Audrey Fuller and Thalia Kennedy
April 14th, 2024
CSCI 331 - Intro to Artificial Intelligence
Prof. Homan

Project 3 Report

Activity 2:

Consider the problem of constructing (not solving) crossword puzzles fitting words into a rectangular grid. The grid, which is given as part of the problem, specifies which squares are blank and which are shaded. Assume that a list of words (i.e., a dictionary) is provided and that the task is to fill in the blank squares by using any subset of the list. Formulate this problem precisely in two ways:
1. [15 points] As a tree search problem. Choose an appropriate search algorithm and specify a heuristic function. Is it better to fill in blanks one letter at a time or one word at a time?

As a tree search problem, depth-first search is the best method for constructing the puzzle. For DFS, we would want to fill in the blanks one word at a time because it would better optimize the construction. The initial state would be the empty crossword puzzle, and from there it would expand into a tree, filling in words where the letter placements allows. With depth-first search, if no possible solutions are available from one internal node, then we can go back up a level in the tree and try from a different node. A heuristic function that could exist in this solution would be to give the words different values. For example, a less common word (ersatz) would be worth more than a common word (thanks). While this would not optimize the creation of the puzzle, it would allow for more interesting/more complex puzzle results.

2. [15 points] As a constraint satisfaction problem. Should the variables be words or letters?
Which formulation do you think will be better? Why?

As a constraint satisfaction problem, the choice of words or letters are both feasible solutions. However, for this example, the variables will be letters. The domain will be the list of words given (either in a dictionary or something else), and the constraint will be that the letters must overlap/agree with one another in the appropriate spots and that a combination of letters in a row or column makes a word that is in the domain.

I believe that the constraint satisfaction problem formulation is a better solution because it is going to be fewer steps than depth first search. With the constraints set in place for the CSP, it is never going to go too far down a path that is invalid, while DFS can go down an invalid path for a bit before realizing it needs to return to a higher level in the tree. Depth-first search is the simpler solution, but I believe that setting up this problem as a constraint satisfaction problem will give more optimal results.

Activity 3:

[20 points] For the 4×3 world shown in Figure 15.4, (or alternately, page #4 of week 7 slides, "chapter 16-17") calculate which squares can be reached from (1,1) by the action sequence [Right,Right,Right,Up,Up] and with what probabilities.

What squares can be reached: (1,1), (1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3, 1), (3,3), (3,4)

	Right	Right	Right	Up	Up
1,1	0.1	0.1	0.019	0.019	0.01658
1,2	0.9	0.09	0.171	0.1468	0.12834
1,3		0.81	0.081	0.09	0.02278
1,4			0.729	0.081	0.0171
2,1				0.0152	0.01824
2,2	-	-	-	-	-
2,3				0.0648	0.07848
2,4				0.5832	0.65448
3,1					0.01216
3,2					
3,3					0.05184
3,4					

Calculations:
RR
(0.1*0.1)+(0.1*0.9) = 0.1
(0.1*0.9) = 0.09
(0.9*0.9) = 0.81
–
RRR
(0.1*0.1)+(0.09*0.1) = 0.019
(0.1*0.9)+(0.81*0.1) = 0.171
(0.09*0.9) = 0.081
(0.81*0.9) = 0.729
–
RRRU
(0.019*0.1)+(0.171*0.1) = 0.019
(0.019*0.1)+(0.081*0.1)+(0.171*0.8) = .1468
(0.171*0.1)+(0.729*0.1) = 0.09

(0.081*0.1)+(0.729*0.1) = 0.081
(0.019*0.8) = 0.0152
(0.081*0.8) = 0.0648
(0.729*0.8) = 0.5832
–
RRRUU
(0.019*0.1)+(0.1468*0.1) = 0.01658
(0.019*0.1)+(0.1468*0.8)+(0.09*0.1) = 0.12834
(0.1468*0.1)+(0.081*0.1) = 0.02278
(0.09*0.1)+(0.081*0.1) = 0.0171
(0.019*0.8)+(0.0152*0.2) = 0.01824
(0.09*0.8)+(0.0648*0.1) = 0.07848
(0.081*0.8)+(0.5832*1)+(0.0648*0.1) = 0.65448
(0.0152*0.8) = 0.01216
(0.0648*0.8) = 0.05184

Activity 4:

Short 1-2 page writeup explains approach and how to run the centipede model.

Our implementation of the Centipede Machine Learning code was decently straightforward, following the hints of the assignment closely and only making minor modifications outside of the neural net code (DQN). For simplicity's sake, I'll be going through our code from top to bottom, elaborating on the differences from the starter code.

For our imports and setup, we stick with using the provided libraries such as Gym, which provides the Centipede environment, along with other libraries like PyTorch for deep learning, and Matplotlib for visualization. One change that took surprisingly long to get working was the integration of NVIDIA CUDA cores. After installing several different cudatoolkit library versions (as only one is compatible with our current version of pytorch), we were finally able to use the computer's full GPU for training the neural network which sped up the process significantly.

The ReplayMemory class is defined to store the agent's experiences (transitions) for experience replay. We did not modify this class for the centipede implementation.

The core of the reinforcement learning algorithm lies in the Deep Q-Network (DQN class), which defines the neural network architecture. This class implements a typical convolutional neural network (CNN) architecture, consisting of convolutional layers to process the stacked frames of the game state, followed by fully connected layers to map the flattened output to Q-values for each action. For our implementation we followed the *Reinforcement Learning: Deep Q-Learning with Atari games* article by Cheng Xi Tsou fairly closely, using two 2D convolutional layers and two linear layers mapping the observations to the actions. We also used their implementation of the built-in Gym wrappers, preprocessing our model to reduce the size of the screen, skip frames and make it grayscale (and thus reduce the amount of data being processed by our learning network). We also utilized framestacking to pass four frames at once into our model, so the model could theoretically learn based off of the movement (speed and direction) of the centipede and spiders.

We didn't modify the hyperparameters or argument parser much, as they seemed to work fine for the balancing pole example. The one thing we did change was changing the model being loaded into the policy net as the –load argument instead of the –save argument, which seemed to be the original intent of the code. Using this line, the policy network weights are loaded from a pre-trained model file, allowing for transfer learning or continuing training from a previously saved checkpoint.

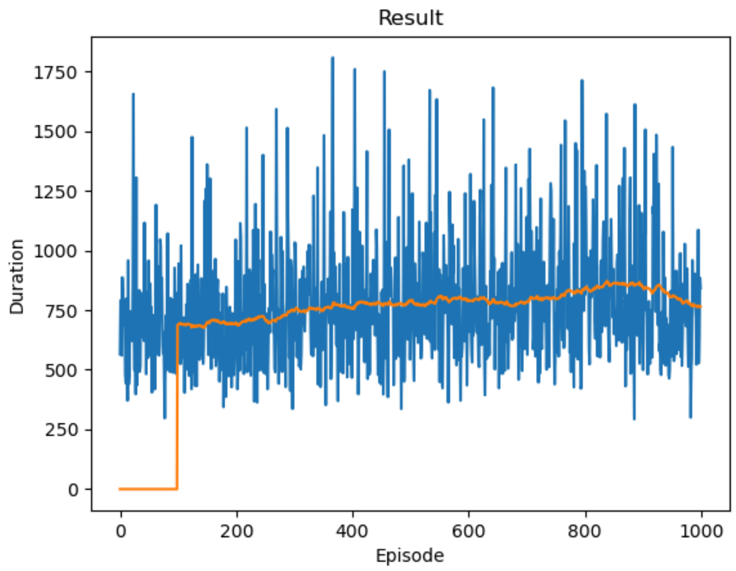
The select_action function implements epsilon-greedy action selection, choosing a random action with probability epsilon or selecting the action with the highest Q-value from the policy network. It was not modified in our centipede code. The plot_duration function also works the same, but automatically saves a png of the most recently finished plot afterwards for referencing post-program run.

The train_model function implements the main training loop. It iterates over a fixed number of episodes, with each episode consisting of multiple steps. In each step, the agent selects an action, interacts with the environment, stores the transition in replay memory, and updates the Q-network parameters using Q-learning with experience replay. The major change implemented here was a negative penalty applied to the reward for when the model loses a life. The intent of this was to discourage the model from doing actions that could result in it dying, thus increasing its survivability and total score. Though the effectiveness of this has yet to be seen in its performance.

The optimize_model function was not changed. It performs the Q-learning update step using a mini-batch of transitions sampled from the replay memory. It calculates the loss between the predicted Q-values and the target Q-values, and then backpropagates the loss to update the network weights.

The run_model function runs the trained agent in the environment for evaluation. It uses the same logic as the train_model function, and incorporates the same changes. While the original program did not include a main function, for good practice I added the model training, evaluation and visualization function calls into it. I also increased the step count of run_model so it won't cut off before finishing. To run the model using the included centipede.pytorch policy net, in the directory of the code run *python centipede_rl.py -l ./centipede.pytorch*, after making sure the train_model and policy net saving function is commented out.

Performance After 1000 Runs:


Result
Duration
Episode