Initializing

In [ ]:

```
import wandb
import gymnasium as gym
import numpy as np

# Versions
print("Gym version: ", gym.__version__)
print("Numpy version: ", np.__version__)

# Hyperparameters
learning_rate = 0.1  # Learning rate
boltzmann = False  # Whether to use Boltzmann exploration or not

# Exploration parameters, based on Epsilon-greedy algorithm
gamma = 0.99  # Discount factor
epsilon = 1.0  # Exploration rate
epsilon_decay = 0.0001  # Decay of epsilon after each episode
epsilon_min = 0.01  # Minimum exploration rate

# Boltzmann exploration parameters
temperature = 1.0  # Temperature parameter
temperature_decay = 0.995  # Decay rate for temperature

episodes = 10000  # Number of episodes to run
is_slippery = True  # Slippery environment

# Creating custom map for a bigger 'challenge'
custom_map = [
    "SFFFFFFF",
    "FFFFFFFF",
    "FFFHFFFF",
    "FFFFFFFF",
    "FFFHFFFF",
    "FHHFFFHF",
    "FHFFHFHF",
    "FFFHFFFG"
]


# Create environment, using custom map
env = gym.make("FrozenLake-v1", is_slippery=is_slippery, desc=custom_map)

# Initialize Q-table
q_table = np.zeros([env.observation_space.n, env.action_space.n])

# Initialize wandb for logging
wandb.init(
    project="RL-FrozenLake"
    )
```

```
Gym version:  0.26.3
Numpy version:  1.20.0
```

Tracking run with wandb version 0.15.12

Run data is saved locally in `c:\Users\jules\git\S7-RL\wandb\run-20231031_223558-rrycsbdx`

Syncing run **mystical-spider-21** to Weights & Biases (docs)

View project at https://wandb.ai/jjuless/RL-FrozenLake

View run at https://wandb.ai/jjuless/RL-FrozenLake/runs/rrycsbdx

Out[ ]: Display W&B run

In [ ]:

```
# custom reward, if goal is reached return 1, taking a step gives -0.01 and falling
def custom_reward(state, reward, done):
    if done and reward == 0:
        return -1
    elif done and reward == 1:
        return 1
    else:
        return -0.01



for episode in range(episodes):
    # Reset environment before each episode
    state_info = env.reset()
    state = state_info[0]  # Extract the state value from the tuple
    done = False
    total_reward = 0
    while not done:

        # Get action
        if boltzmann:
            # Get action using Boltzmann policy
            q_values = q_table[state]
            exp_q_values = np.exp(q_values / temperature)
            action_probs = exp_q_values / np.sum(exp_q_values)
            action = np.random.choice(range(env.action_space.n), p=action_probs)

        else:
            # Get action using Epsilon-greedy policy
            if np.random.rand() < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q_table[state])

        # Perform action
        next_state, reward, done, _, info = env.step(action)

        # Apply custom reward
        custom_reward_value = custom_reward(state, reward, done)

        # Update Q-table
        q_table[state, action] = custom_reward_value + gamma * np.max(q_table[next_s


        # Update state
        state = next_state

        # Update total reward
        total_reward += custom_reward_value

    # Decay
    if boltzmann:
        # Decay temperature
        if temperature > 0.01:
            temperature *= temperature_decay
    else:
        # Decay epsilon
        if epsilon > epsilon_min:
            epsilon -= epsilon_decay
```

```
    # Log metrics
    wandb.log({'Episode': episode, 'Epsilon': epsilon, 'Total Reward': total_reward,

env.close()
# Finish the WandB run
wandb.finish()
```

Waiting for W&B process to finish... **(success).**

Run history:

Episode	
Epsilon	
Temperature	
Total Reward	

Run summary:

Episode	9999
Epsilon	0.0099
Temperature	1.0
Total Reward	-0.15

View run **mystical-spider-21** at: https://wandb.ai/jjuless/RL-FrozenLake/runs/rrycsbdx
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)
Find logs at: `.\wandb\run-20231031_223558-rrycsbdx\logs`

Testing the model

In [ ]:

```
# Load q-table, depending on which exploration method was used
if boltzmann:
    print("Boltzmann exploration")
    # save model
    np.save("q_table_boltzmann_slippery.npy", q_table)

else:
    print("Epsilon-greedy exploration")
    # save the q_table
    np.save("q_table_epsilon_slippery.npy", q_table)
```

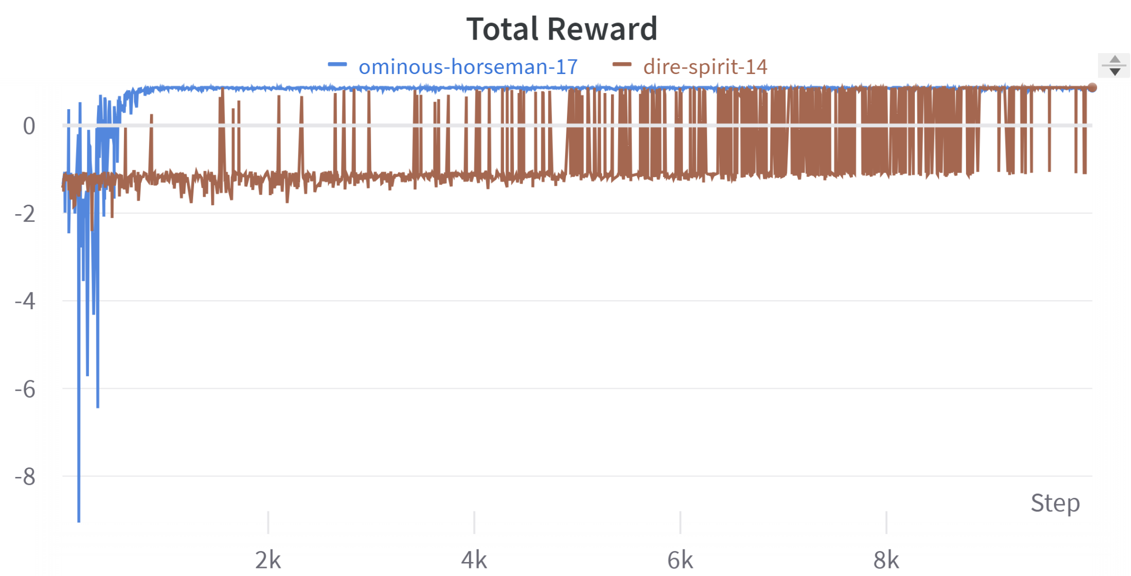
```
Epsilon-greedy exploration
```

In [ ]:

```
# load the q_table
if boltzmann:
```

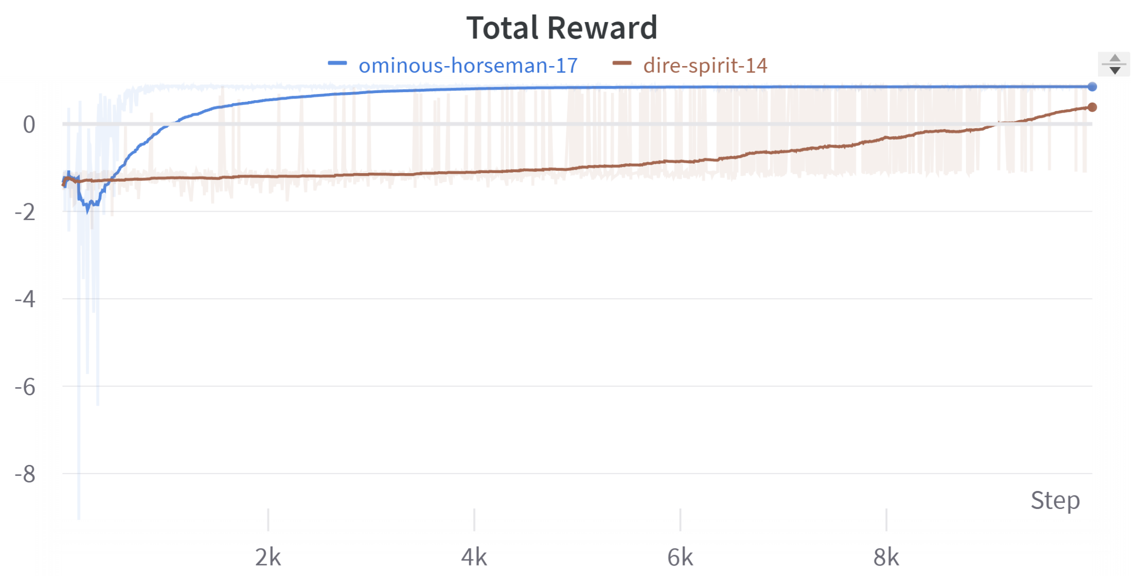
```python
    q_table = np.load("q_table_boltzmann.npy")
else:
    q_table = np.load("q_table_epsilon.npy")

# play the game with the q_table and render the environment to see the result
env = gym.make("FrozenLake-v1", is_slippery=is_slippery, desc=custom_map, render_mod

state_info = env.reset()

state = state_info[0]  # Extract the state value from the tuple

done = False
total_reward = 0
while not done:
    # Get action
    action = np.argmax(q_table[state])

    # Perform action
    next_state, reward, done, _, info = env.step(action)

    # Update state
    state = next_state

    # Update total reward
    total_reward += reward

    # Render environment
    env.render()

env.close()

print("Total reward:", total_reward)
```

```
Total reward: 1.0
```



In the image above, the best runs for the epsilon-greedy (dire-spirit-14, colored brown in the graph) and Boltzman exploration (ominous-horseman-17, colored blue) are visualised. In the graph, it becomes clear that using the epsilon method, the agent takes a while to learn. It can one episode score good, but the next (couple) episodes do worse than before. The learning decay was linear, so eventually the choices for the agent were less random and more based on the Q-table.

The Boltzman eploration starts off way worse than the epsilon-greedy, but it learns quick after about 250 episodes it gets steadily better. After plus minus 1000 episodes it has reached its full potential, as it can not get higher rewards than it already has.



If I apply smoothing to the graph, the trends become more clear. Boltzman starts off quite rough, but as soon as it finds what to do it becomes better quickly. For the epsilon-greedy, that's not the case. It learns very slowly.

Both models/Q-tables work well, but only for this specific environment. As soon as I change the environment, the Q-tables are useless as the 'good' and the 'bad' tiles the agent has learned are no longer located at the same place. The agent will have to learn everything again, so it does not possess any general knowledge.

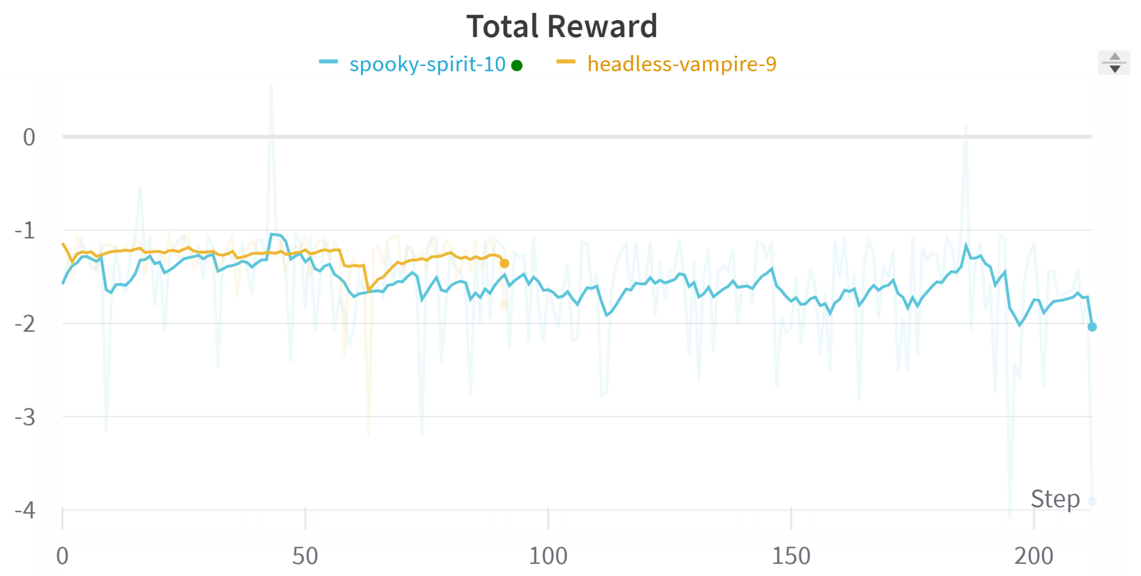
In [ ]:

```
import tensorflow as tf
from tensorflow.keras import layers, Model
import random
import wandb
import gymnasium as gym
import numpy as np


print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

# custom reward, if goal is reached return 1, taking a step gives -0.01 and falling
def custom_reward(state, reward, done):
    if done and reward == 0:
        return -1
    elif done and reward == 1:
        return 1
    else:
        return -0.01

def boltzmann_action_selection(q_values, temperature):
    q_values = q_values / temperature  # Scale the Q-values by the temperature
    exp_q_values = np.exp(q_values)
    probs = exp_q_values / np.sum(exp_q_values)  # Softmax to get probabilities
    action = np.random.choice(len(q_values), p=probs)  # Sample an action
    return action

# Hyperparameters
```

```
learning_rate = 0.1  # Learning rate
boltzmann = True  # Whether to use Boltzmann exploration or not

# Exploration parameters, based on Epsilon-greedy algorithm
gamma = 0.99  # Discount factor
epsilon = 1.0  # Exploration rate
epsilon_decay = 0.0001  # Decay of epsilon after each episode
epsilon_min = 0.01  # Minimum exploration rate

# Boltzmann exploration parameters
temperature = 1.0  # Temperature parameter
temperature_decay = 0.995  # Decay rate for temperature

episodes = 1000  # Number of episodes to run
is_slippery = False  # Slippery environment
batch_size = 512

# Creating custom map for a bigger 'challenge'
custom_map = [
    "SFFFFFFF",
    "FFFFFFFF",
    "FFFHFFFF",
    "FFFFFFFF",
    "FFFHFFFF",
    "FHHFFFHF",
    "FHFFHFHF",
    "FFFHFFFG"
]


# Create environment, using custom map
env = gym.make("FrozenLake-v1", is_slippery=is_slippery, desc=custom_map)

# Versions
print("TensorFlow version: ", tf.__version__)

input_model = layers.Input(shape=(env.observation_space.n,))
model = layers.Dense(32, activation="relu")(input_model)
model = layers.Dense(24, activation="relu")(model)
output_model = layers.Dense(env.action_space.n, activation="linear")(model)

model = Model(inputs=input_model, outputs=output_model)

loss = tf.keras.losses.MeanSquaredError()

initial_learning_rate = 0.01
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate,
    decay_steps=1000,
    decay_rate=0.96,
    staircase=True
)

optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule)

model.compile(loss=loss, optimizer=optimizer)

model.summary()

# Initialize wandb for logging
wandb.init(
    project="RL-FrozenLake-NN"
    )
```

```
replay_buffer = []

for episode in range(episodes):
    # Reset environment before each episode
    state_info = env.reset()
    state = state_info[0]  # Extract the state value from the tuple
    done = False
    total_reward = 0
    while not done:
        current_state = np.identity(env.observation_space.n)[state:state + 1]

        q_values = model.predict(current_state)[0]

        action = boltzmann_action_selection(q_values, temperature)

        # Perform action
        step = env.step(action)
        print(step)
        next_state, reward, done, _, _ = step

        # Apply custom reward
        custom_reward_value = custom_reward(state, reward, done)

        # Get the next state in a format suitable for your network
        next_state_formatted = np.identity(env.observation_space.n)[next_state:next_

        # Update Q-values using the neural network
        q_target = custom_reward_value + gamma * np.max(model.predict(next_state_for
        q_values = model.predict(current_state)
        q_values[0][action] = q_target

        replay_buffer.append((current_state, action, custom_reward_value, next_state
        if len(replay_buffer) >= batch_size:
            # Prepare batch
            batch = random.sample(replay_buffer, batch_size)
            states, actions, rewards, next_states = zip(*batch)
            states = np.vstack(states)
            next_states = np.vstack(next_states)

            # Compute Q-values and Q-targets
            q_values = model.predict(states)
            q_next = model.predict(next_states)
            q_targets = rewards + gamma * np.max(q_next, axis=1)
            q_values[np.arange(batch_size), actions] = q_targets

            # Train the model
            model.fit(states, q_values, epochs=1, verbose=0)
            replay_buffer = []  # Reset replay buffer

        # Update the state and total reward
        state = next_state
        total_reward += custom_reward_value

    # Decay temperature
    if temperature > 0.01:
        temperature *= temperature_decay
    # Log metrics to wandb
    wandb.log({'Episode': episode, 'Total Reward': total_reward, 'Temperature': temp

# Finish the WandB run
wandb.finish()

# Save the trained model
model.save("frozen_lake_nn_model.h5")
```

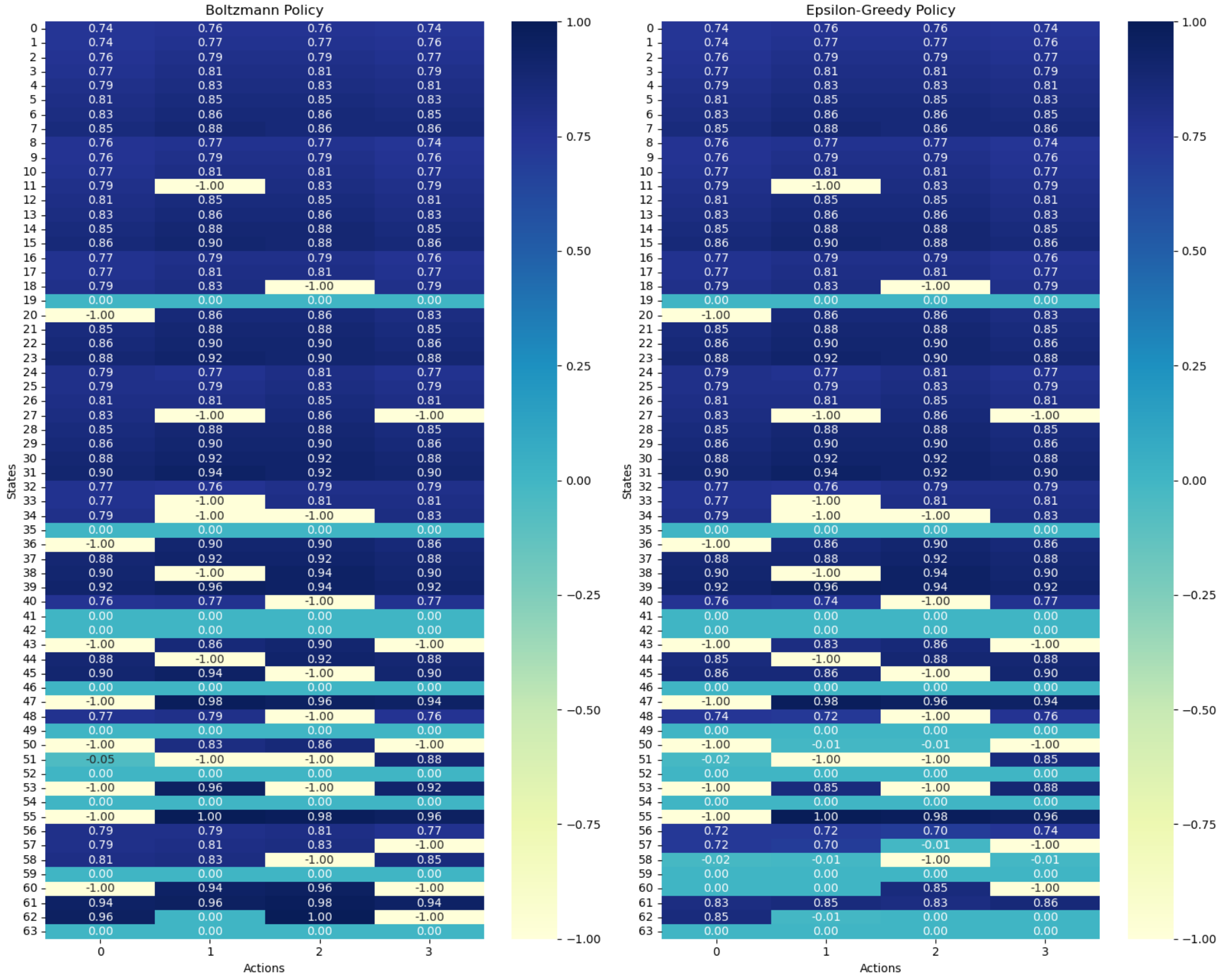
I stopped the previous cell early, as it did not show any progress when training the agent. I believe there is a flaw in my code, as the Q-tables for both epsilon-greedy and Boltzman exploration were generated within 30 seconds. After training more than 30 minutes for DQN, the agent still did not learn anything, despite using epsilon-greedy and Boltzman.



As can be seen in the graph above (with some smoothing aplied) the agent did not learn. By luck it was able to get the reward twice, but after that a crash in the reward was visible.

In [ ]:

```
# visualizing Q-tables for boltzmann and epsilon-greedy exploration
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# load the q_table
q_table_boltzmann = np.load("q_table_boltzmann.npy")
q_table_epsilon = np.load("q_table_epsilon.npy")



# Function to plot a Q-table
def plot_q_table(q_table, title, ax):
    sns.heatmap(q_table, annot=True, fmt=".2f", cmap="YlGnBu", cbar=True, ax=ax)
    ax.set_title(title)
    ax.set_xlabel("Actions")
    ax.set_ylabel("States")

# Create a figure with subplots
fig, axs = plt.subplots(1, 2, figsize=(15, 12))

# Plot the Q-tables
plot_q_table(q_table_boltzmann, "Boltzmann Policy", axs[0])
plot_q_table(q_table_epsilon, "Epsilon-Greedy Policy", axs[1])

# Show the plots
plt.tight_layout()
plt.show()
```



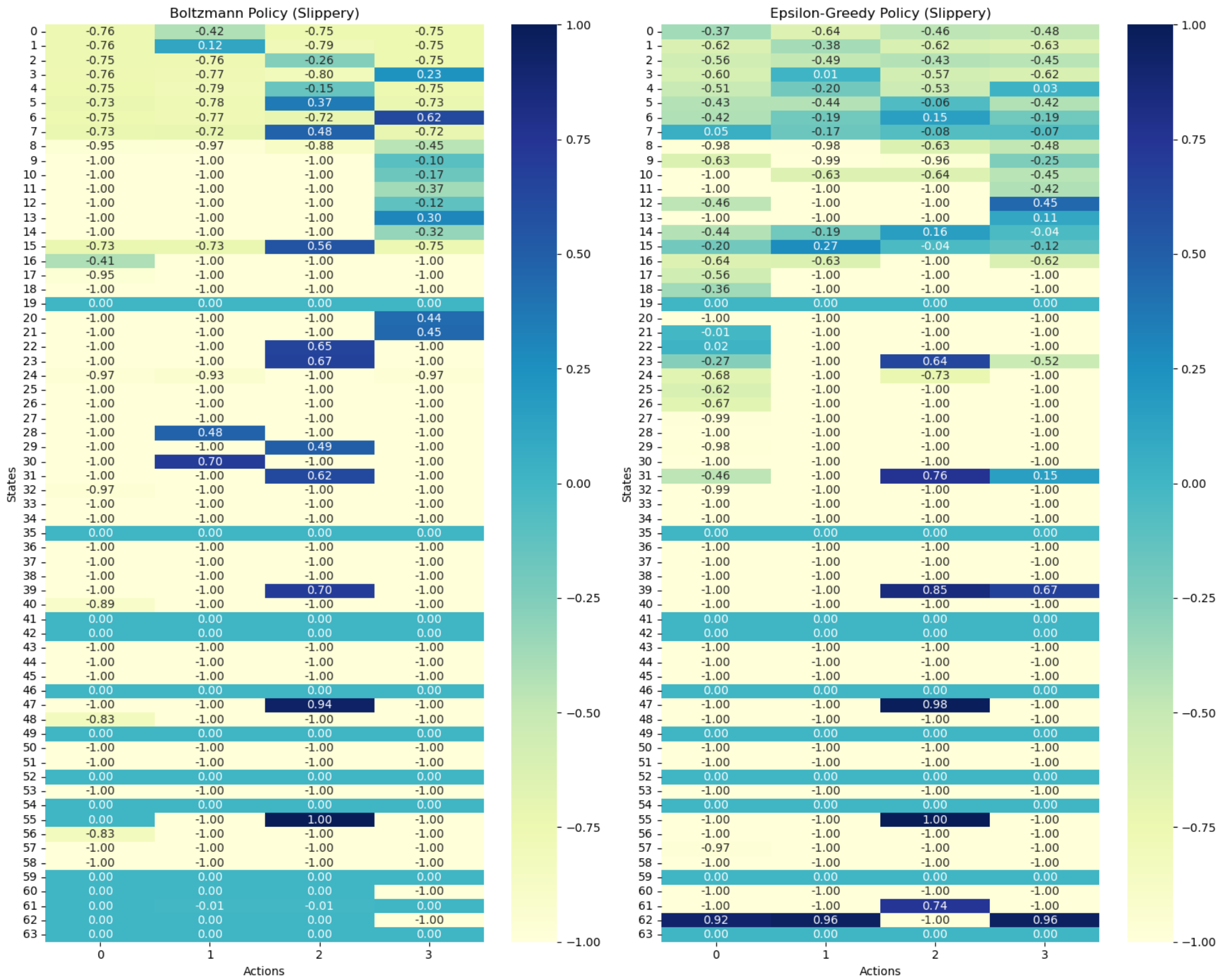
In the graph above are the Q-tables visualised for both algorithms. In first glance they look very similar, but on the bottom halves of the graph, you can spot differences. If a state has 0, then the agent has not explored what it can do in that state. However, the game ends when the agent falls in a hole or get's the reward, so it can never explore further from those states, resulting in a 0. Theoretically, it could mean that there is a frozen tile, but it just has never visited it, but that is kind of unlikely with the amount of episodes it has played and the chance it had to explore.

If a state has an action with a -1, it means that if the agent takes that action, it will fall in a hole and is thus not desired. The opposite is also true, if an action has a 1, it means the agent really desires to take that action, as it will result in a reward. The higher the number, the more the agent desires to take that action. In the beginning, the agent is quite far away from the reward tile and there are no holes close, so it does not really matter which way it goes, that's why the desire is relatively 'low'. As the agent gets closer to the reward, the desire to go to the reward increases, as it is the only way to get a reward.

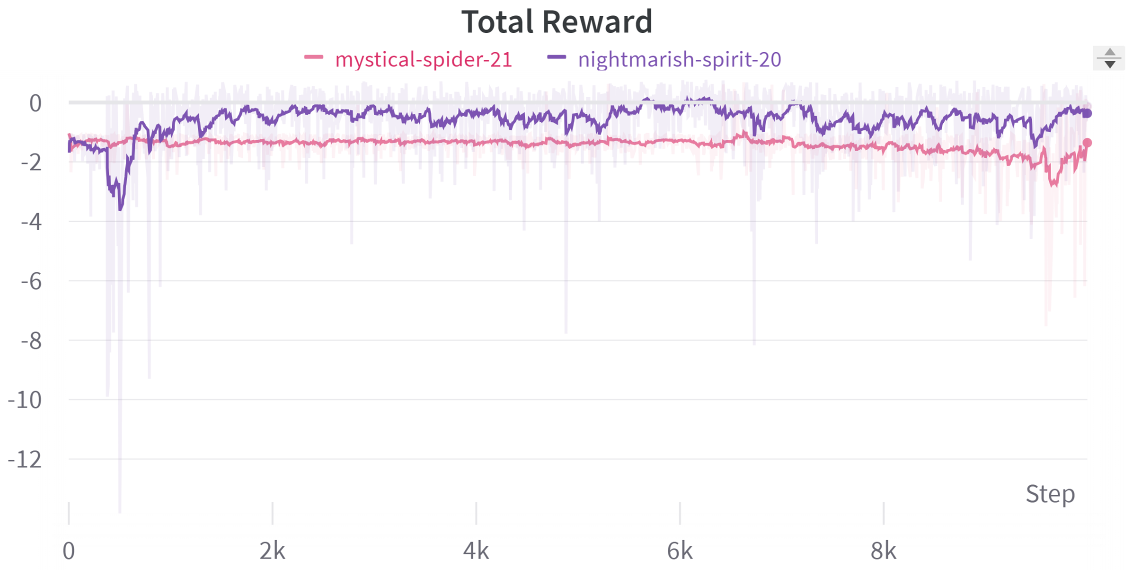
In [ ]:

```
# visualizing Q-tables for boltzmann and epsilon-greedy exploration
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Load the q_table
q_table_boltzmann = np.load("q_table_boltzmann_slippery.npy")
q_table_epsilon = np.load("q_table_epsilon_slippery.npy")



# Function to plot a Q-table
def plot_q_table(q_table, title, ax):
```

```
    sns.heatmap(q_table, annot=True, fmt=".2f", cmap="YlGnBu", cbar=True, ax=ax)
    ax.set_title(title)
    ax.set_xlabel("Actions")
    ax.set_ylabel("States")

# Create a figure with subplots
fig, axs = plt.subplots(1, 2, figsize=(15, 12))

# Plot the Q-tables
plot_q_table(q_table_boltzmann, "Boltzmann Policy (Slippery)", axs[0])
plot_q_table(q_table_epsilon, "Epsilon-Greedy Policy (Slippery)", axs[1])

# Show the plots
plt.tight_layout()
plt.show()
```



When slippery is turned on, the Q-tables change drastically; the majority of the actions is undesired as it could result in a fall in a hole. There are a lot less actions the agent is willing to take, as it brings more risks with it.



The reward progress graph (with some smoothing applied) shows a similar trend as when the environment was not slippery. The agent starts off rough with Boltzmann, but eventually learns better than the epsilon-desire method. Due to the less predictablity of the environment, the agent has more trouble learning what is good and what is bad. I think a DQN, or just a deep neural network, would be better suited as it can learn more complex patterns and can thus learn better what is good and what is bad. But sadly I was not able to get that working properly.