

# CA - agent and helper

## ChatGPT 3.5

**S You**

Be a computer science professor from top university of the world and answer my questions that follows, which can in best way help me to crack interviews based on a certain project.

 **ChatGPT**

Of course, I'd be happy to assist you with your interview preparation for a project-based interview. Please go ahead and ask your questions, and I'll do my best to provide you with comprehensive answers to help you prepare effectively.

**S You**

explain me this code in detail -

```
import torch
import random
import numpy as np
from collections import deque #ds to store memory
from game import SnakeGameAI, Direction, Point
from model import Linear_QNet, QTrainer
from helper import plot

MAX_MEMORY = 100_000
BATCH_SIZE = 1000
LR = 0.001

class Agent:

    def __init__(self):
        self.n_games = 0
        self.epsilon = 0 # randomness
        self.gamma = 0.9 # discount rate (smaller than 1)
        self.memory = deque(maxlen=MAX_MEMORY) # popleft()
        self.model = Linear_QNet(11, 256, 3)
        self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)

    def get_state(self, game):
        head = game.snake[0]
        point_l = Point(head.x - 20, head.y)
        point_r = Point(head.x + 20, head.y)
```

```
point_u = Point(head.x, head.y - 20)
point_d = Point(head.x, head.y + 20)

# only 1 of 4 is bool true , i.e, 1
dir_l = game.direction == Direction.LEFT
dir_r = game.direction == Direction.RIGHT
dir_u = game.direction == Direction.UP
dir_d = game.direction == Direction.DOWN
```

```
state = [
    # Danger straight
    (dir_r and game.is_collision(point_r)) or
    (dir_l and game.is_collision(point_l)) or
    (dir_u and game.is_collision(point_u)) or
    (dir_d and game.is_collision(point_d)),

    # Danger right
    (dir_u and game.is_collision(point_r)) or
    (dir_d and game.is_collision(point_l)) or
    (dir_l and game.is_collision(point_u)) or
    (dir_r and game.is_collision(point_d)),

    # Danger left
    (dir_d and game.is_collision(point_r)) or
    (dir_u and game.is_collision(point_l)) or
    (dir_r and game.is_collision(point_u)) or
    (dir_l and game.is_collision(point_d)),

    # Move direction
    dir_l,
    dir_r,
    dir_u,
    dir_d,

    # Food location
    game.food.x < game.head.x, # food left
    game.food.x > game.head.x, # food right
    game.food.y < game.head.y, # food up
    game.food.y > game.head.y # food down
]
```

```
return np.array(state, dtype=int) # converts T/F to 1/0
```

```
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done)) # popleft if MAX_MEMORY is
reached
    # single tuple = (state, action, reward, next_state, done)
```

```
def train_long_memory(self):
    if len(self.memory) > BATCH_SIZE:
        mini_sample = random.sample(self.memory, BATCH_SIZE) # list of tuples, no of tuples =
```

```

BATCH_SIZE
    else:
        mini_sample = self.memory

        states, actions, rewards, next_states, dones = zip(*mini_sample)
        self.trainer.train_step(states, actions, rewards, next_states, dones)
        #for state, action, reward, next_state, done in mini_sample:
        #    self.trainer.train_step(state, action, reward, next_state, done)

def train_short_memory(self, state, action, reward, next_state, done):
    self.trainer.train_step(state, action, reward, next_state, done)

def get_action(self, state):
    # random moves: tradeoff exploration / exploitation
    self.epsilon = 80 - self.n_games # more no of games => smaller epsilon => lesser random moves
    final_move = [0,0,0]
    if random.randint(0, 200) < self.epsilon:
        move = random.randint(0, 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item() # item() converts multidimensional matrix(tensor) to
one item
        final_move[move] = 1

    return final_move

def train():
    plot_scores = []
    plot_mean_scores = []
    total_score = 0
    record = 0
    agent = Agent()
    game = SnakeGameAI()
    while True:
        # get old state
        state_old = agent.get_state(game)

        # get move
        final_move = agent.get_action(state_old)

        # perform move and get new state
        reward, done, score = game.play_step(final_move)
        state_new = agent.get_state(game)

        # train short memory
        agent.train_short_memory(state_old, final_move, reward, state_new, done)

        # remember

```

```
agent.remember(state_old, final_move, reward, state_new, done)
```

```
if done:
```

```
    # train long memory, plot result
```

```
    game.reset()
```

```
    agent.n_games += 1
```

```
    agent.train_long_memory()
```

```
if score > record:
```

```
    record = score
```

```
    agent.model.save()
```

```
print('Game', agent.n_games, 'Score', score, 'Record:', record)
```

```
plot_scores.append(score)
```

```
total_score += score
```

```
mean_score = total_score / agent.n_games
```

```
plot_mean_scores.append(mean_score)
```

```
plot(plot_scores, plot_mean_scores)
```

```
if __name__ == '__main__':
```

```
    train()
```



## ChatGPT

This code represents an implementation of a Deep Q-Learning agent to play the classic game of Snake. Let's break down the code step by step:

### 1. Imports:

- The code imports necessary libraries including `torch` for neural network operations, `random` for random number generation, `numpy` for numerical operations, `deque` for implementing a fixed-size queue, and imports from custom modules (`game`, `model`, and `helper`).

### 2. Constants Initialization:

- `MAX_MEMORY`: Maximum capacity of the agent's memory buffer.
- `BATCH_SIZE`: Size of the mini-batch sampled from memory during training.
- `LR`: Learning rate for the neural network optimizer.

### 3. Class Definition: `Agent`:

- Constructor (`__init__`): Initializes the agent with parameters such as number of games played (`n_games`), exploration rate (`epsilon`), discount factor (`gamma`), memory buffer (`memory`), neural network model (`model`), and Q-learning trainer (`trainer`).
- `get_state`: Given the game state, this method generates the input state representation for the neural network. It includes features such as positions of the snake's head and food, along with danger indicators in different directions.

- ``remember``: Stores the experiences (state, action, reward, next\_state, done) in the agent's memory buffer.
- ``train_long_memory``: Performs a training step using experiences sampled from the memory buffer.
- ``train_short_memory``: Performs a training step using a single experience.
- ``get_action``: Chooses an action based on the current state. It balances exploration and exploitation, either by selecting a random action with probability ``epsilon`` or by selecting the action with the highest predicted Q-value.

#### 4. Training Function: ``train``:

- Initializes variables for plotting scores and creates an instance of the ``Agent`` class and the ``SnakeGameAI`` environment.
- Enters a training loop where the agent interacts with the environment:
  - Obtains the current state of the game.
  - Selects an action based on the current state.
  - Performs the selected action and observes the reward and next state.
  - Trains the agent using the observed experience.
  - Updates the game state and repeats until the game is over.
- After each game, the agent updates its exploration rate (``epsilon``), saves the model if a new high score is achieved, prints the game number, score, and record, and plots the scores.

#### 5. Main Execution:

- Calls the ``train`` function when the script is run.

This code implements a Deep Q-Learning agent that learns to play Snake by interacting with the game environment and updating its policy based on the observed rewards and states. The neural network model is trained using experiences stored in a replay memory buffer, enabling more stable and efficient learning.

#### You

explain use of `from 'collections import deque'` , explain in short what is deque and why we use it here.

#### ChatGPT

The ``deque`` (pronounced "deck") in Python is a specialized data structure that stands for "double-ended queue". It provides a way to efficiently append and pop elements from both ends of the queue, which makes it particularly useful for implementing a memory buffer with a fixed maximum size.

In the context of the provided code, ``deque`` is used to create a memory buffer to store experiences encountered by the agent during its interaction with the environment. Here's why it's used:

1. **Efficient Memory Management:** ``deque`` allows us to set a maximum size (``MAX_MEMORY``) for the buffer. When the buffer is full and a new experience is added, it automatically discards the oldest experience, ensuring that the buffer doesn't exceed its predefined size. This is crucial for memory efficiency, especially in scenarios where storing all experiences is not feasible due to memory constraints.
2. **Efficient Access:** ``deque`` provides efficient access to both ends of the buffer. This is important for the agent's training process, as it needs to sample experiences randomly from the buffer during training. With ``deque``, accessing the beginning (oldest experiences) or end (newest experiences) of the buffer has a time complexity of  $O(1)$ , making it suitable for real-time applications like reinforcement learning.

In summary, ``deque`` is used in this code to efficiently manage a fixed-size memory buffer for storing experiences, ensuring both memory efficiency and efficient access to experiences during training.

## You

explain use case of 'from helper import plot'

## ChatGPT

The line ``from helper import plot`` in the provided code imports a specific function or class called ``plot`` from a module named ``helper``. Let's explore the use case of this import statement:

### 1. Visualization:

- The ``plot`` function likely serves a visualization purpose, as suggested by its name. It is commonly used to plot data such as game scores or other performance metrics during training.

### 2. Code Modularity:

- Importing ``plot`` from a separate module (``helper``) enhances code modularity. This means that the plotting functionality is encapsulated in its own module, which makes the main code cleaner and easier to understand. It separates concerns, allowing developers to focus on specific functionalities in different modules.

### 3. Reusability:

- By encapsulating plotting functionalities in a separate module, the ``plot`` function can be reused across multiple projects or even within different parts of the same project. This promotes code reuse and avoids duplication of code, leading to more maintainable and scalable codebases.

### 4. Abstraction:

- Importing `plot` from `helper` abstracts away the implementation details of the plotting functionality. This means that the main code doesn't need to know how the plotting is implemented; it only needs to know how to call the `plot` function with appropriate parameters.

## 5. Encapsulation of Utility Functions:

- The `helper` module likely contains utility functions or helper functions that are used across the project. By organizing these functions into a separate module, it promotes better organization and structure within the codebase.

In summary, importing `plot` from `helper` enables the main code to utilize a specific plotting function while promoting modularity, reusability, abstraction, and encapsulation of utility functions. It helps maintain a clean, organized, and maintainable codebase.

### You

explain this -

```
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done)) # popleft if MAX_MEMORY is
reached
    # single tuple = (state, action, reward, next_state, done)
```

### ChatGPT

The `remember` method is a part of a class, likely associated with a reinforcement learning agent. Let's break down what this method does:

#### 1. Method Signature:

- The method `remember` takes six parameters: `self` (implicitly passed to refer to the instance of the class), `state`, `action`, `reward`, `next_state`, and `done`.

#### 2. Functionality:

- The purpose of the `remember` method is to store the experiences encountered by the agent in its memory buffer. These experiences typically consist of the current state, the action taken by the agent, the reward received, the resulting next state, and whether the episode terminated after this transition (indicated by `done`).

#### 3. Memory Buffer:

- The experiences are stored in a data structure called `memory`, which is presumably a deque (double-ended queue) initialized earlier in the class. Deque is used here because it allows efficient appending of new experiences (`append`) and discarding of the oldest experiences if the maximum memory capacity is reached (`popleft`).

#### 4. Tuple of Experiences:

- The experiences are appended to the `memory` deque as a single tuple. Each tuple represents one transition in the agent's interaction with the environment and contains the current state, action, reward, next state, and termination status (`done`).

## 5. Memory Management:

- A comment in the code suggests that if the maximum memory capacity (`MAX_MEMORY`) is reached, the oldest experiences will be discarded from the left side of the deque (`popleft`). This ensures that the memory buffer does not exceed its predefined size, helping manage memory efficiently, especially in scenarios with limited memory resources.

In summary, the `remember` method is responsible for storing experiences encountered by the reinforcement learning agent during its interactions with the environment in a memory buffer, allowing the agent to learn from past experiences through techniques like experience replay.

### You

explain use of this -

```
def train_long_memory(self):
    if len(self.memory) > BATCH_SIZE:
        mini_sample = random.sample(self.memory, BATCH_SIZE) # list of tuples, no of tuples =
BATCH_SIZE
    else:
        mini_sample = self.memory

    states, actions, rewards, next_states, dones = zip(*mini_sample)
    self.trainer.train_step(states, actions, rewards, next_states, dones)
    #for state, action, reward, next_state, done in mini_sample:
    #    self.trainer.train_step(state, action, reward, next_state, done)
```

### ChatGPT

The `train_long_memory` method is responsible for training the agent using a batch of experiences sampled from its memory buffer. Let's break down the steps involved:

#### 1. Sampling from Memory:

- The method first checks if the size of the memory buffer (`self.memory`) is greater than the batch size (`BATCH_SIZE`). If it is, a random sample of experiences of size `BATCH_SIZE` is selected from the memory buffer using `random.sample`. This sample is stored in `mini_sample`, which is a list of tuples representing individual experiences. If the size of the memory buffer is smaller than the batch size, the entire memory buffer is used as the sample.

#### 2. Unpacking Experiences:

- Once the sample is obtained, it is unpacked into separate lists of states, actions, rewards, next states, and termination statuses (`dones`). This is done using the



``zip(*mini_sample)`` construct, which effectively transposes the list of tuples, resulting in five separate lists.

### 3. Training the Neural Network:

- The ``self.trainer.train_step`` method is then called with the unpacked lists of states, actions, rewards, next states, and termination statuses. This method is responsible for updating the parameters of the neural network (usually a Q-network) based on the experiences provided in the batch. It implements the Q-learning algorithm or a variant thereof, such as Deep Q-Networks (DQN).

### 4. Comments:

- The commented-out ``for`` loop suggests an alternative approach for training, where instead of unpacking the sample, each tuple in the sample is iterated over, and ``train_step`` is called individually for each experience. However, the implementation opts for a more efficient batch training approach using unpacking.

In summary, the ``train_long_memory`` method enables the agent to train its neural network using a batch of experiences sampled from its memory buffer. This batch training approach helps stabilize and accelerate the learning process by leveraging multiple experiences simultaneously.

### You

explain use of 'states, actions, rewards, next\_states, dones = zip(\*mini\_sample)' here

### ChatGPT

Certainly! Let's break down the use of ``states, actions, rewards, next_states, dones = zip(*mini_sample)`` in the provided code:

#### 1. Unpacking:

- ``zip(*mini_sample)`` is a Python idiom used for unpacking a list of tuples. ``mini_sample`` is a list of tuples where each tuple represents an experience consisting of ``(state, action, reward, next_state, done)``. By using ``*mini_sample``, we unpack the list of tuples, effectively transposing it.

#### 2. Result:

- After unpacking, the ``zip`` function groups the elements of each tuple by their respective positions. This means the first element of each tuple forms a new tuple, the second elements form another tuple, and so on.

#### 3. Assignment:

- The unpacked results are then assigned to individual variables: ``states``, ``actions``, ``rewards``, ``next_states``, and ``dones``. Each of these variables becomes a list containing all the states, actions, rewards, next states, and termination statuses, respectively, from the experiences in the ``mini_sample``.

#### 4. Purpose:

- This unpacking step is crucial for preparing the data to be used in the training of the neural network. Neural network training often requires data to be separated into distinct inputs (states), targets (next states), and auxiliary information (actions, rewards, and termination statuses). By unpacking the ``mini_sample``, we organize the data in a format that can be directly fed into the training process, making it easier to handle and manipulate during training.

#### 5. Efficiency:

- Unpacking the ``mini_sample`` using ``zip(*mini_sample)`` is a concise and efficient way to separate the experiences into different lists. It avoids the need for explicit iteration over the list and manual extraction of each element, thereby improving code readability and performance.

In summary, ``states, actions, rewards, next_states, dones = zip(*mini_sample)`` is used to unpack a list of tuples containing experiences into separate lists of states, actions, rewards, next states, and termination statuses, facilitating their use in the training of the neural network.

#### You

explain this in detail -

```
def get_action(self, state):
    # random moves: tradeoff exploration / exploitation
    self.epsilon = 80 - self.n_games # more no of games => smaller epsilon => lesser random moves
    final_move = [0,0,0]
    if random.randint(0, 200) < self.epsilon:
        move = random.randint(0, 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item() # item() converts multidimensional matrix(tensor) to one item
        final_move[move] = 1

    return final_move
```

#### ChatGPT

The ``get_action`` method is a crucial part of a reinforcement learning agent, responsible for selecting an action based on the current state. Let's dissect this method step by step:

##### 1. Epsilon-Greedy Strategy:

- At the beginning of the method, the value of ``epsilon`` is updated based on the number of games played (``self.n_games``). This is a common strategy in reinforcement learning

known as the epsilon-greedy strategy. It balances exploration (choosing random actions) and exploitation (choosing actions based on learned values).

- As the number of games increases, ``epsilon`` decreases, leading to less exploration and more reliance on learned values.

## 2. Initialization:

- ``final_move`` is initialized as a list of zeros with a length of 3. This list represents the possible actions the agent can take. For example, if ``final_move`` is `[1, 0, 0]`, it indicates the agent should move left; `[0, 1, 0]` indicates moving straight, and `[0, 0, 1]` indicates moving right.

## 3. Exploration vs. Exploitation:

- The method randomly selects an action with probability ``epsilon`` and exploits the learned policy with probability `1 - epsilon``.
- If a randomly generated number between 0 and 200 is less than ``epsilon``, the agent explores by selecting a random action index (``move``) between 0 and 2. The selected action is then represented by setting the corresponding element of ``final_move`` to 1.
- If the randomly generated number is greater than or equal to ``epsilon``, the agent exploits its learned policy. It first converts the ``state`` to a tensor (``state0``) and feeds it to the neural network model (``self.model``). The model predicts Q-values for each action, and the action with the highest Q-value is selected (``torch.argmax(prediction).item()``). The corresponding element of ``final_move`` is set to 1 to indicate the chosen action.

## 4. Return:

- Finally, ``final_move`` is returned, representing the action chosen by the agent based on the epsilon-greedy strategy.

In summary, the ``get_action`` method implements the epsilon-greedy strategy to select actions for the agent. It balances exploration and exploitation by randomly selecting actions with a decreasing probability as the number of games played increases, while also exploiting the learned policy to maximize rewards.

## You

explain this in detail -

```
def train():
    plot_scores = []
    plot_mean_scores = []
    total_score = 0
    record = 0
    agent = Agent()
    game = SnakeGameAI()
    while True:
        # get old state
        state_old = agent.get_state(game)

        # get move
```

```

final_move = agent.get_action(state_old)

# perform move and get new state
reward, done, score = game.play_step(final_move)
state_new = agent.get_state(game)

# train short memory
agent.train_short_memory(state_old, final_move, reward, state_new, done)

# remember
agent.remember(state_old, final_move, reward, state_new, done)

if done:
    # train long memory, plot result
    game.reset()
    agent.n_games += 1
    agent.train_long_memory()

    if score > record:
        record = score
        agent.model.save()

    print('Game', agent.n_games, 'Score', score, 'Record:', record)

    plot_scores.append(score)
    total_score += score
    mean_score = total_score / agent.n_games
    plot_mean_scores.append(mean_score)
    plot(plot_scores, plot_mean_scores)

```

## ChatGPT

The `train` function serves as the main training loop for the reinforcement learning agent to learn to play the Snake game. Let's dissect each step of this function:

### 1. Initialization:

- `plot_scores`, `plot_mean_scores`, `total_score`, and `record` are initialized to track and visualize the progress of the training.
- An instance of the `Agent` class (`agent`) and the `SnakeGameAI` environment (`game`) are created.

### 2. Training Loop:

- The training loop runs indefinitely (`while True:`) until manually interrupted or a stopping criterion is met.

### 3. Agent-Environment Interaction:

- `state_old` is obtained by querying the agent for the current state of the game (`agent.get_state(game)`).

- `final_move` is obtained by querying the agent for the action to take based on the current state (`agent.get_action(state_old)`).
- The action (`final_move`) is executed in the game environment (`game.play_step(final_move)`), resulting in a reward, termination status (`done`), and the new score (`score`).
- `state_new` is obtained by querying the agent for the new state of the game after taking the action (`agent.get_state(game)`).

#### 4. Training:

- The agent's short-term memory is updated by providing the old state, action, reward, new state, and termination status (`agent.train_short_memory(...)`).
- The experience tuple (old state, action, reward, new state, termination status) is stored in the agent's long-term memory (`agent.remember(...)`).

#### 5. End of Game Handling:

- If the game is done (i.e., the snake collided with itself or the wall):
  - The game is reset (`game.reset()`).
  - The number of games played (`agent.n_games`) is incremented.
  - The agent's long-term memory is used to train the neural network (`agent.train_long_memory()`).
  - If the current score surpasses the previous record, the record is updated, and the model is saved (`agent.model.save()`).
  - Information about the game, including the game number, score, and record, is printed.
  - The current score is appended to `plot_scores`, and the mean score is updated and appended to `plot_mean_scores`.
  - The `plot` function is called to visualize the scores.

#### 6. Continued Training:

- The training loop continues indefinitely, allowing the agent to accumulate experience and learn from it over time.

In summary, the `train` function orchestrates the training process of the reinforcement learning agent, including interaction with the game environment, updating its memory, training its neural network, and visualization of training progress.

### You

explain this code in detail -

```
import matplotlib.pyplot as plt
from IPython import display
```

```
plt.ion()
```

```
def plot(scores, mean_scores):
    display.clear_output(wait=True)
    display.display(plt.gcf())
```

```
plt.clf()
plt.title('Training...')
plt.xlabel('Number of Games')
plt.ylabel('Score')
plt.plot(scores)
plt.plot(mean_scores)
plt.ylim(ymin=0)
plt.text(len(scores)-1, scores[-1], str(scores[-1]))
plt.text(len(mean_scores)-1, mean_scores[-1], str(mean_scores[-1]))
plt.show(block=False)
plt.pause(0.1)
```



## ChatGPT

This code defines a function `plot` responsible for visualizing the training progress of a reinforcement learning agent playing the Snake game. Let's go through each part of the code to understand its functionality:

### 1. Importing Libraries:

- `matplotlib.pyplot` is imported as `plt` for plotting functionalities.
- `IPython.display` is imported for handling display functions within the IPython environment.

### 2. Interactive Mode:

- `plt.ion()` is called to turn on interactive mode in Matplotlib. This mode allows for real-time updates to the plot.

### 3. Plot Function (`plot`):

- This function takes two arguments: `scores` (a list of scores achieved in individual games) and `mean_scores` (a list of mean scores calculated over a certain number of games).
- `display.clear_output(wait=True)` clears the previous output in the IPython display, allowing for updating the plot in-place.
- `display.display(plt.gcf())` displays the current figure (plot) within the IPython environment.
- `plt.clf()` clears the current figure to prepare it for updating.
- `plt.title('Training...')`, `plt.xlabel('Number of Games')`, and `plt.ylabel('Score')` set the title and labels for the plot.
- `plt.plot(scores)` and `plt.plot(mean_scores)` plot the `scores` and `mean_scores` on the same plot.
- `plt.ylim(ymin=0)` sets the minimum value of the y-axis to 0 to ensure all scores are visible.
- `plt.text(len(scores)-1, scores[-1], str(scores[-1]))` and `plt.text(len(mean_scores)-1, mean_scores[-1], str(mean_scores[-1]))` annotate the last data points of `scores` and `mean_scores` respectively with their corresponding values.
- `plt.show(block=False)` displays the plot without blocking the execution of the code.
- `plt.pause(0.1)` pauses the code execution for a short duration (0.1 seconds), allowing the plot to be updated and displayed.

#### 4. Real-Time Visualization:

- By utilizing `display.clear_output` and `plt.pause`, the function enables real-time visualization of training progress within the IPython environment. After each update, the plot is cleared and re-displayed with the latest data.

In summary, the `plot` function provides a mechanism for real-time visualization of training progress, allowing users to monitor the agent's performance as it learns to play the Snake game. It dynamically updates the plot with new scores and mean scores as the training progresses.

#### You

what exactly does these lines do -

```
display.display(plt.gcf())
plt.clf()
```

#### ChatGPT

The lines `display.display(plt.gcf())` and `plt.clf()` are responsible for clearing and updating the current plot displayed in an interactive environment, such as Jupyter Notebook or IPython. Let's break down their functionality:

##### 1. `display.display(plt.gcf())`:

- `plt.gcf()` stands for "get current figure". This function returns a reference to the current figure being displayed by Matplotlib.
- `display.display()` is a function from the `IPython.display` module. It is used to display objects within the IPython environment.

##### Functionality:

- `display.display(plt.gcf())` is used to update the display with the current Matplotlib figure. It ensures that any changes made to the figure are immediately reflected in the IPython environment.

##### 2. `plt.clf()`:

- `plt.clf()` stands for "clear figure". This function clears the current figure, making it ready for a new plot to be drawn.

##### Functionality:

- `plt.clf()` is called after displaying the current figure to clear it. This ensures that the previous plot is removed before drawing the updated plot. It prevents overlapping of plots and ensures that only the latest plot is displayed.

In summary, `display.display(plt.gcf())` updates the display with the current Matplotlib figure, and `plt.clf()` clears the current figure to prepare it for drawing a new plot. Together, they

facilitate real-time updating of plots in interactive environments, allowing for dynamic visualization of data.

### You

what exactly does these lines do -

```
plt.text(len(scores)-1, scores[-1], str(scores[-1]))
plt.text(len(mean_scores)-1, mean_scores[-1], str(mean_scores[-1]))
```

### ChatGPT

These lines are responsible for adding text annotations to the plot at specific coordinates. Let's break down their functionality:

1. `plt.text(len(scores)-1, scores[-1], str(scores[-1]))`:`
  - This line adds a text annotation to the plot.
  - `len(scores)-1` calculates the x-coordinate of the annotation. It is set to len(scores)-1` to place the annotation at the last index of the scores` list.`
  - `scores[-1]` retrieves the y-coordinate of the annotation. It represents the last score in the scores` list.`
  - `str(scores[-1])` converts the last score to a string for display.`
2. `plt.text(len(mean_scores)-1, mean_scores[-1], str(mean_scores[-1]))`:`
  - This line adds another text annotation to the plot.
  - Similar to the previous line, `len(mean_scores)-1` calculates the x-coordinate of the annotation, and mean_scores[-1]` retrieves the y-coordinate of the annotation, representing the last mean score in the mean_scores` list.`
  - `str(mean_scores[-1])` converts the last mean score to a string for display.`

### Functionality:

- These lines annotate the plot with the last recorded score and mean score. They are placed at the coordinates determined by the lengths of the `scores` and mean_scores` lists, ensuring that the annotations are displayed at the end of the plot.`
- By displaying the numerical values of the last score and mean score on the plot, these annotations provide additional information to the viewer, allowing them to quickly assess the performance of the training process.

ChatGPT can make mistakes. Consider checking important information.