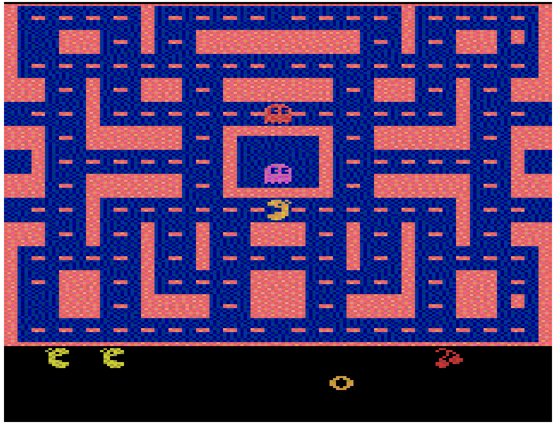
Using a DQN to simulate an Agent playing MsPacman



Ty Turner
Data Science
w/ Dr. Cook

Building the DQN-Model

- Using Pytorch packages, we build a DQN model that fits the Gymnasium Atari screen-size shape, as well as what types of hidden layers we want to handle the processing that our model undergoes to process the environment/state.
- Building a ReplayMemory function to help train our model from past episodes.

```
[ ] # Deep Q-Network Class
    class DQN(nn.Module):
        def __init__(self, input_shape, num_actions):
            super(DQN, self).__init__()
            num_input_channels = input_shape[0]  # stack_size × channels
            self.conv1 = nn.Conv2d(num_input_channels, 32, kernel_size=8, stride=4)
            self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
            self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
            self.fc = nn.Sequential(
                nn.Linear(64 * 7 * 7, 512),
                nn.ReLU(),
                nn.Linear(512, num_actions)
            )

        def forward(self, x):
            x = F.relu(self.conv1(x))
            x = F.relu(self.conv2(x))
            x = F.relu(self.conv3(x))
            x = x.view(x.size(0), -1)  # Flatten the tensor
            return self.fc(x)

        def predict(self, state):
            """
            Predict the best action for a given state.
            """
            with torch.no_grad():  # Disable gradient computation for inference
                state = torch.tensor(state, dtype=torch.float32).unsqueeze(0)  # Add batch dimension
                q_values = self(state)  # Get Q-values from the model
                action = torch.argmax(q_values).item()  # Select the action with the highest Q-value
            return action

        def save_model(self, filepath, optimizer=None):
            """
            Saves the model's state dict and optionally the optimizer's state dict.
            """
            torch.save({
                'model_state_dict': self.state_dict(),
                'optimizer_state_dict': optimizer.state_dict() if optimizer else None
            }, filepath)
            print(f"Model saved to {filepath}")

    class ReplayMemory:
        def __init__(self, capacity):
            self.memory = deque(maxlen=capacity)

        def remember(self, state, action, reward, next_state, done):
            self.memory.append((state, action, reward, next_state, done))

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

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

Necessary Functions to run our DQN-Model

- An environment in which our model will be training in and will be evaluated in.
- A reward structure, to ensure you get the model is properly rewarded for doing “good” tasks and is rewarded negatively for performing “bad” tasks.
- Preprocess Frame and Frame Stack functions, to process the frames and then to stack them properly for our DQN model to perform.
- A Training function to the basic Q-value calculations to help move and train our model properly.
- An evaluation loop with some sort of video recording implementation, so we can visually see how well our model is or isn’t doing.

Overwriting to make our own Custom Mspacman Environment from Gymnasium

If we don’t overwrite the original Gymnasium Environment, we are at the mercy of their base code and their base structure that they have set up for a basic model.

Overwriting this, allows more and more control into our hands, allowing us to insert more order where it wouldn’t exist in the base environment.

```
[ ] class CustomPacmanEnv(gym.Env):
        def __init__(self, render_mode=None):
            super().__init__()  # Initialize the base Gym environment
            self.render_mode = render_mode
            self.observation_space = gym.spaces.Box(low=0, high=255, shape=(210, 160, 3), dtype=np.uint8)
            self.action_space = gym.spaces.Discrete(5)  # Example action space for Pacman, adjust as needed

            # Load the base environment from the ALE/Pacman-v5
            self.base_env = gym.make("MsPacman-v4", render_mode=render_mode)
            self.event = None  # Initialize the event attribute
            self.previous_nearest_reward_distance = float('inf')  # Initialize previous distance as infinity
            self.previous_pacman_position = None  # Initialize previous position as None
            self.steps_since_last_event = 0  # Counter for steps since the last event
            self.max_inactive_steps = 100  # Threshold for inactivity

        def reset(self):
            # Call the reset method of the base environment
            state, info = self.base_env.reset()
            self.event = None  # Reset the event attribute
            self.previous_nearest_reward_distance = float('inf')  # Reset the previous distance
            self.previous_pacman_position = self.get_pacman_position(state)  # Set the initial position
            self.steps_since_last_event = 0  # Reset inactivity counter
            return state, info
```

Overwriting to make our own Custom Mspacman Environment from Gymnasium

```
def step(self, action):
    # Take the action and get the resulting state
    next_state, _, done, info = super().step(action)

    # Calculate custom reward
    reward = custom_reward(self)

    return next_state, reward, done, info
```

To our left, was the original step function for our gymnasium environment.

```
def step(self, action):
    next_state, _, done, truncated, info = self.base_env.step(action)

    # Detect events
    self.event = self.detect_event(next_state, info)
    current_nearest_reward_distance = self.calculate_nearest_reward_distance(next_state)

    # Update inactivity counter
    if self.event:  # Reset counter if an event occurs
        self.steps_since_last_event = 0
    else:  # Increment counter if no event
        self.steps_since_last_event += 1

    # Check for reward collection and update the state
    pacman_position = self.get_pacman_position(next_state)
    if pacman_position is not None:
        # Check if Pacman is on a reward
        reward_positions = self.get_reward_positions(next_state)
        if any(np.array_equal(pacman_position, reward_pos) for reward_pos in reward_positions):
            # Remove the reward from the state
            next_state[pacman_position[0], pacman_position[1]] = 0

    # Calculate the number of remaining rewards
    remaining_rewards = len(self.get_reward_positions(next_state))
    info["rewards_remaining"] = remaining_rewards  # Add this to the info dictionary

    # Calculate custom reward
    current_nearest_reward_distance = self.calculate_nearest_reward_distance(next_state)
    reward = self.custom_reward(current_nearest_reward_distance, next_state)
```

This is our new updated step function, allowing us to implement methods to detect inactivity (running into a wall), detecting whether we are closer or further away from our reward than previously, and anything else you think you would need on a step-by-step basis

Reward Function

```
[ ]  def custom_reward(self):
         reward = 0
         if self.event == 'eat_pellet':
             reward += 1
         elif self.event == 'eat_power_pellet':
             reward += 5
         elif self.event == 'eat_ghost':
             reward += 10
         elif self.event == 'collect_fruit':
             reward += 50
         elif self.event == 'clear_maze':
             reward += 100
         elif self.event == 'caught_by_ghost':
             reward -= 50
         reward -= 0.1  # Small penalty for each time step.
         return reward
```

This was the general reward function given to us on Gymnasium for the Mspacman-v4 game.

It doesn't have enough nuance in itself to properly train the model effectively.

We need to add sets of rules to help guide our model in the direction we want, so our model doesn't need several 10s of thousands of episodes to begin to look functional.

Reward Function

```python
def custom_reward(self, current_distance, state):
    """
    Assign rewards based on the current event, distance to nearest reward, and movement.
    """
    reward = 0

    # Event-based rewards
    if self.event == "eat_pellet":
        reward += 75
    elif self.event == "eat_power_pellet":
        reward += 200
    elif self.event == "eat_ghost":
        reward += 200
    elif self.event == "collect_fruit":
        reward += 500
    elif self.event == "clear_maze":
        reward += 10000
    elif self.event == "caught_by_ghost":
        reward -= 500

    # Closeness-based reward
    if current_distance != float('inf'):
        distance_change = self.previous_nearest_reward_distance - current_distance
        if distance_change > 0:  # Pacman moved closer to the reward
            reward += 8.0
        elif distance_change < 0:  # Pacman moved further from the reward
            reward -= 5.0  # Penalize slightly for moving away

    # Penalize lack of movement
    if self.previous_pacman_position is not None:
        if np.array_equal(self.previous_pacman_position, self.get_pacman_position(state)):
            reward -= 2

    # Time step penalty
    reward -= 0.1  # Small penalty for each time step

    return reward
```

This reward function will act as a huge hyperparameter, to reinforce your model to prioritize the set of rules you give it via +rewards or -rewards.

This is generally called upon when your model takes the .step() action.

Preprocess and Stack Frame Functions

```
[36] # Preprocess frame using PyTorch transforms (as you described)
     def preprocess_frame(frame):
         transform = T.Compose([
             T.ToPILImage(),
             T.Grayscale(num_output_channels=1),  # Ensure single channel
             T.Resize((84, 84)),
             T.ToTensor()
         ])
         return transform(frame)  # Shape: [1, 84, 84]

     # Stack multiple frames
     def stack_frames(frames, new_frame, stack_size=4):
         if frames is None:
             frames = []  # Initialize if None
         frames.append(new_frame)
         if len(frames) > stack_size:
             frames = frames[-stack_size:]  # Keep the most recent `stack_size` frames
         elif len(frames) < stack_size:
             while len(frames) < stack_size:
                 frames.append(new_frame)  # Pad with the current frame
         stacked_frames = torch.cat(frames, dim=0)  # Concatenate along the channel dim
         return stacked_frames, frames
```

This code preprocesses game frames by converting them to grayscale, resizing them to 84x84, and normalizing them into a tensor for efficient model input.

It also manages a stack of consecutive frames to capture temporal information, essential for understanding dynamics in the environment.

By maintaining and padding a fixed number of frames, it ensures consistent input dimensions while emphasizing recent observations.

Training Function

This function trains our DQN model by sampling a batch of experiences from memory, computing the target Q-values using the target network, and minimizing the loss between predicted and target Q-values.

It leverages gradient descent to adjust the model's weights, ensuring better alignment with the optimal action-value function over time.

Key steps include tensor preparation, target computation, and backpropagation using the Adam optimizer.

```
[ ] # Training Function
    def train_dqn(dqn_model, target_model, memory, optimizer, batch_size, gamma, loss_fn):
        if len(memory) < batch_size:
            return  # Skip if there aren't enough samples

        # Use the `sample` method of ReplayMemory
        minibatch = memory.sample(batch_size)
        states, actions, rewards, next_states, dones = zip(*minibatch)

        # Convert to tensors
        states = torch.cat([s.unsqueeze(0) for s in states])  # [batch_size, 4, 84, 84]
        next_states = torch.cat([ns.unsqueeze(0) for ns in next_states])  # [batch_size, 4, 84, 84]
        rewards = torch.tensor(rewards, dtype=torch.float32)  # [batch_size]
        dones = torch.tensor(dones, dtype=torch.bool)  # [batch_size]
        actions = torch.tensor(actions).view(-1, 1)  # [batch_size, 1]

        # Calculate target Q-values
        with torch.no_grad():
            max_next_q_values = target_model(next_states).max(1)[0]  # Shape: [batch_size]
            targets = rewards + (1 - dones.float()) * gamma * max_next_q_values

        # Predicted Q-values for the actions taken
        predicted_q_values = dqn_model(states)  # Shape: [batch_size, n_actions]
        selected_q_values = predicted_q_values.gather(1, actions).squeeze(1)  # Shape: [batch_size]
        # Compute loss
        optimizer.zero_grad()
        loss = loss_fn(selected_q_values, targets)
        loss.backward()
        optimizer.step()
```

Evaluation Loop

```python
for episode in range(n_eval_episodes):
    state, _ = env.reset()
    state = preprocess_frame(state)  # Preprocess the initial state
    stacked_state, frame_stack = stack_frames(None, state, stack_size=4)  # Stack the frames

    done = False
    total_reward = 0

    while not done:
        # Model prediction
        q_values = model(torch.FloatTensor(stacked_state).unsqueeze(0).to(device))
        q_values = q_values.detach().cpu().numpy()

        # Epsilon-greedy action selection
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(q_values)

        # Take action in the environment
        next_state, reward, done, truncated, info = env.step(action)
        done = done or truncated

        # Preprocess and stack the next state
        next_state = preprocess_frame(next_state)
        stacked_next_state, frame_stack = stack_frames(frame_stack, next_state, stack_size=4)

        # Update the current state
        stacked_state = stacked_next_state
        total_reward += reward

    print(f"Episode {episode+1}/{n_eval_episodes}, Total Reward: {total_reward}")
    total_test_rewards.append(total_reward)
env.close()

# Compute and print evaluation metrics
average_reward = np.mean(total_test_rewards)
print(f"Average reward over {n_eval_episodes} episodes: {average_reward}")
```

The evaluation loop and the actually training loop are almost entire similar, with the training loop including the remember function as well as the training function.

This effectively allows our agent to “play” in the environment and then we watch back on whichever episode we want that records.

Agent playing MsPacman

Gets a score of 670. Explores bottom left and top left!