5.6.2 Hands On

Tags

Deep Q-Learning Implementation

Define Q Trainer Part 2

1. Initialising Q Trainer class
    a. setting the learning rate for the optimizer
    b. gamma value that is the discount rate used in bellman equation
    c. initialising the adam optimizer for updation of weight and biases
    d. criterion is the mean squared loss function.
2. Train step function
    a. As you know that pytroch work only on tensors, so we are converting all the input to tensors.
    b. As discussed above we had a short memory training then we would only pass one value of state, action, reward, move so we need to convert them into a vector, so we had used unsqueezed function.
    c. Get the state from the model and calculate the new Q value using the below formula:

    $$Q_new = reward + gamma^* \\ max(next\_predicted\ Qvalue)$$

    d. calculate the mean square error between the new Q value and previous Q value and backpropagate that loss for weight updation.

```
class QTrainer:
  def __init__(self, model, lr, gamma):
    # learning rate for optimizer
    self.lr = lr
    # discount rate
    self.gamma = gamma
```

```
    # linear NN defined above
    self.model = model
    # optimizer for weight and blases updation
    self.optimer = optim.Adam(model.parameters(), lr = self.lr)
    # Mean square error loss function
    self.criterion = nn.MSELoss()

  def train_step(self, state, action, reward, next_state, done):
    state = troch.tensor(state, dtype=troch.float)
    next_state = troch.tensor(next_state, dtype=troch.float)
    action = troch.tensor(action, dtype=troch.long)
    reward = troch.tensor(reward, dtype=troch.float)

    # only one parameter to train, hence convert to tuple of shape(1,x)
    if len.state.shape) == 1:
      state = troch.unsqueeze(state,0)
      next_state = troch.unsqueeze(next_state, 0)
      action = troch.unsqueeze(action, 0)
      reward = troch.unsqueeze(reward, 0)
      done = (done, )

    # 1. predicted q value with curret state
    pred = self.model(state)
    target = pred.clone()
    for idx in range(len(done)):
      Q_new = reward[idx]
      if not node[idx]:
        Q_new = reward[idx] + self.gamma * troch.max(self.model(next_state[idx]))
      target[idx][troch.argmax(action).items()] = Q_new

    # 2. Q_new = reward + gamma * max(next_prediction Qvalue)
    # pred.clone()
    # preds[argmax(action)] = Q_new
    self.optimizer.zero_grad()
    loss = self.criterion(target, pred)
    loss.backward() # black propagation of loss

    self.optimer.step()
```

Turtlebot3 using pygame version

- Define the agent part I

Define the state (11 states)

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

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

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

    # Move direction
    dir_l,
    dir_r,
    dir_u,
    dir_d,

    # Food location
    game.food.x < game.head.x,  # food left
    game.food.x > game.head.x,  # food right
    game.food.y < game.head.y,  # food up
    game.food.y > game.head.y  # food down
    ]
print(state[0],state[1],state[2],state[3],state[4],state[5],state[6],state[7],state[8],state[9],state[10])
```

Define the action
(3 actions : Straight, Turn right, Turn left)

```python
def _move(self, action):

    """
    The Snake Can only move [Straight, Right, Left]
    """
    clock_wise = [Direction.RIGHT, Direction.DOWN, Direction.LEFT, Direction.UP]
    idx = clock_wise.index(self.direction)

    if np.array_equal(action, [1, 0, 0]):
        new_dir = clock_wise[idx] # no change
    elif np.array_equal(action, [0, 1, 0]):
        next_idx = (idx + 1) % 4
        new_dir = clock_wise[next_idx] # right turn r -> d -> l -> u
    else: # [0, 0, 1]
        next_idx = (idx - 1) % 4
        new_dir = clock_wise[next_idx] # left turn r -> u -> l -> d

    self.direction = new_dir

    x = self.head.x
    y = self.head.y
    if self.direction == Direction.RIGHT:
        x += BLOCK_SIZE
    elif self.direction == Direction.LEFT:
        x -= BLOCK_SIZE
    elif self.direction == Direction.DOWN:
        y += BLOCK_SIZE
    elif self.direction == Direction.UP:
        y -= BLOCK_SIZE

    self.head = Point(x, y)
```

- Define the agent part II

```python
def get_action(self, state):
    # random moves: tradeoff exploration / exploitation
    self.epsilon = 80 - self.n_games
    final_move = [0,0,0]
    if random.randint(0, 200) < self.epsilon:
        move = random.randint(0, 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item()
        final_move[move] = 1

    return final_move
```

➢ Define the gradient policy to determine whether to be exploration or exploitation.

Refrences

- Richard S. Sutton and Andrew G. Garto Reinforcement Learning: An Introduction. 2020
- Sudharsan Ravichandrian. Hans-On Reinforcement Learning with python 2018