Plant Finding, Evaluating First, Finally Eating Robot

Plant Classification and Finding Clusters

Tim Ledlie ledlie@fas
Angela Tseng aatseng@fas

May 20, 2002

1 Introduction

Plant finding, evaluating first, finally eating robot, or PFEFFER, is a robot that is born into a two-dimensional world with a simple fate. He must walk around the world avoiding poisonous plants and searching for nutritive plants to eat and keep him alive. He expends life in walking, and loses and gains life from eating poisonous and nutritive plants. PFEFFER's sole purpose in life is to live for as long as possible. The world is infinite in all directions, and the plants are 6x6 binary pixel arrays. Nutritive plants look similar to one another, as do poisonous plants, but the plants vary from world to world. PFEFFER can only see the contents of the square where he is at any given time.

Our task was to program PFEFFER to solve two problems. First, he must distinguish nutritive plants from poisonous plants, and second, he must move effectively around the world. We used a combination of the Autoclass algorithm and a Neural Network for the plant classification problem, and we created our own Utility Maximization algorithm for movement which finds and exploits nutritive clusters.

With the default game parameters, 100 initial health, +10 for eating a nutritive plant, -10 for a poisonous plant, and -1 for taking a step, PFEFFER lives for an average of 395 rounds (this is an average over 50 games on random boards), and the his longest lifetime was 1400 rounds.

2 Plant Classification

In order to classify the plants, we decided to combine the forces of a Neural Nets (NN) and the Autoclass Algorithm (AC) to address the assorted challenges posed by the plant world, including noisy data, data acquisition, and uncertainty in the world. Our decisions were mostly informed by our initial manual explorations of the world which employed the simple algorithm of looking at 100 random plants, classifying them, and summing the number of "on" pixels in each location for poisonous plants and nutritive plants. The results of this exercise indicated that there seemed to be at least one pixel position in each world which single-handedly determined the classification of the plant; ie, both poisonous and nutritive plants had a position in their pixel array that was almost always one or almost always zero, and that critical location was usually different between the poisonous and nutritive plants.

2.1 Designing the Neural Net

After initial tests of our NN, we found that NNs had the advantage of being very consistent: they always choose the same plant classifications when presented with identical boards. The disadvantages of NNs are that they need supervised training data and many instances compared to other algorithms, so there is a high setup cost associated since Pfeffer would have to walk around blindly eating plants until enough data had been gathered. We also found that when Pfeffer was using a NN as its sole input in deciding whether or not to eat, he was very conservative at the outset if he saw a majority of poisonous plants as the first few training samples. In fact, in order for him to not classify everything as poisonous, the NN needed training data with a majority of nutritive plants.

2.2 Neural Net Structure

Based on the information garnered from the initial exploration, we thought that a NN with no hidden nodes would be effective because the backprop algorithm might be able to detect the "critical node" mentioned above and attach a large enough weight to that input. However, we found that the NN with no hidden nodes performed worse than a NN with at least one layer of hidden nodes. The optimal NN we found had one layer of 20 hidden nodes. Even trial NNs with more than one layer of hidden nodes did not out-perform the single hidden layered NN. We suspect that there is probably

a superior network structure out there, but we were just not able to find it and so decided upon the simple single hidden layer. This structure also had the benefit of efficiency – we had only 58 total nodes so we were able to run forwardprop and backprop as often as we liked.

We used distributed encoding to encode the output, though this was not a particularly significant decision since there are only two classes. One advantage was that we wanted to be able to measure "certainty" of the classification which could be achieved by making sure the activation levels of each output node had a difference of at least some threshold. This refinement essentially made PFEFFER more conservative (if the threshold was high) or aggressive (if the threshold was low) in his eating. Various trials showed that it was best if he were always aggressive, however, and that being conservative in mid-life did not increase his lifespan.

2.3 Neural Net Training

We found that the NN did not actually need hundreds of training samples as we first suspected, and in certain favorable conditions, PFEFFER could learn to adequately classify after only one poisonous and one nutritive instance. In order to make the most of our data as early in the game as possible, we re-learned the NN each time we ate a plant. This turned out not to be too time-intensive since we were only running forwardprop and backprop for 10 epochs on 58 nodes. Furthermore, we limited the number of plants in the supervised training set to 200 since adding more plants to the training set beyond this point did not improve PFEFFER's classification accuracy or lifespan and the larger training sets were slowing PFEFFER down considerably. Finally, the learning rate was the last aspect of the NN that we adjusted using trial and error, and we found that the optimal rate was 0.3 which was similar to the optimal rate in the handwritten digit recognition assignment.

We actually found that the NN with the specifications discussed above was on average a 95% accurate classifier in the long run (after having eaten 50+ plants) if it ate and collected every plant it encountered. In practice when PFEFFER was not allowed to live forever and when he had to only eat the plants he decided were nutritive, he was much less accurate. If he ate many poisonous plants in the early stages of the game, he typically wouldn't live long enough to reap the long term benefits of his well-trained network. PFEFFER needed a different algorithm to make him more effective in the beginning of his life.

2.4 Autoclass

In order to counteract some of the fundamental problems with the NN - the need for a large collection of supervised training sample and the danger of failing miserably if too many poisonous plants were eaten before PFEFFER was able train his NN - we used another decision algorithm to guide his exploration and collection of supervised training instances. For this task, we chose the Autoclass algorithm because it can classify unlabeled training data and because of its efficiency. We found that for small training sets (typically fewer than 10 samples), the NN needs to have a majority of nutritive instances; otherwise, it is too conservative and starves. So we use AC to aid decisions at the outset using the following algorithm:

1. Explore (without eating) the first 6 plants and eat one to build a training set for AC.
2. Vote using AC and NN whether to eat the next plant (if either votes to eat, then eat).
3. Repeat 2 until the labeled training set has more nutritive than poisonous instances.
4. Use NN as the exclusive classifier unless PFEFFER ends up eating more poisonous than nutritive plants, in which case return to step 2.

Basically AC kicks in whenever NN is failing or is expected to fail. We found that the OR vote (rather than an AND vote) between AC and NN was preferable since it would be better to be more aggressive when PFEFFER is exploring.

We also tried using AC as the exclusive classifier but found that although AC was often a very accurate classifier (often slightly better than NN) and allowed PFEFFER to live longer on certain occasions, it was inevitably an inconsistent performer. The inconsistency is due to the use of random initial parameter values which seem to tend to converge at local minima. This randomness led to wildly fluctuating results and we decided that AC was too risky as an exclusive classifier, even if did outperform NN on occasion.

3 Movement

Not only must PFEFFER evaluate the plants that he sees, but he also must navigate efficiently around his environment. We found from manually exploring a number of worlds that there tend to be clusters of plants. If there is

a plant in a square, there is more likely to be another plant nearby, and also likely that plants nearby are of the same type. Therefore, we want PFEFFER to find and exploit clusters of nutritive plants and avoid poisonous clusters. We would also prefer that PFEFFER revisit explored territory as little as possible, as this is usually a waste of life.

3.1 Possible Algorithms

We had a number of ideas for approaches to the movement problem. The most obvious algorithm seemed to be a Markov Decision Process. We have a situation where we would like to know the best action to take given a state in the world. A state is what contents we know to be in some region around us, and the actions are moving in any of the four directions. Slightly more complicated would be the reward model, since the state space is very complex. But more importantly, MDPs assume knowledge of transition probabilities between states. These probabilities, however, are what we would like PFEFFER to learn. Therefore, the problem would really be one of learning some of the parameters to an MDP.

A neural net was another possible solution for the movement problem. The input to the net would be the contents of the squares in some region around PFEFFER, and the output would be the direction to move in. A drawback to this approach is coming up with training data, and labeling the datum with utilities. This task would be difficult to automate, and of course the neural net would require a lot of samples to train it effectively.

Another way to use a neural net is as a predictor of what is in the squares we have yet to visit. This seems a more feasible use of a neural net, but alone does not solve the problem of where to move next.

3.2 Utility Maximization Movement Algorithm

3.2.1 General Description

We decided to consider movement a utility maximization problem. We look in each of the four possible directions of movement and calculate some utility of that direction. We determine the utility of each direction, and choose to step in the one of highest utility. Here is a relative ordering of the utility of square contents:

High Utility	Nutritive plants
	Unexplored territory
	Explored territory with no plants
Low Utility	Poisonous plants

Already we see that PFEFFER needs to have a memory of the world that it has seen. This idea of "world memory" is described in the implementation section. We simply keep track of the types of plants, if any, seen in each square traveled, and consider all other spaces unexplored.

To calculate the utility of moving up, we find the weighted sum of the utilities in PFEFFER's field of view in the up direction. The contribution of the utility of a square is stronger for squares that are closer, weaker for more distant squares. Here is a diagram of PFEFFER's field of view, and what he sees when he's considering the utility of moving up:

/	N	N				/
/	/		/	P	N	/
/	/				/	/
x	x	x	**Pf**	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x

The depth of vision is a variable in our program, and we found a distance of 5 from PFEFFER in each direction to work well, with the utility contribution of squares decreasing approximately linearly as the squares get farther away.

3.2.2 Refinements

PFEFFER would like to avoid revisiting previously explored territory, since he would have already eaten any nutritive plants that were there. But if there is high utility in moving in a particular direction, PFEFFER would like to find a way to get there while going through unexplored territory, even if it means a slightly longer route. Therefore, if we sense high utility in moving in a direction but the adjacent square has already been explored, PFEFFER instead choose to move in a direction *perpendicular* to the optimal, assuming that he will probably be able to turn in the optimal direction after a few sidesteps. We found this strategy to be extremely effective. If the

optimal and both perpendicular directions have already been visited, then PFEFFER will chose to move in the direction opposite the optimal, to avoid getting sucked into a cluster of already-eaten nutritive plants. Finally, if PFEFFER finds himself "stranded," or that the squares in all directions have been previously visited, PFEFFER goes in the direction that has the closest unexplored square.

From manually exploring a number of worlds, we found that in general, the farther you get from the origin, the less likely it is to find a plant, and the more likely that the plants are poisonous. Therefore, we wanted PFEFFER to tend towards the origin of the world, so as not to wander off. We accomplished this by simply adding some utility to moving in a direction closer to the origin, proportional to the distance from the origin, and found this effective in keeping PFEFFER from wandering, but not so influential that it stopped him from finding nutritive clusters.

3.2.3 Spiral Walking vs. Utility Maximization Walking

Although simple, we found that walking in a spiral from the origin often performed well, so much so that we did a comparison of spiral walking with PFEFFER's specialized utility maximizing walking system. For 39 different random seeds, we compared the lifespan of the spiral walker against that of our specialized walking system. In the table below, UM is our utility maximization movement algorithm, and SP is spiraling.

seed	UM	SP		seed	UM	SP		seed	UM	SP
1002	89	159		1015	239	409		1028	259	45
1003	919	329		1016	119	73		1029	80	77
1004	429	109		1017	369	789		1030	309	51
1005	56	69		1018	449	79		1031	509	809
1006	119	59		1019	149	80		1032	339	59
1007	759	99		1020	48	588		1033	199	169
1008	139	609		1021	89	179		1034	99	685
1009	119	79		1022	779	1149		1035	500	79
1010	57	59		1023	60	69		1036	59	49
1011	79	1279		1024	119	59		1037	149	589
1012	1309	56		1025	72	70		1038	68	63
1013	649	969		1026	1129	1159		1039	1099	459
1014	85	79		1027	1309	839		1040	559	109
UM average = 358, SP avg = 327, UM better 6 more times than SP										

We found our walker to perform about 10% better than the spiral walker. However, in running the tests, we saw that the spiral walker was often better at finding plants at the beginning of the game. Therefore, we decided to combine the benefit of spiral walking at the beginning of the game with the benefit of cluster-exploitation later in the game. To this end, PFEFFER will

walk in a spiral for the first 100 rounds unless he is deviated by finding a large utility in some area (a bunch of nutritive plants), at which point he will take on the utility maximization algorithm described earlier.

4 Implementation

Files we use:

- ac.c/h – autoclass code
- game_util.c/h – communicating with the game server
- nn.c/h – neural net code
- player.c – the player, main function
- player_util.c/h – utils for the player
- socklib.c/h – socket networking interface
- WorldMemory.c/h – world mapping and movement code

4.1 Plant Classification

The implementations of both the NN and AC classification algorithms were pretty straightforward and can be found in the nn.c/h and ac.c/h files. Some notable aspects:

- The plant data is stored in the PLANT_STRUCT data struct, with the actual pixels stored in a 1-dimensional array.
- Nutritive = True = 1 and Poisonous = False = 0.
- Variables like the number of epochs, the number of hidden nodes and learning rate are hard-coded into nn.c
- Although the NN is passed a vector of both labeled and unlabeled training data, only the labeled instances are used.
- We generated a vector of random numbers at the beginning of the game for the initial values of AC, and reused these numbers every time we called AC. This made the algorithm more consistent since the tendency to converge at a local minimum is very dependent on the initial parameter values.

Finally, all testing of classification algorithms was done using a spiral walking motion to ensure that differences in performance were not due to the movement algorithm. This is of course not a perfect reflection of actual game-play but allowed for a consistent testing environment where we could evaluate the parameters of the classifier.

4.2 Movement

The WorldMemory object controls the movement of PFEFFER. It stores a map of the world a two-dimensional an array of integers, each integer representing the contents of the square. At first, we thought to store the chartered world as a four-way linked list of nodes, each node representing a square on the board; this would allow for travel arbitrarily far in any direction, limited only by the memory allotted to the program by the operating system. But since we never travel too far in any direction (especially because plants become scarce and almost all poisonous far from the origin), we decided to simply use a two-dimensional array of integers and initialize the center of the array as the origin of PFEFFER. We have found the finite limitation of the array irrelevant.

The player.c file observes the contents of the current square through a call to the server game, and communicates the contents to the WorldMemory object which stores the contents in its map. The WorldMemory object then tells the player a direction to move in, and this process repeats.

There are a number of adjustable parameters for the Utility Maximization movement algorithm which can be found at the beginning of WorldMemory.h.

Note that the walking strategy is non-deterministic for two reasons. First, it is dependent on how PFEFFER classifies the plants, which itself may vary from game to game since the initial AC parameters are random. Second, if PFEFFER is presented with an equal utility of moving in one direction or the other, he will choose from these two directions randomly. We chose this behavior to reduce the chance of PFEFFER exploring the world in an overly unsymmetrical fashion.

5 Conclusion

One area of further study would be to improve the NN accuracy by finding a better fundamental structure. Although there's no way for us to know that one exists which would be applicable to all the different boards, it would be nice to find this optimal NN construction since it would definitely improve the accuracy of the classifier. Ideally there would be a way to learn NN structure.

We feel that the Neural Net structure could have been improved by representing more information about the world in the NN. Specifically, we would liked to have incorporated location into the NN, both location in the world and with respect to other plants. However, we couldn't think of an

straightforward and efficient way to accomplish this.

Generalization over different worlds would also have been desirable. We could not find a way to do this, so we started afresh each game, especially for the neural net and autoclass. However, we did do human learning between games by watching PFEFFER's performance, and programmed that knowledge into the movement algorithm through tweaking its parameters and modifying the algorithm as described in the movement refinements section.