1.train.GradientDescentOptimizer,” TensorFlow, n.d. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/GradientDescentOptimizer. [Accessed 5 10 2023].

[8] Ray, “Welcome to Ray! — Ray 2.7.1,” 2023. [Online]. Available: https://docs.ray.io/en/latest/. [Accessed 5 10 2023].

[9] Machine Learning, “How Policy Gradient Reinforcement Learning Works,” 2 5 2019. [Online]. Available: https://www.youtube.com/watch?v=A_2U6Sx67sE. [Accessed 10 10 2023].

[10] S. Kapoor, “Policy Gradients in a Nutshell,” Medium, 3 6 2018. [Online]. Available: https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d. [Accessed 12 10 2023].

[11] S. Saxena, “Introduction to Softmax for Neural Network,” Analytics Vidhya, 1 8 2023. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/04/introduction-to-softmax-for-neural-network/. [Accessed 11 10 2023].

[12] D. Karunakaran, “REINFORCE — a policy-gradient based reinforcement Learning algorithm,” Medium, 4 6 2020. [Online]. Available: https://medium.com/intro-to-artificial-intelligence/reinforce-a-policy-gradient-based-reinforcement-learning-algorithm-84bde440c816. [Accessed 13 10 2023].

[13] B. Gopaul, “Policy Gradient Algorithms | Reinforcement Learning,” 15 5 2019. [Online]. Available: https://www.youtube.com/watch?v=_coI2wFNKOY. [Accessed 12 10 2023].

[14] J. Hui, “RL — Reinforcement Learning Algorithms Comparison,” Medium, 20 10 2021. [Online]. Available: https://jonathan-hui.medium.com/rl-reinforcement-learning-algorithms-comparison-76df90f180cf. [Accessed 5 10 2023].

[15] J. Landy, “Deep reinforcement learning, battleship,” EFAVDB, 15 10 2016. [Online]. Available: https://www.efavdb.com/battleship. [Accessed 5 10 2023].