



Imitation Learning for Aerial Agents

Environment: Unreal Engine 5 w/AirSim

Dimitrios Yfantidis (3938)

Evangelos Spatharis (3949)

Inspiration and First Thoughts (I)

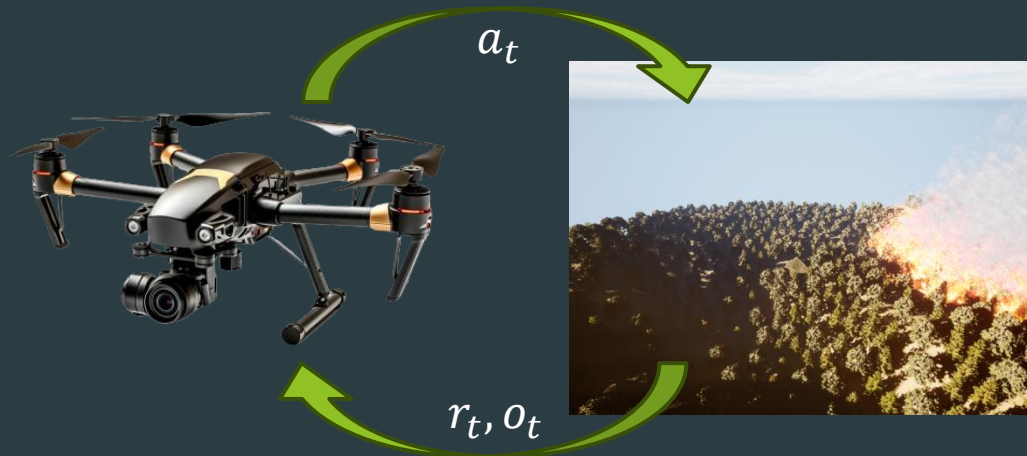
“Why use Unreal Engine?”

- ▶ Derived from our personal interest for **Computer Graphics**.
- ▶ Familiarity with **3D environments** and tools such as game/physics engines.
- ▶ Willingness to contribute to the solution of complex real-world problems (with respect to the corresponding specifications).
- ▶ Computer Simulations: fundamental element of **Reinforcement Learning**.

Inspiration and First Thoughts (I)

“Why use Unreal Engine?”

- ▶ Derived from our personal interest for **Computer Graphics**.
- ▶ Familiarity with **3D environments** and tools such as game/physics engines.
- ▶ Willingness to contribute to the solution of complex real-world problems (with respect to the corresponding specifications).
- ▶ Computer Simulations: fundamental element of **Reinforcement Learning**.



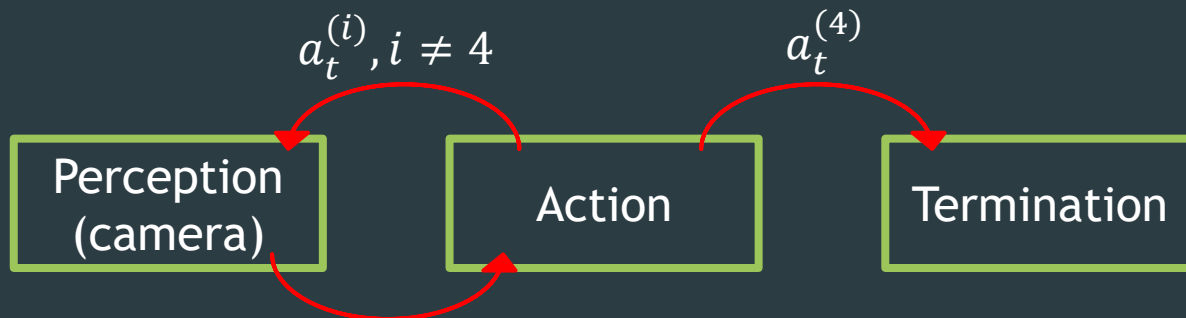
Inspiration and First Thoughts (II)

“Why focus on Unmanned Aerial Vehicles (UAVs)?”

- ▶ Microsoft’s **AirSim** library offers convenient connectivity between Python scripts and 3D environments such as Unreal Engine environments.
- ▶ UAVs are widely used in a plethora of applications in the **emergency planning and disaster management** sector such as a) sensing and locating forest fires and b) search and rescue operation -among others.
- ▶ This specific software pipeline had already **well-established foundations from a previous researcher’s work**.

The Problem (I)

- ▶ The **drone** is placed in an Unreal environment which depicts a natural scenery (mainly a **forest**).
- ▶ In this environment, a small **fire** is placed in a tree which starts spreading to nearby trees. After a specific, finite amount of time, the fire will have levelled either most of the forest or all of it.
- ▶ At the same time, the drone uses its **camera** to **locate** any hints of smoke or fire. Furthermore, it can roam freely within the environment towards any direction.
- ▶ The procedure is completed successfully when the drone both locates the fire and approaches it at a sufficiently close distance.



Δράσεις	Περιγραφή
$a_t^{(0)}$	“Approaching”
$a_t^{(1)}$	“Stepping Back”
$a_t^{(2)}$	“Moving Right”
$a_t^{(3)}$	“Moving Left”
$a_t^{(4)}$	“Finished”

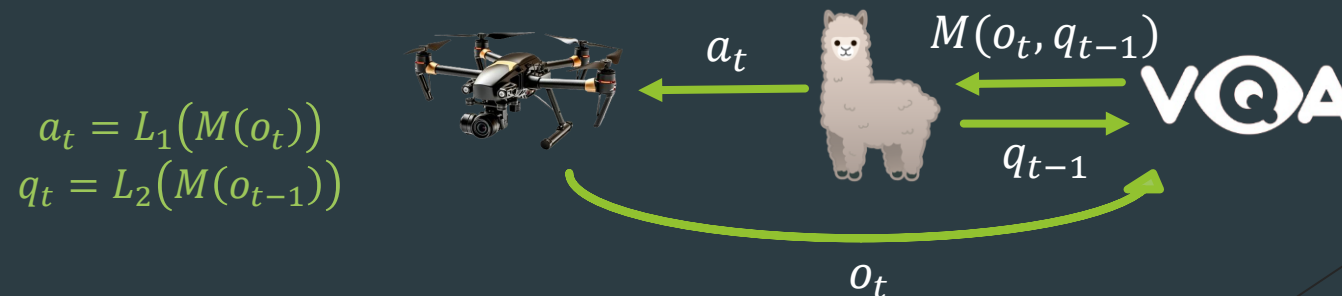
The Problem (II)

- How could such a drone be trained to trace and tack down the fire using only its camera and changing its position (in an alternating manner)?



System of Neural Networks (I)

- ▶ Multimodal Language Model (MLM), $M : O \times T^* \rightarrow T^*$
 - ✓ **Input:** Image $\{0, 1, \dots, 255\}^{W \times H \times 3} \rightarrow$ **Output:** Image description (text)
 - ✓ Responsible for the UAV's **perception**.
 - ✓ A linguistic description is attributed to what its camera observes.
 - ✓ The **LAVIS** Visual Question Answering (VQA) model by Salesforce was used for this task.
- ▶ Large Language Model (LLM), $L_1 : T^* \rightarrow A, L_2 : T^* \rightarrow T^*$
 - ✓ Processes the description derived by the MLM based on which it **decides** the next action.
 - ✓ The LLM's decisions are regarded as **optimal**, and thus the training's **ground truth**.
 - ✓ The models that were used for this were Microsoft's **Phi3-mini** (initially) and Meta's **LLaMa3** (afterwards).



System of Neural Networks (I)

► Multimodal Language Model (MLM), $M : O \times T^* \rightarrow T^*$

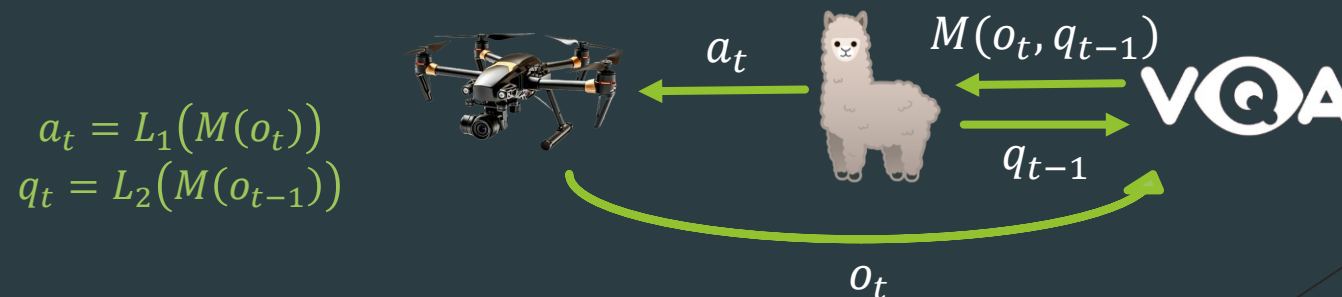
- ✓ **Input:** Image $\{0, 1, \dots, 255\}^{W \times H \times 3} \rightarrow$ **Output:** Image description (text)
- ✓ Responsible for the UAV's **perception**.
- ✓ A linguistic description is attributed to what its camera observes.
- ✓ The **LAVIS** Visual Question Answering (VQA) model by Salesforce was used for this task.

Your controls are:

```
("Move closer, "question about the scene")  
("Move back, "question about the scene") i  
("Move right, "question about the scene")  
("Move left, "question about the scene")  
("I know enough") when you have a full und
```

► Large Language Model (LLM), $L_1 : T^* \rightarrow A$, $L_2 : T^* \rightarrow T^*$

- ✓ Processes the description derived by the MLM based on which it **decides** the next action.
- ✓ The LLM's decisions are regarded as **optimal**, and thus the training's **ground truth**.
- ✓ The models that were used for this were Microsoft's **Phi3-mini** (initially) and Meta's **LLaMa3** (afterwards).



System of Neural Networks (I)

► Multimodal Language Model (MLM), $M : O \times T^* \rightarrow T^*$

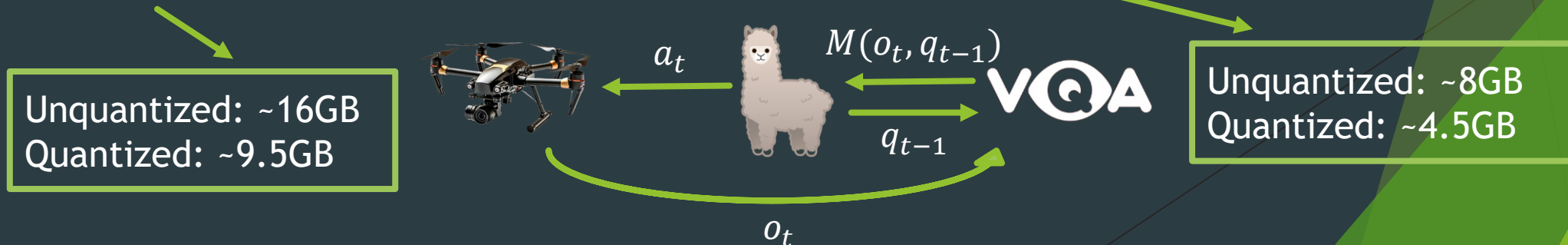
- ✓ **Input:** Image $\{0, 1, \dots, 255\}^{W \times H \times 3} \rightarrow$ **Output:** Image description (text)
- ✓ Responsible for the UAV's **perception**.
- ✓ A linguistic description is attributed to what its camera observes.
- ✓ The **LAVIS** Visual Question Answering (VQA) model by Salesforce was used for this task.

► Large Language Model (LLM), $L_1 : T^* \rightarrow A, L_2 : T^* \rightarrow T^*$

- ✓ Processes the description derived by the MLM based on which it **decides** the next action.
- ✓ The LLM's decisions are regarded as **optimal**, and thus the training's **ground truth**.
- ✓ The models that were used for this were Microsoft's **Phi3-mini** (initially) and Meta's **LLaMa3** (afterwards).

Your controls are:

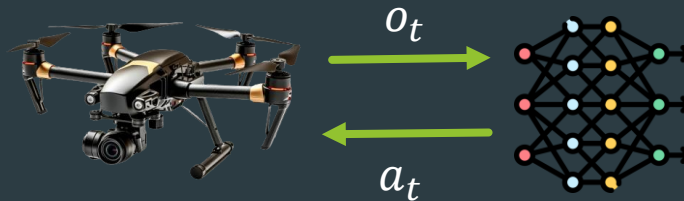
```
("Move closer, "question about the scene")  
("Move back, "question about the scene") i  
("Move right, "question about the scene")  
("Move left, "question about the scene")  
("I know enough") when you have a full und
```



System of Neural Networks (II)

- ▶ This system can locate the fire and translate the drone towards it.
- ▶ Obviously, a drone cannot be equipped with large scale deep neural networks with billions of parameters in real world applications as it is a real-time system with computational time and memory restrains.
- ▶ Utility of a new neural network, $f^* : O \rightarrow A$, of realistic memory size and data inference speed which performs an equivalent functionality:

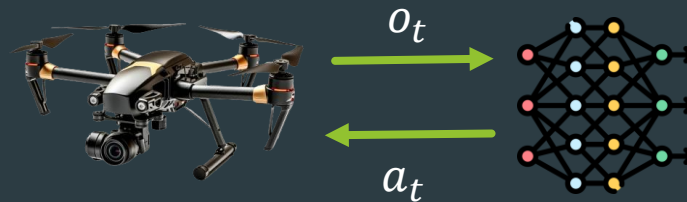
$$f^* \equiv L_1 \circ M$$



System of Neural Networks (II)

- ▶ This system can locate the fire and translate the drone towards it.
- ▶ Obviously, a drone cannot be equipped with large scale deep neural networks with billions of parameters in real world applications as it is a real-time system with computational time and memory restrains.
- ▶ Utility of a new neural network, $f^* : O \rightarrow A$, of realistic memory size and data inference speed which performs an equivalent functionality:

$$f^* \equiv L_1 \circ M$$



*How could we
train such a
model?*

Our Distributed System

- ▶ Remote computer, $2 \times \text{RTX } 4090 \text{ } 24\text{GB} = 48\text{GB VRAM}$:
 - ✓ Large Language Model and VQA $\rightarrow \text{llm_server.py}, \text{llm_client.py}, \text{llm_init.py}, \text{VQAWrapper.py}, \text{vqa_llm.py}$
 - ✓ RL Agent and Deep Q-Network $\rightarrow \text{Agent.py}$
 - ✓ AirSim Server API $\rightarrow \text{AirSimServer.py}$



Our Distributed System

- ▶ Remote computer, 2 × RTX 4090 24GB = 48GB VRAM:
 - ✓ Large Language Model and VQA → *llm_server.py*, *llm_client.py*, *llm_init.py*, *VQAWrapper.py*, *vqallm.py*
 - ✓ RL Agent and Deep Q-Network → *Agent.py*
 - ✓ AirSim Server API → *AirSimServer.py*
- ▶ Local Computer:
 - ✓ AirSim Client API → *AirSimClient.py*
 - ✓ Unreal Engine 5 environment



Deep Reinforcement Learning (I)

- ▶ Representation of the UAV as an RL Agent.
- ▶ The reward, r_t , is equal to -1 if $f(o_t) \neq L_1 \left(M \left(o_t, L_2 \left(M(o_{t-1}) \right) \right) \right)$, else 0.
- ▶ Thus, our aerial agent is trained from the knowledge derived from the multimodal and the large language models.
- ▶ However, the learning process is not supervised as this would not be compatible with the core idea behind reinforcement learning.
- ▶ The agent's training method is the classic Deep Q-Network (DQN) algorithm with slight variations.

Deep Reinforcement Learning (II)

- ▶ Constructing the agent and loading it on the GPU.
- ▶ Training for 25 episodes with training evaluation afterwards.

```
drone = UnrealDroneAgent(  
    env=AirSimEnv(),  
    device=torch.device("cuda:0")  
)  
EXEC_INFO = { 'num_checkpoints' : 1, 'num_episodes' : 25 }  
  
for checkpoint in range(EXEC_INFO['num_checkpoints']):  
    # Enter training mode  
    drone.log(">>>>> Now entering TRAINING mode <<<<<", color, Colors.purple)  
    drone.train(num_episodes=EXEC_INFO['num_episodes'])  
    # Reset the LLM's context window  
    end_train_msg = drone.env.teacher.llm.ask("!CHAT_RESET")  
    drone.log(end_train_msg, color, Colors.orange)  
    # Enter evaluation mode  
    drone.log(">>>>> Now entering EVALUATION mode <<<<<", color, Colors.purple)  
    drone.test()  
    # Reset the LLM's context window  
    end_test_msg = drone.env.teacher.llm.ask("!CHAT_RESET")  
    drone.log(end_test_msg, color, Colors.orange)  
# Shut down LLM  
drone.env.teacher.llm.disconnect()
```

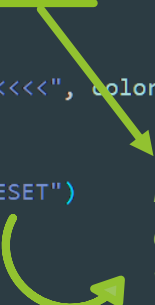
Deep Reinforcement Learning (II)

- ▶ Constructing the agent and loading it on the GPU.
- ▶ Training for 25 episodes with training evaluation afterwards.

```
drone = UnrealDroneAgent(  
    env=AirSimEnv(),  
    device=torch.device("cuda:0")  
)  
EXEC_INFO = { 'num_checkpoints' : 1, 'num_episodes' : 25 }
```



```
for checkpoint in range(EXEC_INFO['num_checkpoints']):  
    # Enter training mode  
    drone.log(">>>>> Now entering TRAINING mode <<<<<", color, Colors.purple)  
    drone.train(num_episodes=EXEC_INFO['num_episodes'])  
    # Reset the LLM's context window  
    end_train_msg = drone.env.teacher.llm.ask("!CHAT_RESET")  
    drone.log(end_train_msg, color, Colors.orange)  
    # Enter evaluation mode  
    drone.log(">>>>> Now entering EVALUATION mode <<<<<", color, Colors.purple)  
    drone.test()  
    # Reset the LLM's context window  
    end_test_msg = drone.env.teacher.llm.ask("!CHAT_RESET")  
    drone.log(end_test_msg, color, Colors.orange)  
# Shut down LLM  
drone.env.teacher.llm.disconnect()
```



Restarting the LLM by deleting its context window.

Deep Reinforcement Learning (III)

- ▶ The environment computes a new state.
- ▶ For simplicity we assume that $s_t \equiv o_t$, i.e. the camera input describes the state of the world perfectly even if the environment is not fully observable ([read more](#)).

```
def train(self, num_episodes: int):
    for i_episode in range(num_episodes):
        # Initialize the environment and get its state
        state, info = self.env.reset()
        state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
        for t in count():
            action = drone.select_action(state)
            screenshot, reward, terminated, truncated, _ = self.env.step(action.item())
            reward = torch.tensor([reward], device=drone.device)
            done = terminated or truncated
            next_state = None if terminated else torch.tensor(screenshot, dtype=torch.float32, device=self.device).unsqueeze(0)
            # Store the transition in memory
            self.memory.push(state, action, next_state, reward)
            # Move to the next state
            state = next_state
            # Perform one step of the optimization (on the policy network)
            self.optimize_model()
            # Soft update of the target network's weights
            #  $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ 
            target_net_state_dict = self.target_net.state_dict()
            policy_net_state_dict = self.policy_net.state_dict()
            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[key] * self.TAU + target_net_state_dict[key] * (1-self.TAU)
            self.target_net.load_state_dict(target_net_state_dict)
            if done:
                # self.episode_durations.append(t + 1)
                # self.plot_durations()
                break
```

Deep Reinforcement Learning (IV)

- Choosing an action $f(o_t) = a_t^{(i)} \in A$

```
def train(self, num_episodes: int):
    for i_episode in range(num_episodes):
        # Initialize the environment and get its state
        state, info = self.env.reset()
        state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
        for t in count():
            action = drone.select_action(state)
            screenshot, reward, terminated, truncated, _ = self.env.step(action.item())
            reward = torch.tensor([reward], device=drone.device)
            done = terminated or truncated
            next_state = None if terminated else torch.tensor(screenshot, dtype=torch.float32, device=self.device).unsqueeze(0)
            # Store the transition in memory
            self.memory.push(state, action, next_state, reward)
            # Move to the next state
            state = next_state
            # Perform one step of the optimization (on the policy network)
            self.optimize_model()
            # Soft update of the target network's weights
            #  $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ 
            target_net_state_dict = self.target_net.state_dict()
            policy_net_state_dict = self.policy_net.state_dict()
            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[key] * self.TAU + target_net_state_dict[key] * (1-self.TAU)
            self.target_net.load_state_dict(target_net_state_dict)
            if done:
                # self.episode_durations.append(t + 1)
                # self.plot_durations()
                break
```

Deep Reinforcement Learning (V)

- With probability $1 - \varepsilon_t$ the agent decides on its own the (what it considers to be) the optimal solution based on its policy network.

```
def select_action(self, state: torch.Tensor):
    # Value between [0, 1)
    sample = random.random()
    eps_threshold = self.EPS_END + (self.EPS_START - self.EPS_END) * \
        math.exp(-1. * self.steps_done / self.EPS_DECAY)
    self.steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return the largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return self.policy_net(state).max(1).indices.view(1, 1)
    else:
        return torch.tensor([[self.env.teacher.ask(False)[1]]], device=self.device, dtype=torch.long)
```

Deep Reinforcement Learning (VI)

- ▶ With probability $1 - \varepsilon_t$ the agent decides on its own the (what it considers to be) the optimal solution based on its policy network.
- ▶ Otherwise, the agent leaves it on the large models to decide with probability ε_t .

```
def select_action(self, state: torch.Tensor):
    # Value between [0, 1)
    sample = random.random()
    eps_threshold = self.EPS_END + (self.EPS_START - self.EPS_END) * \
        math.exp(-1. * self.steps_done / self.EPS_DECAY)
    self.steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return the largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return self.policy_net(state).max(1).indices.view(1, 1)
    else:
        return torch.tensor([[self.env.teacher.ask(False)[1]]], device=self.device, dtype=torch.long)
```

↓
*Avoids the early change
of state.*

Deep Reinforcement Learning (VII)

- The environment transitions to a new state, which it returns along with the reward/punishment, r_t .

```
def train(self, num_episodes: int):
    for i_episode in range(num_episodes):
        # Initialize the environment and get its state
        state, info = self.env.reset()
        state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
        for t in count():
            action = drone.select_action(state)
            screenshot, reward, terminated, truncated, _ = self.env.step(action.item())
            reward = torch.tensor([reward], device=drone.device)
            done = terminated or truncated
            next_state = None if terminated else torch.tensor(screenshot, dtype=torch.float32, device=self.device).unsqueeze(0)
            # Store the transition in memory
            self.memory.push(state, action, next_state, reward)
            # Move to the next state
            state = next_state
            # Perform one step of the optimization (on the policy network)
            self.optimize_model()
            # Soft update of the target network's weights
            #  $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ 
            target_net_state_dict = self.target_net.state_dict()
            policy_net_state_dict = self.policy_net.state_dict()
            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[key] * self.TAU + target_net_state_dict[key] * (1-self.TAU)
            self.target_net.load_state_dict(target_net_state_dict)
            if done:
                # self.episode_durations.append(t + 1)
                # self.plot_durations()
                break
```

Deep Reinforcement Learning (VIII)

- This new experience is stored in the Replay Buffer for future training.

```
def train(self, num_episodes: int):
    for i_episode in range(num_episodes):
        # Initialize the environment and get its state
        state, info = self.env.reset()
        state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
        for t in count():
            action = drone.select_action(state)
            screenshot, reward, terminated, truncated, _ = self.env.step(action.item())
            reward = torch.tensor([reward], device=drone.device)
            done = terminated or truncated
            next_state = None if terminated else torch.tensor(screenshot, dtype=torch.float32, device=self.device).unsqueeze(0)
            # Store the transition in memory
            self.memory.push(state, action, next_state, reward)
            # Move to the next state
            state = next_state
            # Perform one step of the optimization (on the policy network)
            self.optimize_model()
            # Soft update of the target network's weights
            #  $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ 
            target_net_state_dict = self.target_net.state_dict()
            policy_net_state_dict = self.policy_net.state_dict()
            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[key] * self.TAU + target_net_state_dict[key] * (1-self.TAU)
            self.target_net.load_state_dict(target_net_state_dict)
            if done:
                # self.episode_durations.append(t + 1)
                # self.plot_durations()
                break
```

Deep Reinforcement Learning (IX)

- Train the policy network.

```
def train(self, num_episodes: int):
    for i_episode in range(num_episodes):
        # Initialize the environment and get its state
        state, info = self.env.reset()
        state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
        for t in count():
            action = drone.select_action(state)
            screenshot, reward, terminated, truncated, _ = self.env.step(action.item())
            reward = torch.tensor([reward], device=drone.device)
            done = terminated or truncated
            next_state = None if terminated else torch.tensor(screenshot, dtype=torch.float32, device=self.device).unsqueeze(0)
            # Store the transition in memory
            self.memory.push(state, action, next_state, reward)
            # Move to the next state
            state = next_state
            # Perform one step of the optimization (on the policy network)
            self.optimize_model()
            # Soft update of the target network's weights
            #  $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ 
            target_net_state_dict = self.target_net.state_dict()
            policy_net_state_dict = self.policy_net.state_dict()
            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[key] * self.TAU + target_net_state_dict[key] * (1-self.TAU)
            self.target_net.load_state_dict(target_net_state_dict)
            if done:
                # self.episode_durations.append(t + 1)
                # self.plot_durations()
                break
```

Deep Reinforcement Learning (IX)

► Train the policy network.

```
def optimize_model(self) -> None:
    if len(self.memory) < self.BATCH_SIZE:
        return
    transitions = self.memory.sample(self.BATCH_SIZE)
    # Transpose the batch. This converts batch-array of Transitions to Transition of batch-arrays.
    batch = UnrealDroneAgent.Transition(*zip(*transitions))
    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                             batch.next_state)), device=self.device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                       if s is not None])

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the columns of actions taken.
    # These are the actions which would've been taken for each batch state according to policy_net
    state_action_values = self.policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1).values
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(self.BATCH_SIZE, device=self.device)
    with torch.no_grad():
        next_state_values[non_final_mask] = self.target_net(non_final_next_states).max(1).values
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * self.GAMMA) + reward_batch

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    self.optimizer.zero_grad()
    loss.backward()
    # In-place gradient clipping
    torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 100)
    self.optimizer.step()
```


Deep Reinforcement Learning (X)

- ▶ Train the policy network.
- ▶ Train the target network.

```
def train(self, num_episodes: int):
    for i_episode in range(num_episodes):
        # Initialize the environment and get its state
        state, info = self.env.reset()
        state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
        for t in count():
            action = drone.select_action(state)
            screenshot, reward, terminated, truncated, _ = self.env.step(action.item())
            reward = torch.tensor([reward], device=drone.device)
            done = terminated or truncated
            next_state = None if terminated else torch.tensor(screenshot, dtype=torch.float32, device=self.device).unsqueeze(0)
            # Store the transition in memory
            self.memory.push(state, action, next_state, reward)
            # Move to the next state
            state = next_state
            # Perform one step of the optimization (on the policy network)
            self.optimize_model()
            # Soft update of the target network's weights
            #  $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ 
            target_net_state_dict = self.target_net.state_dict()
            policy_net_state_dict = self.policy_net.state_dict()
            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[key] * self.TAU + target_net_state_dict[key] * (1-self.TAU)
            self.target_net.load_state_dict(target_net_state_dict)
            if done:
                # self.episode_durations.append(t + 1)
                # self.plot_durations()
                break
```

Experiments and Results (I)

[21:02:43] vqa said :

Answer: Answer: ['a blurry image of the sky over some cattle.']

Caption: [Caption: [['a blurry image of the sky over some cattle']] 0.9]

[21:02:44] llm answer : [21:02:44] "Move closer, "Are there any people or vehicles in the scene?"



[21:02:44] Agent's action: Move closer

Experiments and Results (II)

[21:02:46] vqa said :

Answer: Answer: ['closer,']

Caption: [Caption: [['a camera - view photograph of a desert area with a mountain distant']] 0.9]

[21:02:47] llm answer : [21:02:47] "Move closer, "What is the terrain like in this desert area?"



[21:03:04] Agent's action: Move closer

Experiments and Results (III)

[21:03:06] vqa said :

Answer: Answer: ['closer,']

Caption: [Caption: [['a large cluster of large red lights in a video game']] 0.9]

[21:03:07] llm answer : [21:03:07] "Move back, "Are there any objects or structures in the desert area that could be related to the red lights?"



[21:03:22] Agent's action: Move right

Experiments and Results (IV)

[21:03:21] vqa said :

Answer: Answer: ['no']

Caption: [Caption: [['a image in forest blazing on fires fireworks fire fire, oil fire fire fire fire']] 0.9]

[21:03:22] llm answer : [21:03:22] "Move right, "Is there a fire or flames in the desert area?"



[21:03:41] Agent's action: Move right

Experiments and Results (V)

[21:03:25] vqa said : Answer:

Answer: ['right,']

Caption: [Caption: [['fire in fire explosion bonfire fire red fire fire fire flames the a this fire fire']] 0.9]

[21:03:26] llm answer : [21:03:26] "Move right, "Is there anyone or anything near the fire that could be in danger?"



[21:04:00] Agent's action: Move left

Experiments and Results (VI)

[21:03:40] vqa said :

Answer: Answer: ['move right,']

Caption: [Caption: [['a fire is raging with flames on fire']] 0.9]

[21:03:41] llm answer : [21:03:41] "Move right, "Are there any people or animals near the fire that could be hurt?"



[21:04:18] Agent's action: I know enough

[21:05:03] Chat has been reset at the end of episode #0

Experiments and Results (VII)

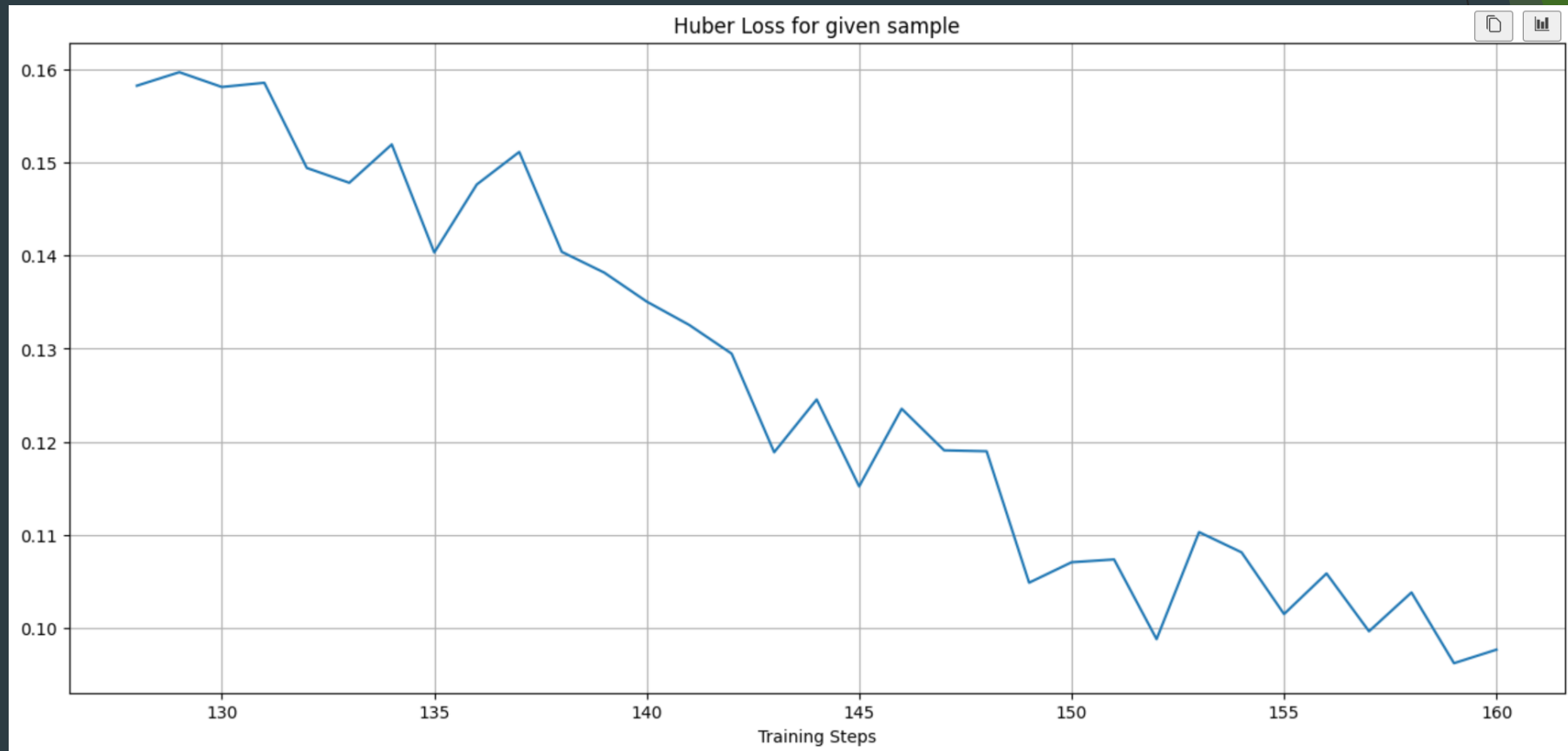
- The next episode starts and the process is repeated.



```
[21:02:38] >>>>> Now entering TRAINING mode <<<<<<
[21:02:38] reset train start
[21:02:40] reset train end
[21:02:44] Agent's action: Move closer
[21:03:04] Agent's action: Move closer
[21:03:22] Agent's action: Move right
[21:03:41] Agent's action: Move right
[21:04:00] Agent's action: Move left
[21:04:18] Agent's action: I know enough
[21:05:03] Chat has been reset at the end of episode #0
[21:05:03] reset train start
[21:05:05] reset train end
[21:05:08] Agent's action: Move closer
[21:05:23] Agent's action: Move closer
[21:05:41] Agent's action: Move back
[21:05:59] Agent's action: Move left
[21:06:22] Agent's action: Move right
[21:06:45] Agent's action: I know enough
[21:07:31] Chat has been reset at the end of episode #1
[21:07:31] reset train start
[21:07:33] reset train end
[21:07:36] Agent's action: Move closer
[21:07:59] Agent's action: Move right
[21:08:17] Agent's action: Move left
[21:08:32] Agent's action: Move closer
[21:08:52] Agent's action: Move closer
[21:09:09] Agent's action: I know enough
[21:09:54] Chat has been reset at the end of episode #2
[21:09:54] Chat has been reset
[21:09:54] >>>>> Now entering EVALUATION mode <<<<<<
```


Experiments and Results (VIII)

- In the following graph we can see the development of the loss function with respect to time (Huber Loss). It is computed after each renewal step.



Suggestions for Future Research

- ▶ Further **quantization** of the models, or at least the LLaMa3 one's.
- ▶ Transitioning to more powerful RL algorithms of higher accuracy such as **PPO**.
- ▶ Improvement of the **reward mechanism** (reward shaping).
- ▶ Add **LSTM** neurons to integrate sequential patterns as this specific problem is based **sequential data**.
- ▶ **Fine-tuning** the hyperparameters of the Teacher και Agent instances.
- ▶ Appropriate increasing of the action space (e.g. $\pm 90^\circ$ rotation command).
- ▶ Replacement of Salesforce LAVIS with **Phi3-Vision**.
- ▶ This project is **distributed** to different computers and different conda environments. The accumulation of the entire project's code in the same machine is suggested, as well as the improvement of the **interoperability** between the different local runtime environments.

Suggestions for Future Research (I)

- ▶ Further **quantization** of the models, or at least the LLaMa3 one's.
 - ✓ The models take up a **sizeable part of VRAM**.
 - ✓ Explicit cleaning of the CUDA cache memory is considered, but the models' trend for **increasing their size** is **ever-growing**.
 - ✓ This obstructs the execution of the program for an increased time period, e.g. **10000 episodes**.
 - ✓ This is, however, the realistic time period after which the agent will, theoretically, start to exhibit increase in its performance.

```
[21:05:03] GPU usage
[21:05:03] > cuda:0 usage: 11720/24564 MiB
[21:05:03] > cuda:1 usage: 6446/24564 MiB
[21:05:07] Context Window: Discarding interaction
[21:05:07] [('127.0.0.1', 45844)] prompted with a message of 127 characters
[21:05:07] Prompt type: <class 'str'>
[21:05:07] > Prompt size: 2893
[21:05:07] > Pipeline size: 48
[21:05:08] Prompt processed after 0.742 seconds)
[21:05:08] GPU usage
[21:05:08] > cuda:0 usage: 12078/24564 MiB
[21:05:08] > cuda:1 usage: 6990/24564 MiB
[21:05:10] Context Window: Saving interaction
[21:05:10] [('127.0.0.1', 45844)] prompted with a message of 111 characters
[21:05:10] Prompt type: <class 'str'>
[21:05:10] > Prompt size: 3056
[21:05:10] > Pipeline size: 48
[21:05:11] Prompt processed after 0.618 seconds)
[21:05:11] GPU usage
[21:05:11] > cuda:0 usage: 12154/24564 MiB
[21:05:11] > cuda:1 usage: 7020/24564 MiB
[21:05:25] Context Window: Saving interaction
[21:05:25] [('127.0.0.1', 45844)] prompted with a message of 143 characters
[21:05:25] Prompt type: <class 'str'>
[21:05:25] > Prompt size: 3346
[21:05:25] > Pipeline size: 48
[21:05:26] Prompt processed after 0.803 seconds)
[21:05:26] GPU usage
[21:05:26] > cuda:0 usage: 12224/24564 MiB
[21:05:26] > cuda:1 usage: 7094/24564 MiB
[21:05:40] Context Window: Discarding interaction
[21:05:40] [('127.0.0.1', 45844)] prompted with a message of 127 characters
```

Suggestions for Future Research (II)

- Transitioning to more powerful RL algorithms of higher accuracy such as PPO.
 - ✓ Avoids the algorithm's hypersensitivity to changes in hyperparameters → **Stability**.
 - ✓ Online learning → Avoids storing past experiences in the Replay Buffer → **Increased efficiency in space and training time**.
 - ✓ Features prior knowledge of the rewards → **Increased performance**.

Algorithm 1 PPO, Actor-Critic Style

```
for iteration=1,2,... do  
  for actor=1,2,...,N do  
    Run policy  $\pi_{\theta_{\text{old}}}$  in environment for  $T$  timesteps  
    Compute advantage estimates  $\hat{A}_1, \dots, \hat{A}_T$   
  end for  
  Optimize surrogate  $L$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$   
   $\theta_{\text{old}} \leftarrow \theta$   
end for
```

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right]$$

$$r_t(\theta) = \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)},$$

Suggestions for Future Research (III)

- ▶ Improvement of the **reward mechanism** (reward shaping).
 - ✓ The current reward mechanism is probably too simplistic for the task at hand.
 - ✓ Not enough attention has been paid to improving its implementation as, so far, it is considered a detail and not an essential part of the logic of the wider project.
 - ✓ In the future, however, the improvement of reward shaping is a function of vital importance for the quality of the results and the speed of convergence of the algorithm.

Suggestions for Future Research (IV)

- ▶ Add **LSTM** neurons to integrate sequential patterns as this specific problem is based **sequential data**.
 - ✓ As mentioned before, **the environment is partially observable** and not fully observable, so the simplification of the form $s_t = f(o_t)$ applied is not allowed.
 - ✓ The agent must pay attention to **the path he is following**, not just the given snapshot.
 - ✓ Usage of recurrent architecture → The state is now defined as:

$$s_t = f(o_1, o_2, \dots, o_t)$$

similar to the case of the **A3C Labyrinth** problem.



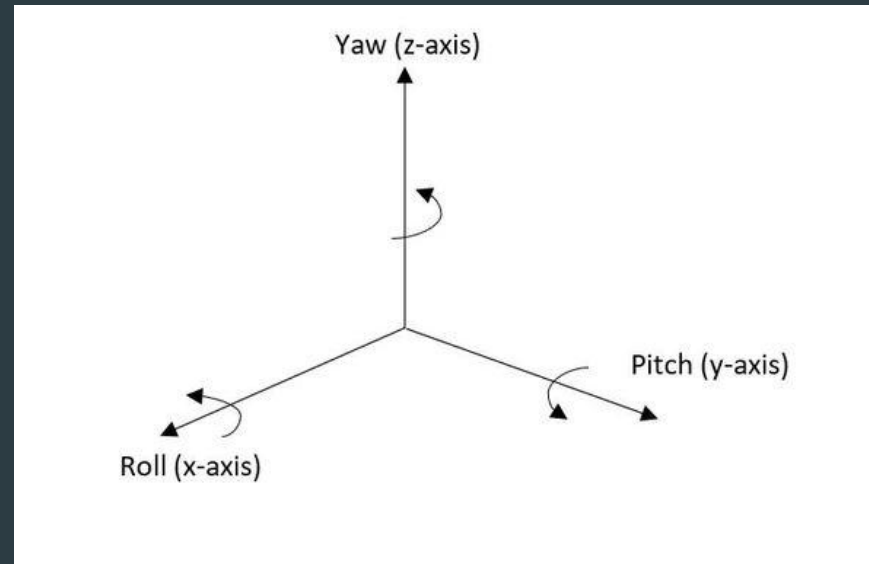
Suggestions for Future Research (V)

- Fine-tuning the hyperparameters of the Teacher και Agent instances.
 - ✓ The default values are:

```
batch_size: int = 128,  
gamma: float = 0.99,  
eps_start: float = 0.9,  
eps_decay: float = 1000.0,  
eps_end: float = 0.05,  
tau: float = 5e-3,  
learning_rate: float = 1e-4,  
replay_buffer_size: int = 10000,
```

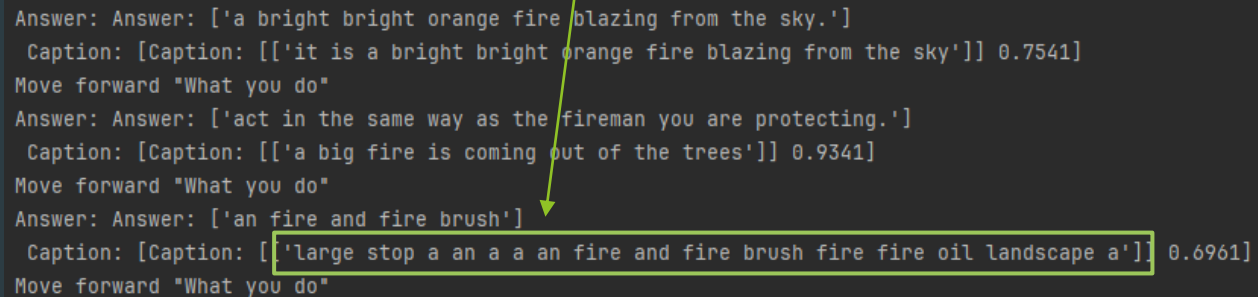
Suggestions for Future Research (VI)

- ▶ Appropriate increasing of the action space (e.g. $\pm 90^\circ$ rotation command).
 - ✓ At the moment, the drone only features actions for changing its position but not its orientation.
 - ✓ The existence of an option to change angles (mainly **yaw**) would greatly increase the agent's capabilities.



Suggestions for Future Research (VII)

- Replacement of Salesforce LAVIS with **Phi3-Vision**.
 - ✓ In general, LAVIS's way of speaking is a bit strange (unprocessed, raw language).
 - ✓ In its answers it may repeat words or simply ramble nonsense.



```
Answer: Answer: ['a bright bright orange fire blazing from the sky.']  
Caption: [Caption: [['it is a bright bright orange fire blazing from the sky']] 0.7541]  
Move forward "What you do"  
Answer: Answer: ['act in the same way as the fireman you are protecting.']  
Caption: [Caption: [['a big fire is coming out of the trees']] 0.9341]  
Move forward "What you do"  
Answer: Answer: ['an fire and fire brush']  
Caption: [Caption: [['large stop a an a a an fire and fire brush fire fire oil landscape a']] 0.6961]  
Move forward "What you do"
```

Suggestions for Future Research (VII)

- Replacement of Salesforce LAVIS with **Phi3-Vision**.
 - ✓ In general, LAVIS's way of speaking is a bit strange (unprocessed, raw language).
 - ✓ In its answers it may repeat words or simply ramble nonsense.
 - ✓ There is even a special directive in LLM's system prompt to ignore nonsensical answers (presumably because it is a common phenomenon).
 - ✓ A lightweight and possibly more efficient model could be the **multimodal version of Microsoft's Phi3**.

```
Answer: Answer: ['a bright bright orange fire blazing from the sky.']
Caption: [Caption: [['it is a bright bright orange fire blazing from the sky']] 0.7541]
Move forward "What you do"
Answer: Answer: ['act in the same way as the fireman you are protecting.']
Caption: [Caption: [['a big fire is coming out of the trees']] 0.9341]
Move forward "What you do"
Answer: Answer: ['an fire and fire brush']
Caption: [Caption: [['large stop a an a a an fire and fire brush fire fire oil landscape a']] 0.6961]
Move forward "What you do"
```

Suggestions for Future Research (VIII)

- ▶ This project is **distributed** to different computers and different conda environments. The accumulation of the entire project's code in the same machine is suggested, as well as the improvement of the **interoperability** between the different local runtime environments.
 - ✓ The approach of implementing the project on different machines and environments leads to increased **overhead**.
 - ✓ Communication via internet → **latency**, program **instability**.
 - ✓ Use of unorthodox communication programming techniques e.g. "scp" commands (disk communication).
 - ✓ This is because the remote machine's OS (Linux) does not support AirSim as it does not feature a headless mode.



Thank You!

A project by:

Dimitrios Yfantidis (3938)

Evangelos Spatharis (3949)