Crossy Road Reinforcement Learning

Neel Shah, Anish Bagri, Nathan Sankar, Akul Gokaram, Aditya Sharda

May 7, 2024

1 Introduction

Crossy Road is an arcade video game built around the age-old joke "Why did the chicken cross the road?" In the game, the chicken (controlled by the player) has to cross the road without getting hit by vehicles.

We want to develop a reinforcement learning (RL) game agent capable of playing Crossy Road. The game's endless and random nature makes it a great candidate for RL.

The agent will learn to maximize its score by getting the chicken to cross the road and avoid obstacles in its path, with the ultimate goal of crossing the road as many times as possible without collisions. Once the agent is capable of successfully getting the chicken to cross the road and reach the goal position, another goal could be to minimize the time it takes for the chicken to cross the road.

2 Configuration

There are several things we need to configure before we can begin.

2.1 Importing Libraries

We'll start by importing the necessary Python libraries.

[762]:
```
import copy
import math
import random
from collections import deque, namedtuple
from datetime import datetime
from itertools import count
from pathlib import Path

import cv2
import gymnasium as gym
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
import torch.optim as optim
from gymnasium.wrappers import TransformObservation, TransformReward
from matplotlib.ticker import MaxNLocator
from torchvision.transforms import v2
from tqdm import trange

try:
    from google.colab.patches import cv2_imshow
except ImportError:
    # code taken from: https://github.com/googlecolab/colabtools/blob/main/
↪google/colab/patches/__init__.py
    import PIL.Image
    from IPython import display

    def cv2_imshow(a, convert=True):
        """A replacement for cv2.imshow() for use in Jupyter notebooks.

        Args:
            a: np.ndarray. shape (N, M) or (N, M, 1) is an NxM grayscale image.␣
↪For
            example, a shape of (N, M, 3) is an NxM BGR color image, and a␣
↪shape of
            (N, M, 4) is an NxM BGRA color image.
            convert: boolean.
        """
        a = a.clip(0, 255).astype("uint8")
        # cv2 stores colors as BGR; convert to RGB
        if convert:
            if a.ndim == 3:
                if a.shape[2] == 4:
                    a = cv2.cvtColor(a, cv2.COLOR_BGRA2RGBA)
                else:
                    a = cv2.cvtColor(a, cv2.COLOR_BGR2RGB)
        display.display(PIL.Image.fromarray(a))
```

2.2 Controlling Randomness

Next, we'll make our results more reproducible (deterministic) by controlling sources of randomness, following the suggestions outlined in the PyTorch docs.

[763]:
```
random.seed(0)
np.random.seed(0)
torch.manual_seed(0)
```

[763]: <torch._C.Generator at 0x166ba4990>

2.3 Setting Device

We'll also select a device to store our tensors on.

[764]:
```python
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using mps device
```

3 Crossy Road Environment

Our first step is to implement a Crossy Road environment, which will encapsulate our representation of the reinforcement learning problem that the game poses.

For this, we will utilize the Gymnasium library (a fork of the OpenAI Gym library), which provides a standard API for RL and various reference environments.

Specifically, we will use the Freeway environment, which models an Atari game that closely resembles Crossy Road. This gives us a Pythonic interface to work with, which we can later use to develop RL models and create an agent that can play Crossy Road successfully.

3.1 Initializing the Environment

We will start off by initializing the environment using Gymnasium.

We pass in the following arguments (documented here) to specify the environment:

Environment Flavor:

The environment `id`, `mode`, and `difficulty` combine to specify the specific flavor of the environment:

- `id="ALE/Freeway-v5"`: simulates the Atari game Freeway via the Arcade Learning Environment (ALE) through the Stella emulator
- `mode=0`: selects Game 1 (Lake Shore Drive, Chicago, 3 A.M.) as the map to use
- `difficulty=0`: selects the default difficulty setting

Stochasticity:

As stated in the documentation:

> As the Atari games are entirely deterministic, agents can achieve state-of-the-art performance by simply memorizing an optimal sequence of actions while completely ignoring observations from the environment.

To combat this, we use `frameskip` and `repeat_action_probability`:

- `frameskip=4`: enables frame skipping (sets the number of frames to skip on each skip to 4)

- `repeat_action_probability=0.25`: enables sticky actions (sets the probability of repeating the previous action instead of executing the current action to 25%)

Simulation:

The parameters `full_action_space` and `render_mode` are used to specify how the environment is simulated:

- `full_action_space=False`: limits the action space to the 3 legal actions we will actually use instead of all 18 possible actions that can be performed on an Atari 2600 console
- `render_mode="rgb_array"`: specifies that the game should be rendered as an RGB frame

[765]:
```python
env = gym.make(
    id="ALE/Freeway-v5",
    mode=0,
    difficulty=0,
    obs_type="rgb",
    frameskip=4,
    repeat_action_probability=0.25,
    full_action_space=False,
    render_mode="rgb_array",
)
```

We will also modify the metadata to set `render_fps` to 30, meaning the game will run at 30 frames per second.

[766]:
```python
env.metadata["render_fps"] = 30
```

Now, we are ready to learn a little more about how out environment is implemented.

3.2 Observations

Let's start with the observation space.

[767]:
```python
env.observation_space
```

[767]:
```
Box(0, 255, (210, 160, 3), uint8)
```

This observation space represents the RGB image that is displayed to a human player.

3.3 Actions

Next, let's move on to the action space.

[768]:
```python
env.action_space
```

[768]:
```
Discrete(3)
```

This action space represents the actions that the chicken can take in each step:

[769]:
```python
action_meaning = {
    0: "NOOP",
```

```
    1: "UP",
    2: "DOWN",
}
print(action_meaning)
```

```
{0: 'NOOP', 1: 'UP', 2: 'DOWN'}
```

3.4 Rewards

Finally, let's move on to the reward range.

[770]:
```
env.reward_range
```

[770]:
```
(-inf, inf)
```

We can see that the reward range is $(-\infty, \infty)$. However, this is not very informative.

As the documentation tells us:

> You receive a point for every chicken that makes it to the top of the screen after crossing all the lanes of traffic.

3.4.1 Reward Modification

As the reward is currently, it is not suitable for training.

There are two actions that can be taken, moving the chicken up or down. However, the current reward function does not give a positive or negative reward for each individual action. There is only point for when you reach the end. We cannot train using RL techniques if the model can not associate and calculate rewards with each action.

As such, we will modify the reward such that there is a slight negative penality at each step. This will encourage the model to be quicker and move towards the goal since if it moves backwards or stays in the same spot the reward becomes more negative.

[771]:
```
env = TransformReward(env, lambda r: r - 0.05)
```

4 Agent-Environment Interaction

Our next step is to create a mechanism by which an agent can interact with the environment.

First, let's create a `log_step()` function that logs information about a certain time step.

[772]:
```
def log_step(step, action, observation, reward, terminated, truncated, info):
    print(f"\n***** Step {step} *****\n")
    if action is not None:
        print(f"Action: {action} ({action_meaning[action]})")
    if observation is not None:
        print("Observation:", observation)
    if reward is not None:
        print("Reward:", reward)
```

```
    if terminated is not None:
        print("Terminated:", terminated)
    if truncated is not None:
        print("Truncated:", truncated)
    if info is not None:
        print("Info:", info)
```

Now, let's create a `simulate()` function that takes in an environment, agent, and number of episodes.

It simulates running `num_episodes` episodes in the environment `env`, where the player's actions are defined by the behavior of `agent`.

[773]:
```
def simulate(
    env: gym.Env, transform: any, agent: any, num_episodes: int, step_show_freq:
 ↪ int
):
    for episode in range(num_episodes):
        print(f"########## Episode {episode} ##########")
        step = 0
        action = None
        observation, info = env.reset()
        observation = (
            transform(observation).to(device).clone().detach().unsqueeze(dim=0)
        )
        reward, terminated, truncated = 0.0, False, False
        if step % step_show_freq == 0:
            log_step(step, action, None, reward, terminated, truncated, info)
            view = env.render()
            cv2_imshow(view, convert=False)
        while not (terminated or truncated):
            step += 1
            action = agent.sample_action(observation)
            observation, reward, terminated, truncated, info = env.step(action.
 ↪item())
            observation = transform(observation).clone().detach().
 ↪unsqueeze(dim=0)
            if step % step_show_freq == 0:
                log_step(step, action.item(), None, reward, terminated,␣
 ↪truncated, info)
                view = env.render()
                cv2_imshow(view, convert=False)
```

5 Random Agent

To test our environment, let's create a `RandomAgent`.

5.1 Sampling an Action at Random

The agent will simply move the chicken randomly by sampling actions at random from the action space of the environment.

[774]:
```
class RandomAgent:
    def __init__(self, env):
        self.env = env

    def sample_action(self, _observation):
        action = self.env.action_space.sample()
        return torch.tensor(data=[[action]], dtype=torch.long, device=device)
```

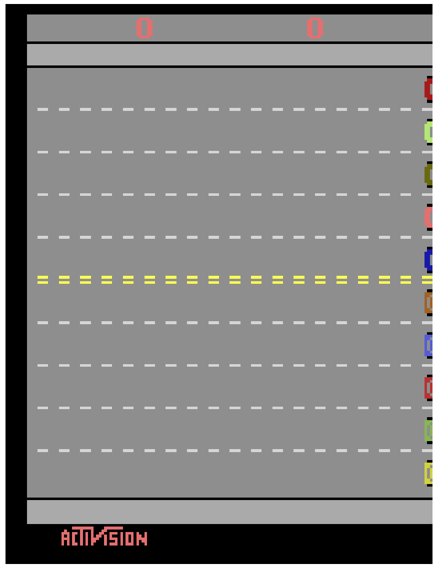
5.2 Simulation

Now, let's run a simulation.

To limit the amount of output we need to scroll through, we will only show every 100 steps.

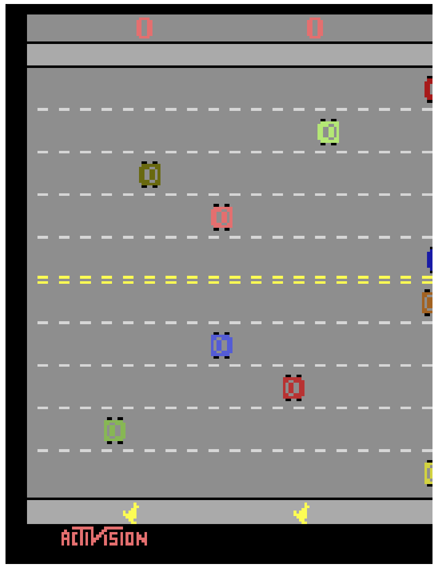
[775]:
```
transform = v2.Compose(
    [
        v2.ToImage(),
        v2.ToDtype(dtype=torch.float32, scale=True),
    ]
)
random_agent = RandomAgent(env)
simulate(env, transform, random_agent, 1, 100)
```

```
########## Episode 0 ##########

***** Step 0 *****

Reward: 0.0
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 0, 'frame_number': 0}
```

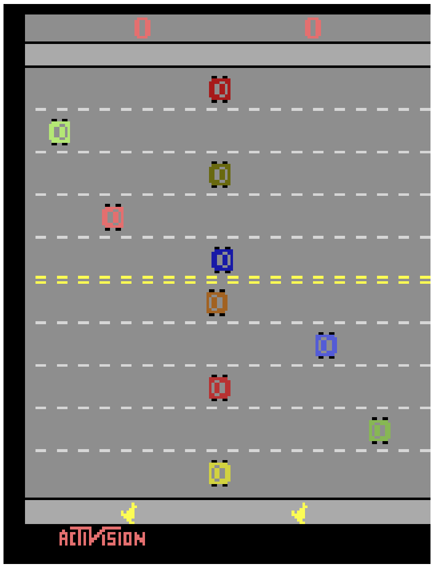



```
***** Step 100 *****

Action: 0 (NOOP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 400, 'frame_number': 400}
```

```
***** Step 200 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 800, 'frame_number': 800}
```

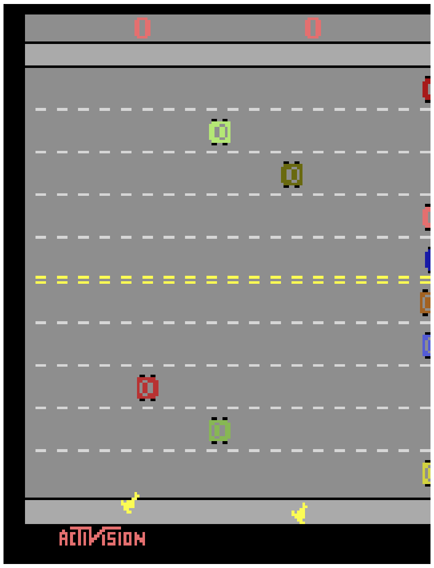



```
***** Step 300 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 1200, 'frame_number': 1200}
```



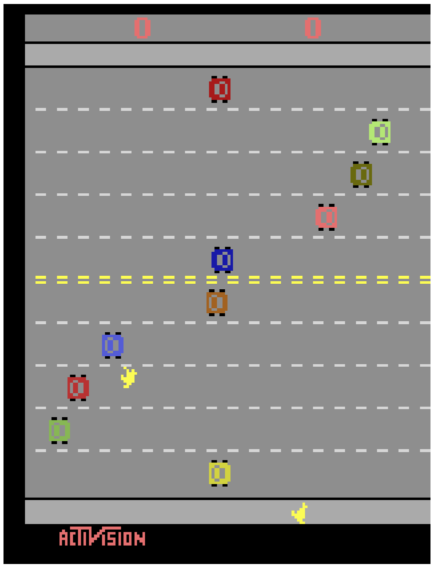
***** Step 400 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 1600, 'frame_number': 1600}



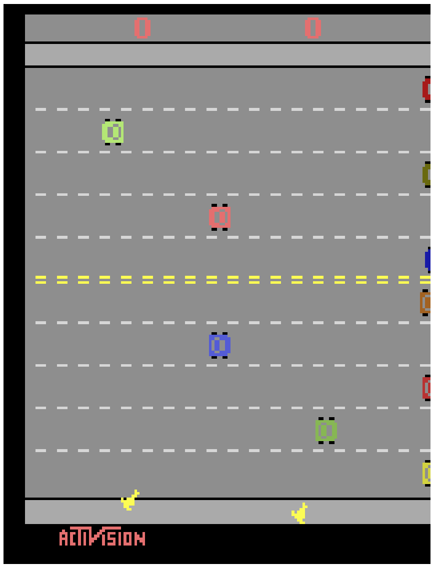
***** Step 500 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 2000, 'frame_number': 2000}

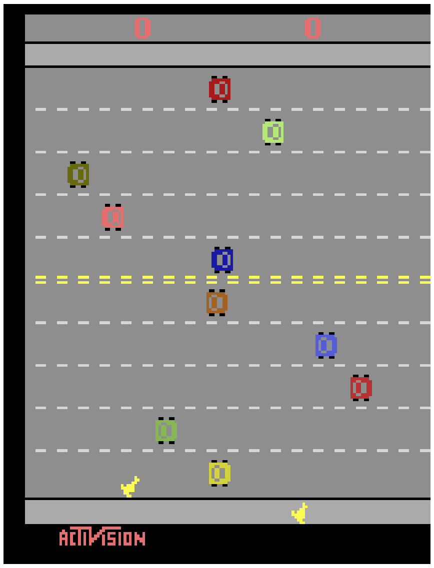


***** Step 600 *****

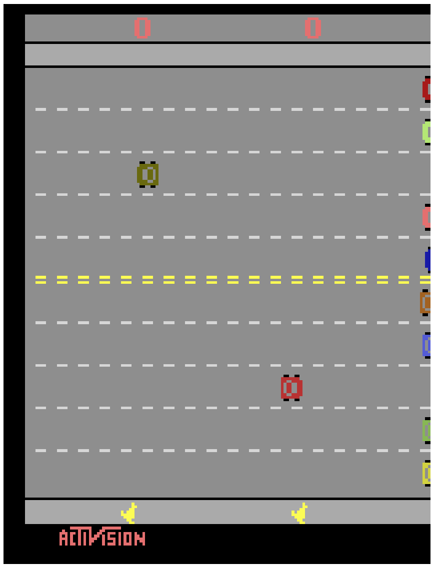
Action: 1 (UP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 2400, 'frame_number': 2400}



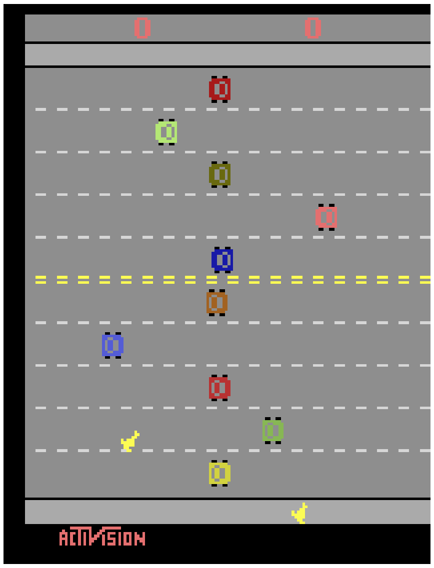
```
***** Step 700 *****

Action: 1 (UP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 2800, 'frame_number': 2800}
```



```
***** Step 800 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 3200, 'frame_number': 3200}
```

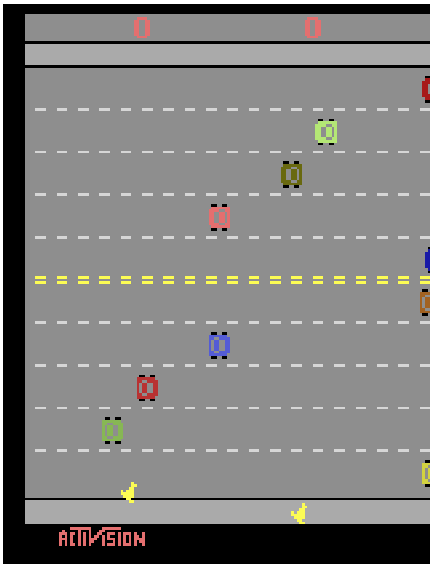



```
***** Step 900 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 3600, 'frame_number': 3600}
```

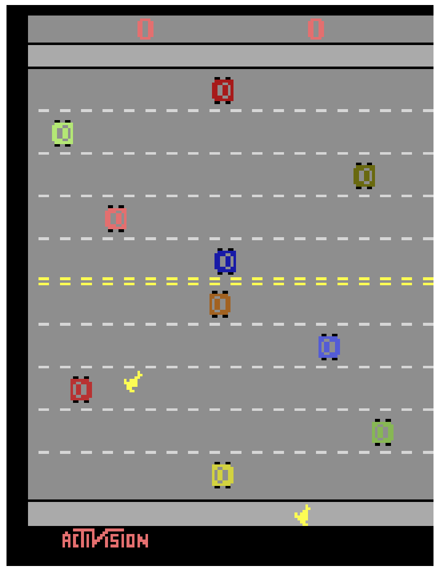


***** Step 1000 *****

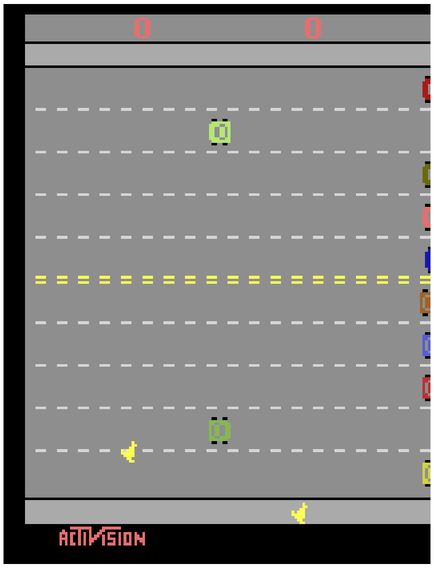
Action: 0 (NOOP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 4000, 'frame_number': 4000}



```
***** Step 1100 *****

Action: 0 (NOOP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 4400, 'frame_number': 4400}
```

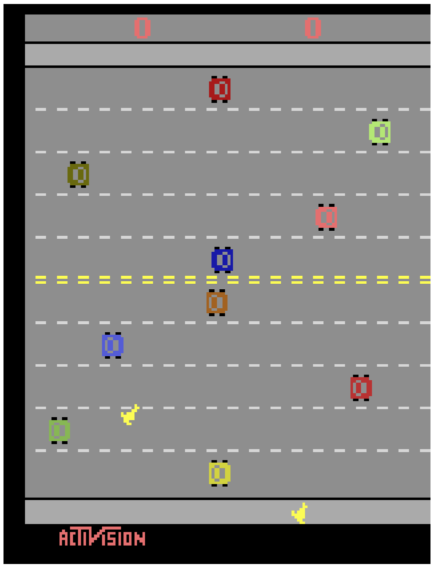



```
***** Step 1200 *****

Action: 0 (NOOP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 4800, 'frame_number': 4800}
```

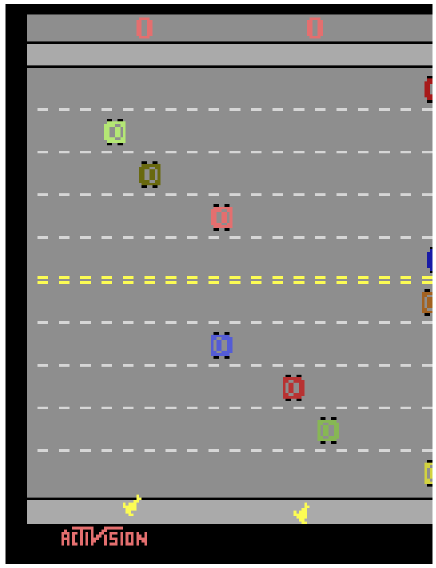


***** Step 1300 *****

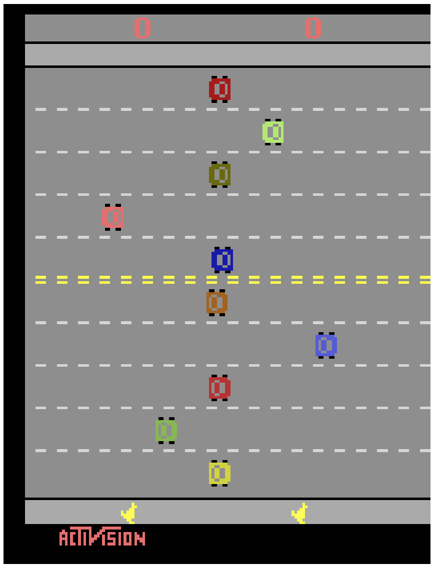
Action: 1 (UP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 5200, 'frame_number': 5200}



```
***** Step 1400 *****

Action: 0 (NOOP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 5600, 'frame_number': 5600}
```

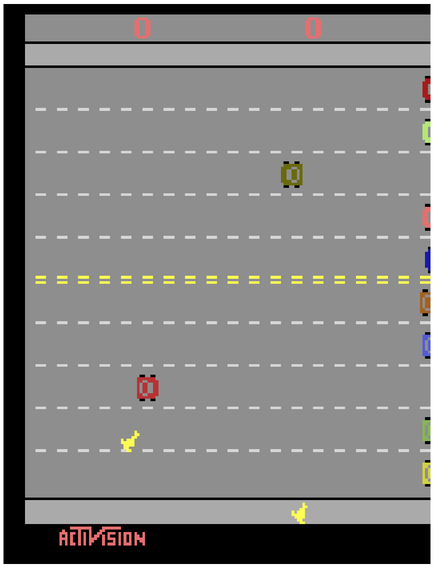



```
***** Step 1500 *****

Action: 0 (NOOP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 6000, 'frame_number': 6000}
```



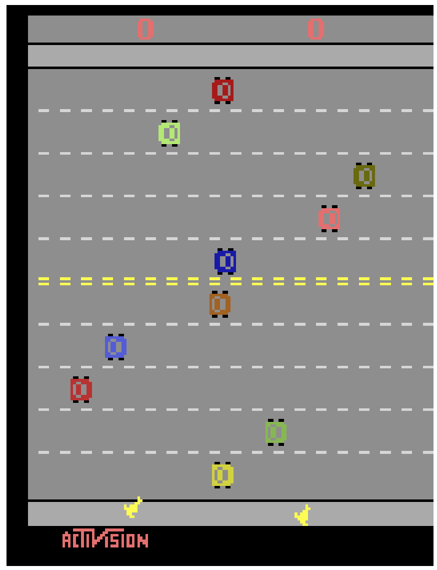
***** Step 1600 *****

Action: 0 (NOOP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 6400, 'frame_number': 6400}



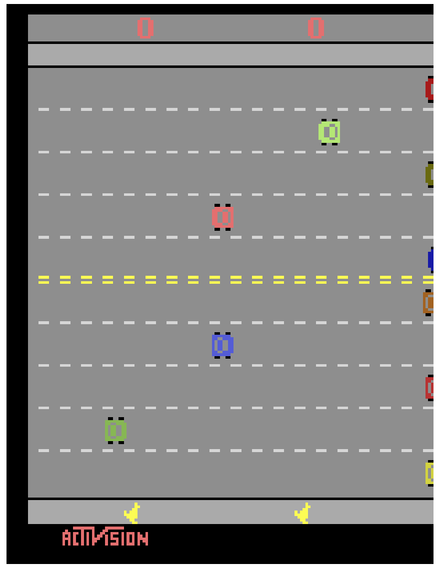
***** Step 1700 *****

Action: 1 (UP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 6800, 'frame_number': 6800}



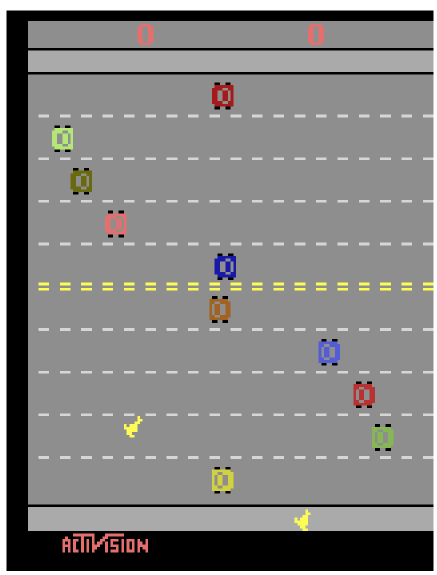
***** Step 1800 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 7200, 'frame_number': 7200}



***** Step 1900 *****

Action: 1 (UP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 7600, 'frame_number': 7600}



```
***** Step 2000 *****

Action: 1 (UP)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 8000, 'frame_number': 8000}
```

6 Reinforcement Learning Agent

Now that we have a way for an agent to interact with our environment, we can go ahead and develop a RL agent capable of playing the game Crossy Road.

This is a fairly complicated game, so ordinary Q-learning won't necesarily work.

Instead, we will use a variation called deep Q-learning that utilizes deep Q-networks (DQNs).

It works similarly to ordinary Q-learning, except it approximates the Q-values for each possible action at a state using a deep neural network instead of calculating the exact values.

We will follow PyTorch's official Reinforcement Learning (DQN) Tutorial as a guide, and adapt it to suit our needs.

6.1 Defining Hyperparameters

Let's start off by defining the hyperparameters we will use.

[776]:
```
hparams = {
    "memory_capacity": 10000,
    "batch_size": 128,
    "num_episodes": 5,
```

```
    "learning_rate": 1e-4,  # alpha
    "exploration_rate": {  # epsilon
        "start": 0.9,
        "end": 0.05,
        "decay": 1000,
    },
    "discount_factor": 0.99,  # gamma
    "update_rate": 0.005,  # tau
    "gradient_clip_value": 100,
}
```

6.2 Preprocessing Data

Before we can perform deep Q-learning, we'll need to preprocess our data to make it suitable for training.

We can specify a transformation function to transform or augment our data to fit our needs.

Here, we will:

- Turn the RGB values into an image
- Make the image grayscale
- Change the datatype to `float32`
- Flatten the image into a single dimension

[777]:
```
transform = v2.Compose(
    [
        v2.ToImage(),
        v2.Grayscale(),
        v2.ToDtype(dtype=torch.float32, scale=True),
        v2.Lambda(lambda x: torch.flatten(input=x).to(device)),
    ]
)
```

6.3 Defining Models

Next, we can define our models, i.e. build our neural networks for use in the deep Q-learning algorithm. We will be using a deep Q-network (DQN).

Our model will be a feed-forward neural network that takes in the difference between the current and previous screen patches in order to predict the Q-values.

Essentially, the network is used to predict the expected return of taking each action given the current state.

[778]:
```
class DQN(nn.Module):
    def __init__(self, observation_size, action_size):
        super().__init__()
        self.layer1 = nn.Linear(observation_size, 128)
        self.layer2 = nn.Linear(128, 128)
```

```
        self.layer3 = nn.Linear(128, action_size)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

6.4 Defining Loss Function and Optimizer

With our network defined, we can now define our loss function and optimizer.

6.4.1 Loss Function

We will use smooth L1 loss, as is common for DQN.

[779]:
```
loss_fn = nn.SmoothL1Loss()
```

6.4.2 Optimizer

We will use the AdamW (adaptive momentum estimation with weight decay) optimizer, since it is particulalry robust.

It combines the benefits of momentum (such as in SGD (stochastic gradient descent) with momentum) and adaptive learning rates (such as in RMSprop (root mean square propagation)).

[780]:
```
optimizer = optim.AdamW
```

6.5 Defining Experience Replay Memory

To perform deep Q-learning, we'll rely on a technique known as experience replay memory.

6.5.1 Representing a Transition

First, we'll represent a transition using a tuple of the form `(observation, action, next_observation, reward)`, which maps a `(observation, action)` pair to the corresponding result `(next_observation, reward)`.

[781]:
```
Transition = namedtuple(
    "Transition", ("observation", "action", "next_observation", "reward")
)
```

6.5.2 Representing Replay Memory

Next, we'll represent replay memory using a deque (double-ended queue) of fixed capacity. It will store the transitions observed most recently.

We'll also have a method for selecting a random batch of transitions.

[782]:
```
class ReplayMemory:
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)
```

```python
    def push(self, *args):
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

6.6 Deep Q-Network Agent

Now, we are ready to build our DQN agent.

6.6.1 Initializing the Agent

In the `__init__()` method, we do the following:

- Define the hyperparameters using the given values:
  - `hparams`: the dictionary of hyperparameters
- Preprocess the data to form a suitable training environment `env`:
  - Wrap the given enivronment `env` in a `TransformObservation` wrapper that transforms the observation using the given transformation function `transform`.
  - Calculate the relevant sizes of the environment:
    * `observation_size`: the size of an observation in the observation space
    * `action_size`: the size of an action in the action space
- Define the models (neural networks):
  - `policy_net`: used for action selection (to select the best action to take at a given state, i.e. the action with the highest Q-value)
  - `target_net`: used for action evaluation (to calculate the target Q-value of taking that action at the given state)
    * Note that `target_net` is just a copy of `policy_net` that's updated less frequently.
- Define the loss function and optimizer using the given values:
  - `loss_fn`: the loss function
  - `optimizer`: the optimizer
- Define the experience replay memory:
  - `memory`: the memory
- Define variables used for training:
  - `total_steps`: the total number of steps (the number of times an action was sampled)
  - `episode_durations`: how long each episode takes

[783]:
```python
class DQNAgent:
    def __init__(self, hparams, env, transform, loss_fn, optimizer):
```

```
        # defining hyperparameters
        self.hparams = copy.deepcopy(hparams)

        # preprocessing data
        self.env = TransformObservation(env, transform)
        observation, _ = self.env.reset()
        self.observation_size = len(observation)
        self.action_size = self.env.action_space.n

        # defining models
        self.policy_net = DQN(self.observation_size, self.action_size).
↪to(device)
        self.target_net = DQN(self.observation_size, self.action_size).
↪to(device)
        self.target_net.load_state_dict(self.policy_net.state_dict())

        # defining loss function and optimizer
        self.loss_fn = loss_fn
        self.optimizer = optimizer(
            params=self.policy_net.parameters(),
            lr=self.hparams["learning_rate"],
            amsgrad=True,
        )

        # defining experience replay memory
        self.memory = ReplayMemory(self.hparams["memory_capacity"])

        # defining variables for training
        self.total_steps = 0
        self.episode_durations = []
```

6.6.2 Sampling an Action from the Policy

In the `sample_action()` method, the agent will sample an action using an epsilon-greedy policy, which means balancing the amount of exploration vs. exploitation as per the hyperparameter for exploration rate (epsilon, ϵ).

Note that we are using epsilon decay, so the exploration rate will decrease exponentially over time.

[784]:
```
class DQNAgent(DQNAgent):
    def sample_action(self, observation):
        # epsilon
        exploration_rate = self.hparams["exploration_rate"]["end"] + (
            self.hparams["exploration_rate"]["start"]
            - self.hparams["exploration_rate"]["end"]
        ) * math.exp(
            -1.0 * self.total_steps / self.hparams["exploration_rate"]["decay"]
        )
```

```
        self.total_steps += 1

        if np.random.uniform(0, 1) < exploration_rate:  # explore
            action = self.env.action_space.sample()
        else:  # exploit
            action = self.policy_net(observation).argmax(dim=1).item()

        # convert action to tensor
        return torch.tensor(
            data=[[action]],
            dtype=torch.long,
            device=device,
        )
```

6.6.3 Plotting Episode Durations

To visualize our training progress, we will create a `plot_episode_durations()` method that plots the durations of episodes, along with a rolling average over the last 100 episodes.

[785]:
```
class DQNAgent(DQNAgent):
    def plot_episode_durations(self):
        # convert episode durations to tensor
        episode_durations_tensor = torch.tensor(
            self.episode_durations, dtype=torch.float32
        )

        # set up figure
        plt.figure()
        plt.title("Episode Durations")
        plt.xlabel("Episode")
        plt.ylabel("Duration")

        # make x-axis use integers
        plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True,␣
↪min_n_ticks=0))

        # plot episode durations
        plt.plot(episode_durations_tensor.numpy())

        # plot rolling average of episode durations
        if len(episode_durations_tensor) >= 100:
            means = (
                episode_durations_tensor.unfold(dimension=0, size=100, step=1)
                .mean(dim=1)
                .view(size=-1)
            )
            means = torch.cat((torch.zeros(99), means))
```

```
            plt.plot(means.numpy())
```

6.6.4 Optimizing the Policy Network

In the `optimize_policy_net()` method, we update the policy network's weights.

We do the following:

- Sample a batch of `Transitions` from the experience replay memory.
- Convert the batch-array of `Transitions` into a `Transition` of batch-arrays.
- Concatenate all of the tensors, accounting for terminal observations as necessary.
- Compute the Q-values (utility for observation-action pairs).
- Compute the V-values (utility for next observations) and the corresponding expected Q-values.
- Compute the loss and optimize the policy network's weights.

[786]:
```
class DQNAgent(DQNAgent):
    def optimize_policy_net(self):
        # sample batch of transitions from memory
        if len(self.memory) < self.hparams["batch_size"]:
            return
        transitions = self.memory.sample(self.hparams["batch_size"])

        # convert batch-array of Transitions into Transition of batch-arrays
        batch = Transition(*zip(*transitions, strict=True))

        # concatenate tensors
        # observation, action, reward
        observation_batch = torch.cat(batch.observation)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)
        # nonterminal next observation
        nonterminal_next_observation_mask = torch.tensor(
            tuple(map(lambda obs: obs is not None, batch.next_observation)),
            dtype=torch.bool,
            device=device,
        )
        nonterminal_next_observation_batch = torch.cat(
            [obs for obs in batch.next_observation if obs is not None]
        )

        # compute Q-values
        observation_action_values = self.policy_net(observation_batch).gather(
            dim=1, index=action_batch
        )

        # compute V-values and corresponding expected Q-values
        next_observation_values = torch.zeros(
```

```
            size=(self.hparams["batch_size"],), device=device
        )
        with torch.no_grad():
            next_observation_values[nonterminal_next_observation_mask] = (
                self.target_net(nonterminal_next_observation_batch).max(dim=1).
↪values
            )
        expected_observation_action_values = (
            next_observation_values * self.hparams["discount_factor"]
        ) + reward_batch

        # compute loss and optimize policy net
        loss = self.loss_fn(
            observation_action_values,
            expected_observation_action_values.unsqueeze(dim=1),
        )
        self.optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_value_(
            parameters=self.policy_net.parameters(), clip_value=100
        )
        self.optimizer.step()
```

6.6.5 Optimizing the Target Network

In the `optimize_target_net()` method, we update the target network's weights. It's a soft update based on the hyperparameter for update rate (tau, τ).

[787]:
```
class DQNAgent(DQNAgent):
    def optimize_target_net(self):
        # get state dictionaries
        policy_net_state_dict = self.policy_net.state_dict()
        target_net_state_dict = self.target_net.state_dict()

        # update target net state dictionary
        for key in policy_net_state_dict:
            old_weight = (1 - self.hparams["update_rate"]) *␣
↪target_net_state_dict[key]
            new_weight = self.hparams["update_rate"] *␣
↪policy_net_state_dict[key]
            target_net_state_dict[key] = old_weight + new_weight

        # load target net state dictionary into target net
        self.target_net.load_state_dict(target_net_state_dict)
```

6.6.6 Training the Agent

In the `train()` method, we train our agent in order to make its policy "better" at solving the environment.

We also handle saving and loading of the models and plots as necessary.

[788]:
```
class DQNAgent(DQNAgent):
    def train(self):
        # load models and plots if available
        if (
            Path("policy_net.pth").is_file()
            and Path("target_net.pth").is_file()
            and Path("episode_durations.png").is_file()
        ):
            self.policy_net.load_state_dict(torch.load("policy_net.pth"))
            self.target_net.load_state_dict(torch.load("target_net.pth"))
            img = plt.imread("episode_durations.png")
            plt.axis("off")
            plt.imshow(img)
            return

        for _ in trange(self.hparams["num_episodes"], desc="Episodes"):
            # reset environment
            display.clear_output(wait=True)
            step = 0
            observation, _ = self.env.reset()
            observation = observation.clone().detach().unsqueeze(dim=0)
            reward, terminated, truncated = 0.0, False, False

            while not (terminated or truncated):
                step += 1

                # sample action from policy
                action = self.sample_action(observation)
                action = action.clone().detach()
                next_observation, reward, terminated, truncated, _ = self.env.
↪step(
                    action.item()
                )
                next_observation = (
                    None
                    if terminated
                    else next_observation.clone().detach().unsqueeze(0)
                )
                reward = torch.tensor(data=[reward], dtype=torch.float32,␣
↪device=device)
                reward = reward.clone().detach()
```

```
                # store transition in memory
                self.memory.push(observation, action, next_observation, reward)

                # move to next observation
                observation = next_observation

                # optimize model
                self.optimize_policy_net()
                self.optimize_target_net()

            # log episode duration and update plot
            self.episode_durations.append(step)
            self.plot_episode_durations()
            plt.show()

        # update plot
        display.clear_output(wait=True)
        self.plot_episode_durations()

        # get current timestamp
        timestamp = datetime.now().isoformat(timespec="minutes")

        # same models and plots in current directory
        torch.save(self.policy_net.state_dict(), "policy_net.pth")
        torch.save(self.target_net.state_dict(), "target_net.pth")
        plt.savefig("episode_durations.png")

        # save archive of models and plots in runs/ directory
        Path(f"runs/{timestamp}").mkdir(parents=True, exist_ok=True)
        torch.save(self.policy_net.state_dict(), f"runs/{timestamp}/policy_net.
↪pth")
        torch.save(self.target_net.state_dict(), f"runs/{timestamp}/target_net.
↪pth")
        plt.savefig(f"runs/{timestamp}/episode_durations.png")

        # show plot
        plt.show()
```

6.6.7 Inference Mode

In the `inference_mode()` method, we set the hyperparameters for exploration rate (epsilon, ϵ) to 0, so we don't explore anymore and only exploit.

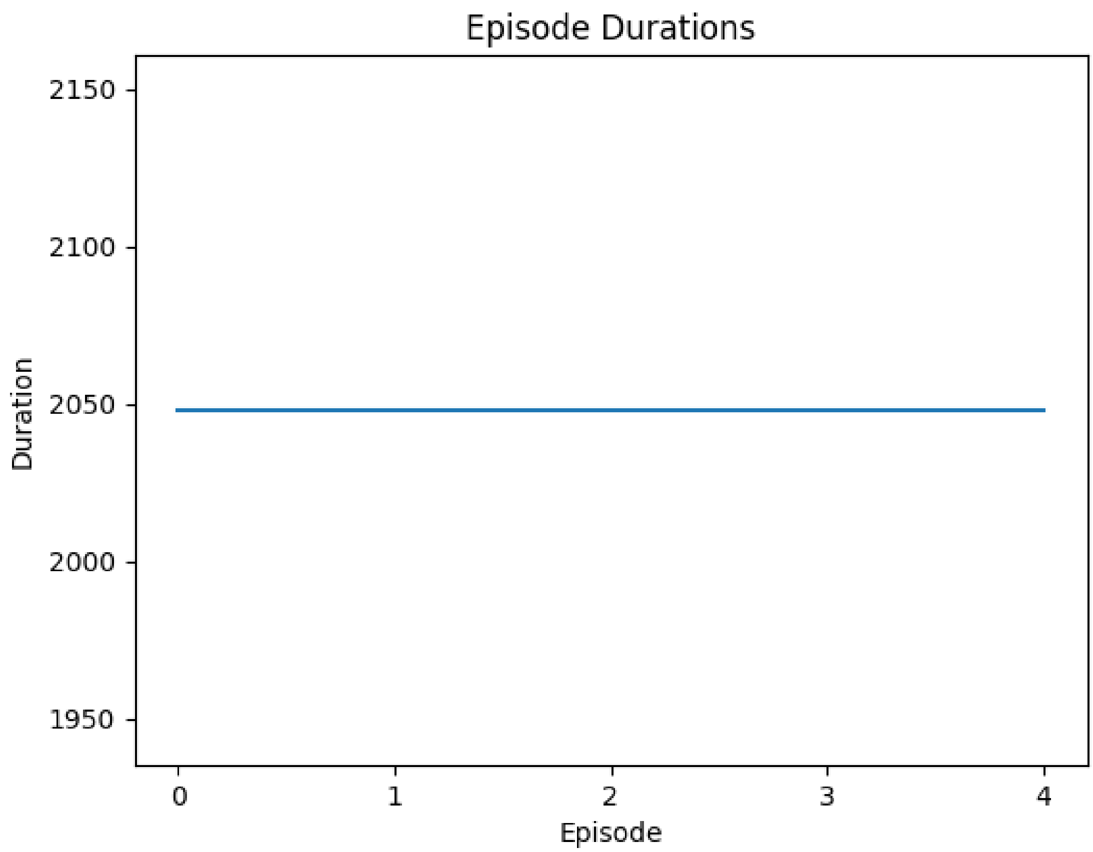
[789]:
```
class DQNAgent(DQNAgent):
    def inference_mode(self):
        self.hparams["exploration_rate"]["start"] = 0.0
```

```
        self.hparams["exploration_rate"]["end"] = 0.0
```

6.7 Training the Agent

Now, we're equipped with everything we need to train our DQN agent.

[790]:
```
dqn_agent = DQNAgent(hparams, env, transform, loss_fn, optimizer)
dqn_agent.train()
```



6.8 Simulation

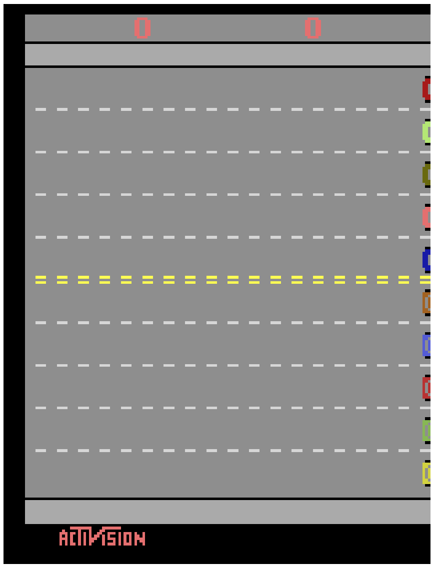
Now, let's run a simulation.

To limit the amount of output we need to scroll through, we will only show every 100 steps.

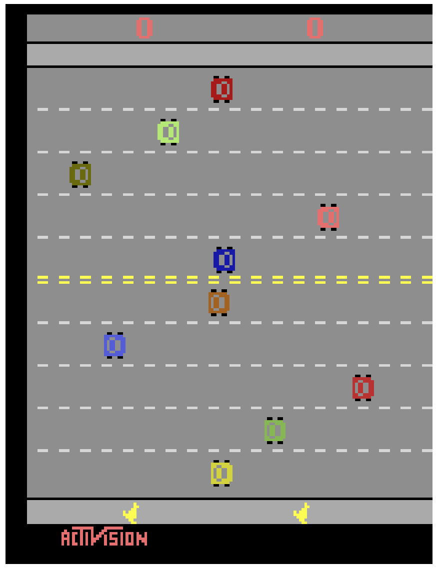
[791]:
```
dqn_agent.inference_mode()
simulate(env, transform, dqn_agent, 1, 100)
```

```
########## Episode 0 ##########


***** Step 0 *****
```

```
Reward: 0.0
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 0, 'frame_number': 49151}
```

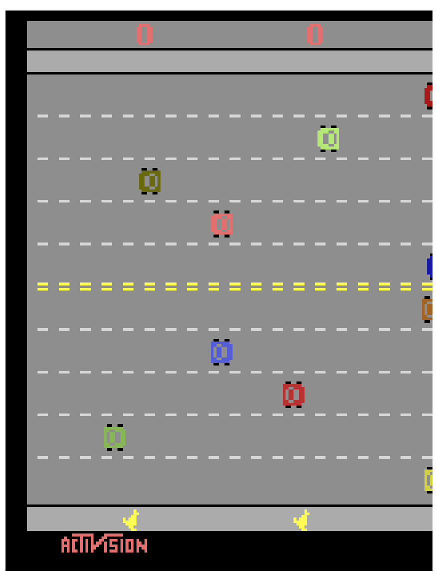



```
***** Step 100 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 400, 'frame_number': 49551}
```

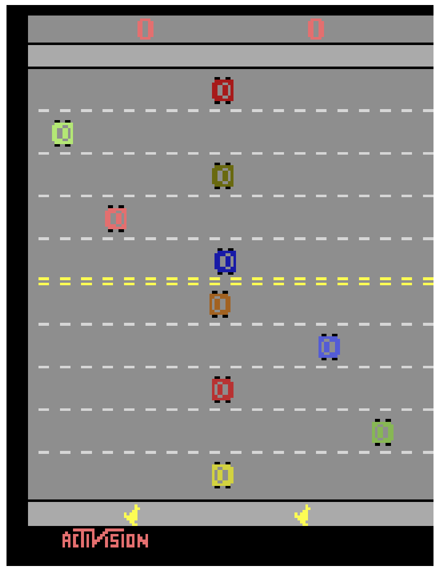


***** Step 200 *****

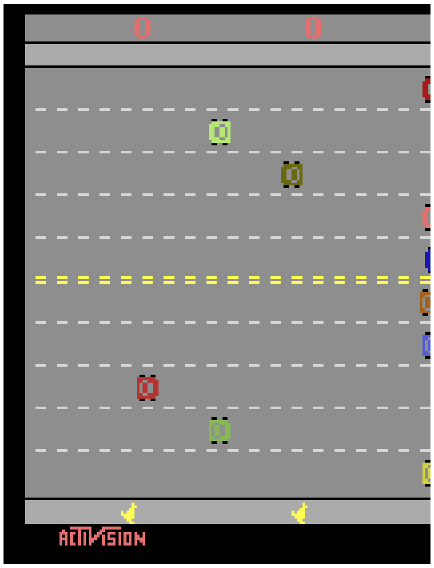
Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 800, 'frame_number': 49951}



```
***** Step 300 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 1200, 'frame_number': 50351}
```

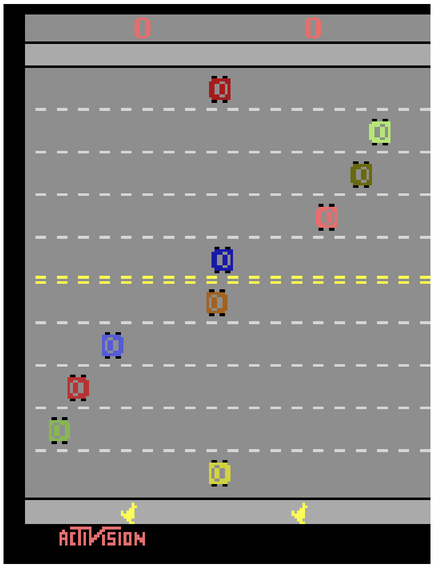


```
***** Step 400 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 1600, 'frame_number': 50751}
```



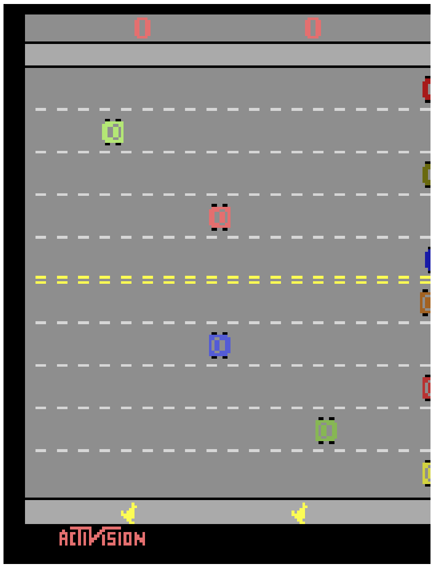
***** Step 500 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 2000, 'frame_number': 51151}



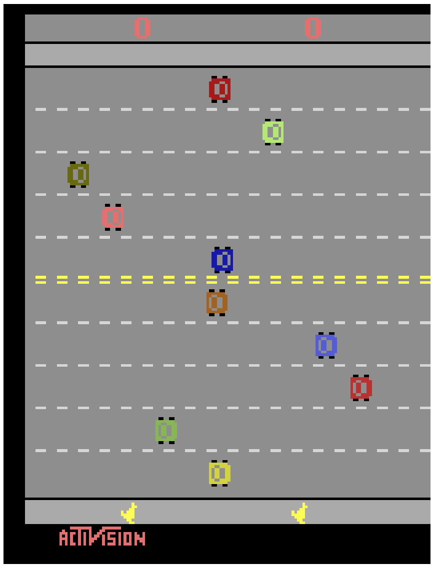
***** Step 600 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 2400, 'frame_number': 51551}



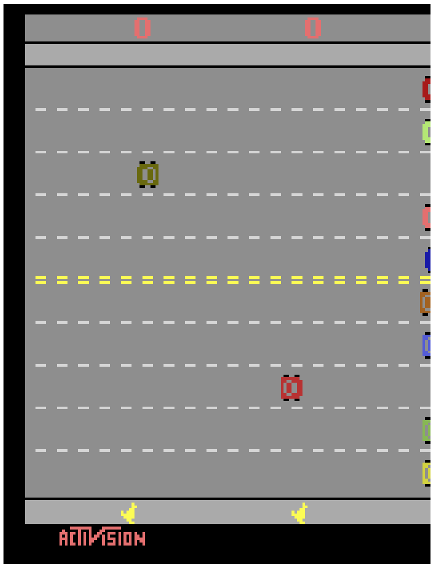
***** Step 700 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 2800, 'frame_number': 51951}



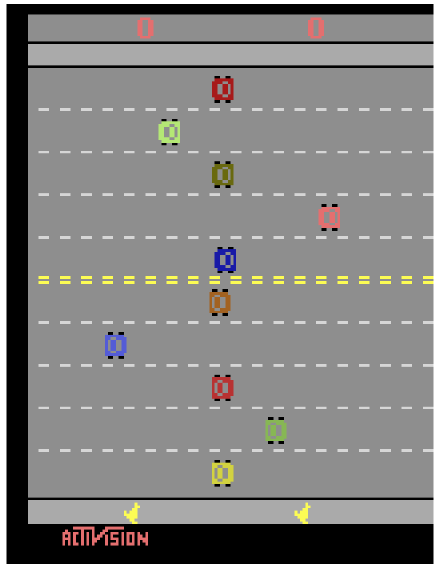
***** Step 800 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 3200, 'frame_number': 52351}



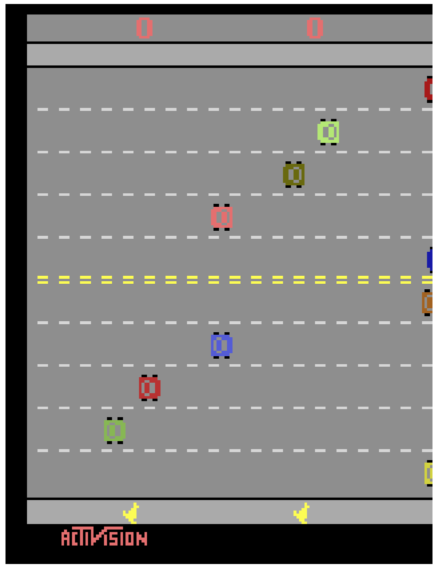
***** Step 900 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 3600, 'frame_number': 52751}



***** Step 1000 *****

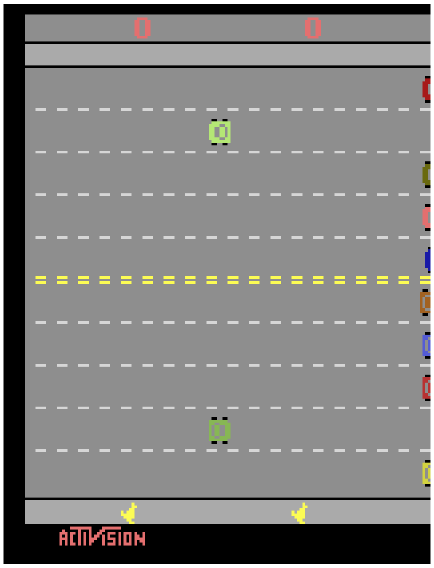
Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 4000, 'frame_number': 53151}



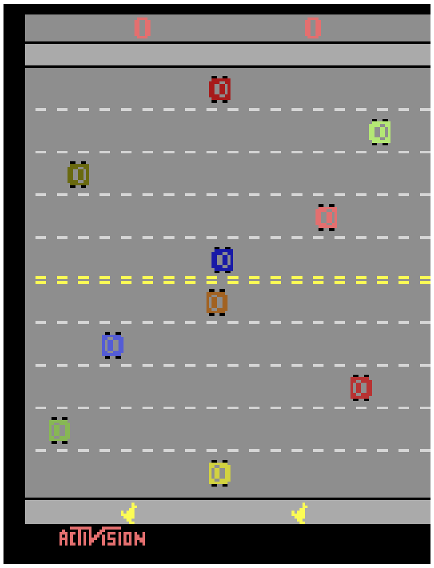
***** Step 1100 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 4400, 'frame_number': 53551}

```
***** Step 1200 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 4800, 'frame_number': 53951}
```

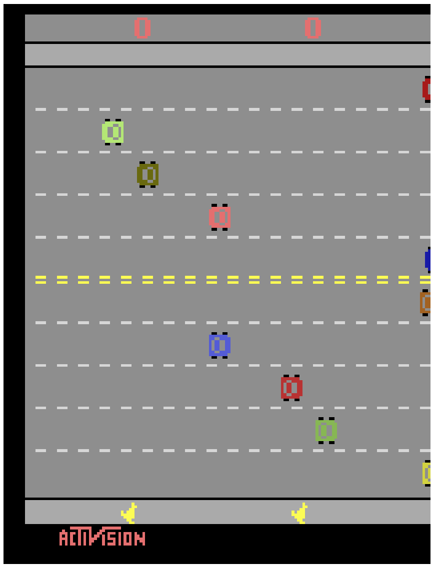



```
***** Step 1300 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 5200, 'frame_number': 54351}
```

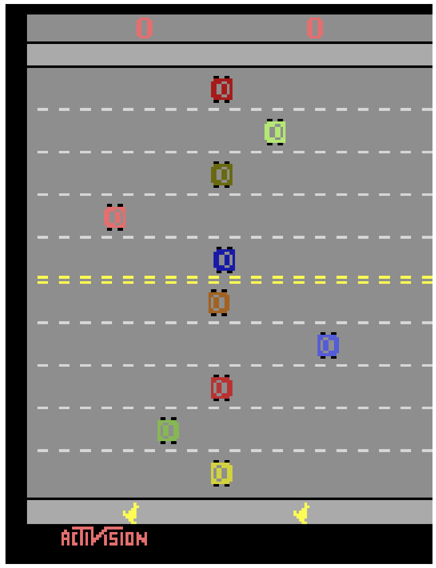


***** Step 1400 *****

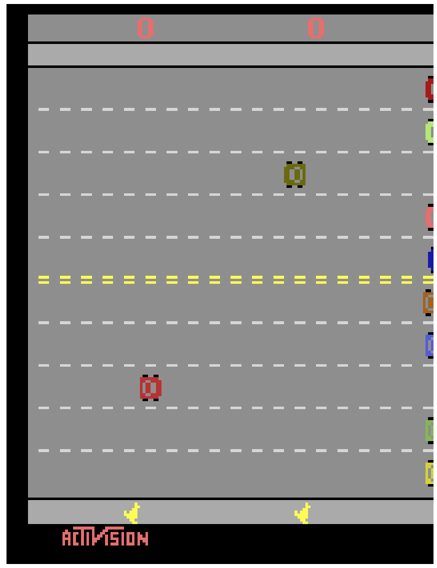
Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 5600, 'frame_number': 54751}



```
***** Step 1500 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 6000, 'frame_number': 55151}
```




```
***** Step 1600 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 6400, 'frame_number': 55551}
```

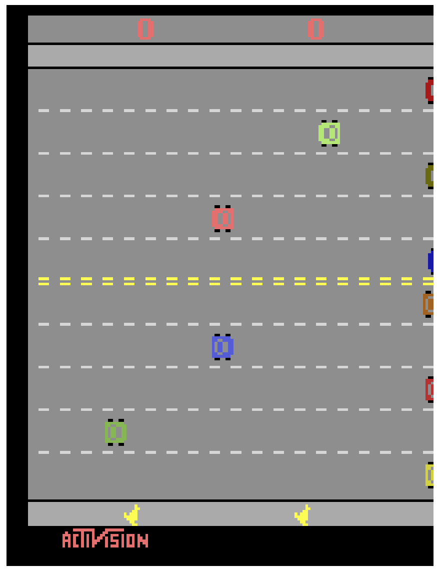


***** Step 1700 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 6800, 'frame_number': 55951}

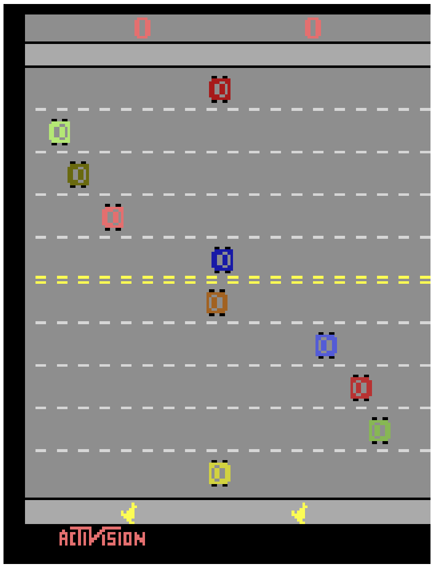
***** Step 1800 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 7200, 'frame_number': 56351}



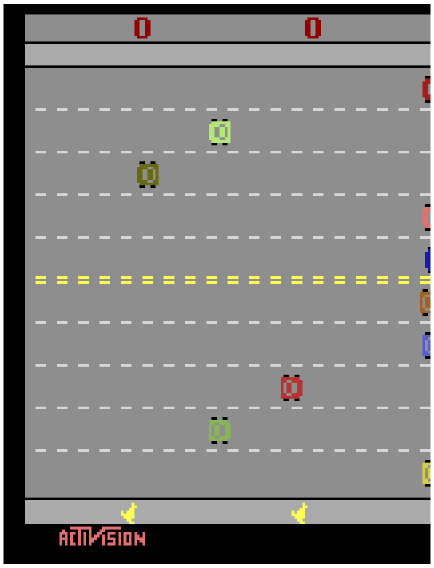
***** Step 1900 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 7600, 'frame_number': 56751}



***** Step 2000 *****

Action: 2 (DOWN)
Reward: -0.05
Terminated: False
Truncated: False
Info: {'lives': 0, 'episode_frame_number': 8000, 'frame_number': 57151}



7 Conclusion

As we can see, using the power of deep Q-learning, we were able to develop a RL game agent capable of playing Crossy Road that performs significantly better than a random agent.

While the agent is not perfect, it has impressive performance, considering the complexity of the game and the compute constraints we were under.