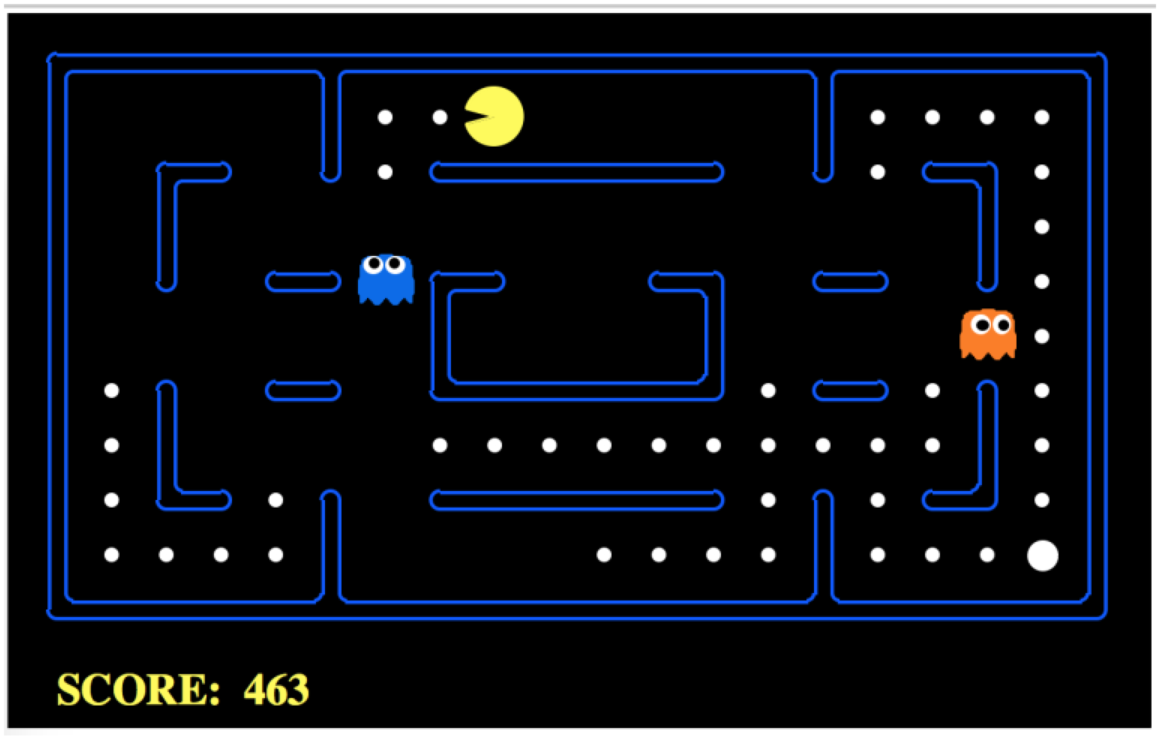


C.I. LAB REPORT

PLAYING PACMAN USING VARIOUS MACHINE LEARNING ALGORITHMS

Dhruv Chadha, Keshav Goyal | Under Guidance of Dr. Aruna Tiwari

INTRODUCTION

In this project, we apply various machine learning algorithms to teach the computer to play Pacman. The major areas of ML algorithms used are -

1. Intelligent Searches
2. Adversarial Techniques
3. Reinforcement Learning
4. Bayesian Learning
5. Classification

TECHNIQUES AND ALGORITHMS USED

In each of the broad areas mentioned above, many techniques algorithms have been applied to learn Pacman.

INTELLIGENT SEARCHES –

First, we start with applying simple DFS, BFS and uniform cost search algorithm. After that, we start involving heuristics and perform A* search. In A* search, we use a special function that helps us in the selection of the next state. This function contains 2 parts –

$$f(X) = g(X) + h(X)$$

So, f(X) measures the estimated cost of getting to the goal state from the current state (h(X)) and the cost of the existing path to it (g(X)). This cost function gives a better result as compared to uniform cost search or DFS/BFS.

ADVERSARIAL TECHNIQUES –

When multiple agents are present in the game, apart from the main player Pacman, we cannot perform ordinary search techniques. The approach needs to be more dynamic.

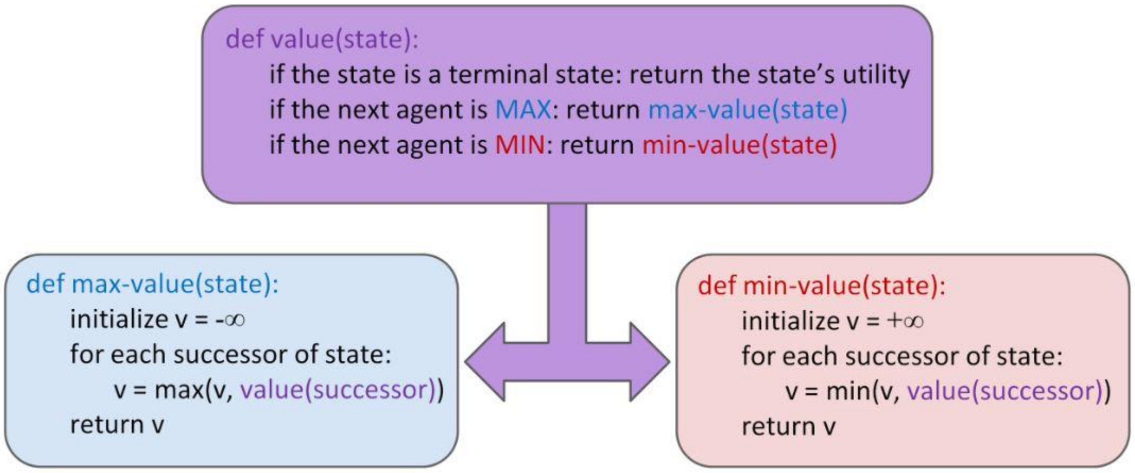
Method 1 –

Use a state evaluator function that scores each state with respect to some features like how far the ghost is or how near is the food. This function will give the best direction to go next every time we take a move.

Method 2 (Minimax) –

It is a decision rule for minimizing the possible loss for a worst case scenario. The maximin value of a player in the highest value that the player can be sure to get without knowing the actions of the other players. It is also the lowest value the other player can force the player to receive, when they know the player's action.

Minimax Algorithm -



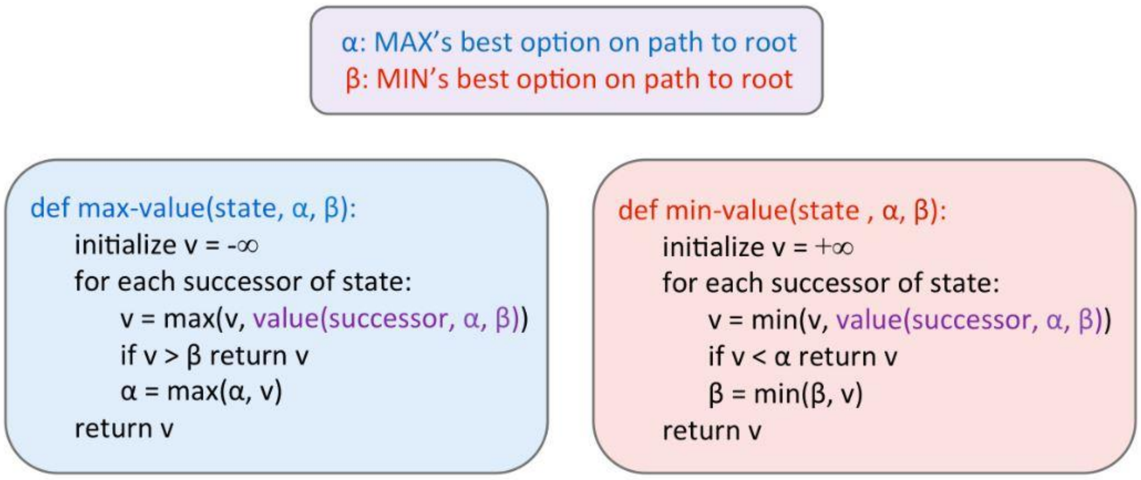
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Alpha-Beta Pruning –

This is a technique that optimize the minimax algorithm. It decreases the number of nodes that are evaluated by the minimax algorithm. The minimax algorithm can be optimised to incorporate alpha-beta pruning by modifying the max-value and min-value functions –



```
α: MAX's best option on path to root
β: MIN's best option on path to root
```

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

Expectimax –

Minimax and Alpha-Beta Pruning are great but they assume that the opponent makes optimal decisions. This is not always the case. In expectimax, the outcome depends on the combination of player's skill and chance elements such as dice rolls. In an expectimax tree, the "chance" nodes are interleaved with the max and min nodes. Instead of taking the max or min of the utility values of the children, chance nodes take a weighted average of the children's utilities. For most games of chance, the child nodes will be equally weighted.

REINFORCEMENT LEARNING –

The game of Pacman can be modelled as a Markov Decision Process. An MDP provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker. At each time step, the process is in some state 's', and the decision maker may choose any action 'a', that is available in state 's1'. The process responds at the next time step by randomly moving into a new state 's2' and giving the decision maker a corresponding reward. The probability that the process moves into the new state 's2' is given by the state transition function $P_a(s1, s2)$. The state transitions of an MDP satisfy Markov Property, which states that given 's1' and 'a', the state 's2' is conditionally independent of all the previous states and actions.

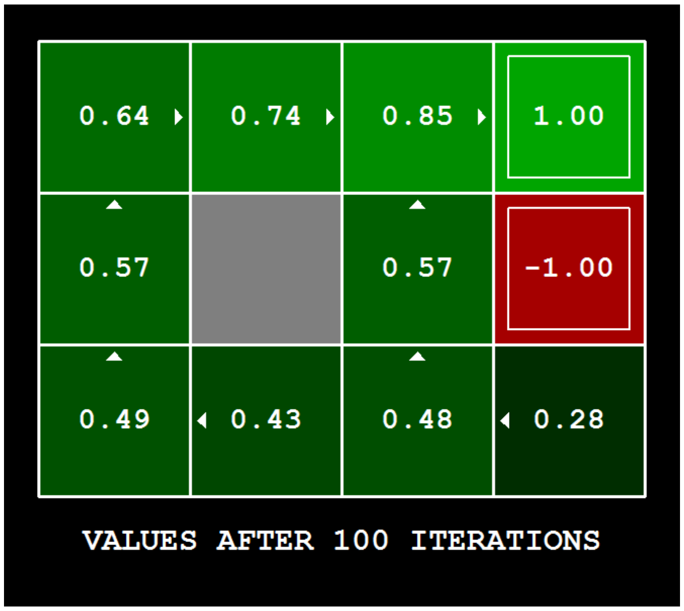
Solving an MDP with given rewards and transition probabilities -

Value Iteration -

$$V_{i+1}(s1) = \max_a \left\{ \sum_{s2} P_a(s1, s2) \times (R_a(s1, s2) + \gamma V_i(s2)) \right\}$$

where $V_i(s)$ is the value of the state 's' at the ith iteration and V_0 is initialized randomly.

But in the real world, where the probabilities and the rewards are unknown, this problem needs to be solved with Q-Learning. It works by learning an action-value function $Q(s, a)$, which ultimately gives the expected utility of a given action 'a', while in a given state 's', and following an optimal policy thereafter.



Q-Learning –

$$Q(s_t, a_t) = (1 - \alpha).Q(s_t, a_t) + \alpha.(r_t + \gamma.max_a Q(s_{t+1}, a))$$

where r_t is the reward observed for the current state 's_t', α is the learning rate, and γ is the discount factor.

After applying the above algorithm, we apply the epsilon greedy strategy, which is nothing but controlling the exploration vs exploitation trade-off. This strategy will choose random actions an epsilon fraction of the time and follow its current best Q-values otherwise.

Approximate Q-Learning –

The above Q-learning approach is computationally expensive and is not scalable. So instead, we perform approximate Q-learning, in which every state has a feature vector associated with it, and $Q(s, a)$ is defined as –

$$Q(s, a) = \sum_{i=1}^{n} w_i \times f_i(s, a)$$

The algorithm proceeds like this –

$$w_i = w_i + \alpha.difference.f_i(s, a)$$

$$difference = \left(r + \gamma.max_{a'} Q(s', a')\right) - Q(s, a)$$

BAYESIAN INFERENCE –

In the game of Pacman, there is a special powerup, which when consumed, allows pacman to start hunting ghosts. To track the ghosts, we use exact and approximate inference using Bayes nets. A Bayesian (belief) network is a probabilistic graphical model that represents a set of variables and their conditional dependencies via a directed acyclic graph. The nodes represent variables (observable quantities or unknown parameters or hypotheses). The Edges represent conditional dependencies. Nodes that aren't connected represent variables that are conditionally independent.

Solving by Inference –

Current Belief that a ghost is in position X at time $t \ = P(X_t|E_{1:t})$

After one time step –

$$P(X_{t+1}|E_{1:t}) = \sum_{X_t} P(X_{t+1}|X_t) \times \ P(X_t|E_{1:t})$$

Then after evidence comes in –

$$P(X_{t+1}|E_{1:t+1}) \propto P(E_{t+1}|X_{t+1}) \times P(X_{t+1}|E_{1:t})$$

After calculating the values of $P(X_{t+1}|E_{1:t+1})$ for all X's, we normalize the probabilities, to make them sum up to 1.

Sometimes, the number of X's are too large to use inference. So, we track samples of X, and not all the values. These samples are called particles. In the memory, we store the particles, not the states.

P(X) is now approximated by the number of particles with value X. Generally, total number of particles N << total number of X's.

Basically, now the probability distribution of X's is nothing but a list of N particles.

Solving by particle filtering –

First, initialize the particles uniformly.

After one time step, each particle moved by sampling its next position from its transition model (which tells the probability of moving to a particular state from the particle's old state).

$$X' = sample(P(X'|X))$$

Then after the evidence comes in –

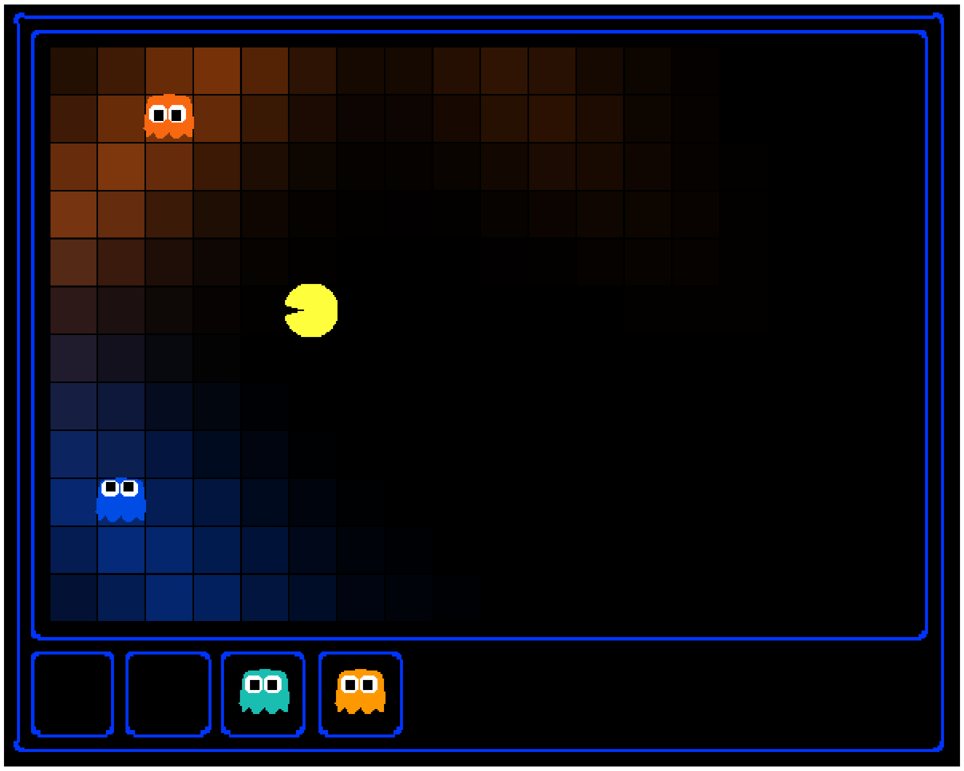
Downweigh each of the samples based on evidence –

$$w(X) = P(E|X)$$

Since number of particles at a particular position tells us about the probability of ghost being in that position –

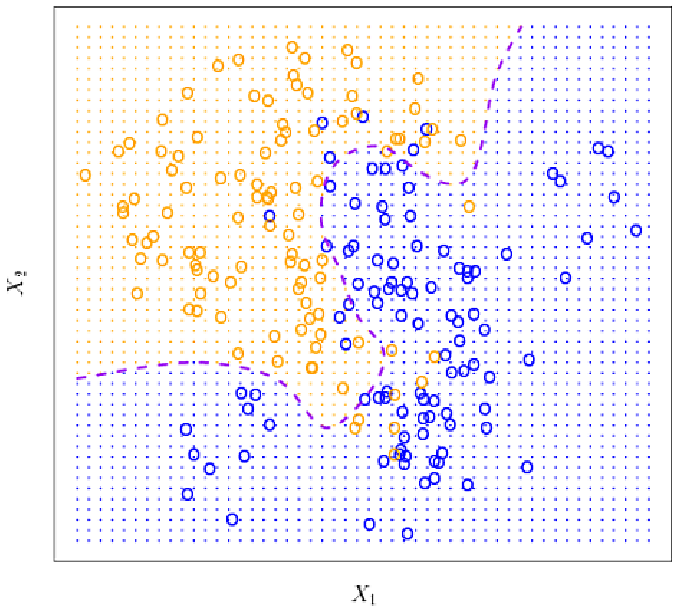
So new distribution $P(X) = \sum_i w(X_i = X)$

The above probabilities do not sum to one. For the next time step, we resample particles N times from the above distribution. This is equivalent to renormalizing the distribution. Now the update is complete for this time step, continue with the next one.



CLASSIFICATION-

Classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known.



Perceptron –
Scoring -
In this project, we have a weight vector w^y corresponding to each class/label y. The perceptron computes the class y whose weight vector is most similar to the input vector.
Let score of a particular class y corresponding to an input vector f be –

$$score(f, y) = \sum_i f_i w_i^y$$

Then we choose the class with the highest score as the predicted label for that data instance.

Learning –
We scan over the data, one instance at a time, and find the instance with the highest score.

$$y' = \underset{y''}{\operatorname{argmax}}\, score(f, y'')$$

We compare y' to the true label y. If $y' = y$, we've gotten the instance correct, and we do nothing. Otherwise, we guessed y' but we should have guessed y. That means that w^y should have scored f higher, and $w^{y'}$ should have scored f lower, in order to prevent this error in the future. We update these two weight vectors accordingly:

$$w^y = w^y + f$$
$$w^{y'} = w^{y'} - f$$

Results-
The method we used will yield an in the range between 40% to 70% and test accuracy between 40% and 70%(with the default 3 iterations).

Difficulties-
One of the problems with the perceptron is that its performance is sensitive to several practical details, such as how many iterations you train it for.

Solutions-
To deal with iteration problem, we used validation set to figure out when to stop training. In this way we can train our network without losing generality.

MIRA –
MIRA is an online learner which is closely related to both the support vector machine and perceptron classifiers. It is designed to learn a set of parameters (vector or matrix) by processing all the given training examples one-by-one and updating the parameters according to each training set. The change of the parameters is kept as small as possible. The algorithm is as follows –

$$y' = \underset{y''}{\operatorname{argmax}}\, score(f, y'')$$
$$w^y = w^y + \gamma f$$
$$w^{y'} = w^{y'} - \gamma f$$
$$\text{where } \gamma = \min(C, \frac{(w^{y'} - w^y)f + 1}{2\|f\|_2^2})$$

For pacman, we can use the above algorithms in the following way –
the data will be states, and the labels for a state will be all legal actions possible from that state. All of the labels share a single weight vector w, and the features extracted are a function of both the state and possible label.
For each action, calculate the score as follows:

$$score(s,a) = w.f(s,a)$$

Then the classifier assigns whichever label receives the highest score –

$$a' = \underset{a''}{\operatorname{argmax}}\, score(f,a'')$$

Training updates occur in much the same way that they do for the standard classifiers. Instead of modifying two separate weight vectors on each update, the weights for the actual and predicted labels, both updates occur on the shared weights as follows:

$$w = w + f(s,a) \qquad \text{for correct action}$$
$$w = w - f(s,a') \qquad \text{for guessed action}$$

We try to adapt the behavior of few types of Pacman agents with the following properties and their respective accuracies (given 1000 test cases of each type) –

Type of Agent	How well our program learnt its behaviour
Stop Agent – (remains still)	100%
Food Agent – (only eats food, doesn't care about environment)	80.5%
Suicide Agent – (only moves towards nearest ghost)	72.5%
Contest Agent – (smartly avoids ghosts and eats food)	82.8%

CONCLUSION –

All the above algorithms allow the programmer to train the computer to play pacman, without the need to explicitly code all the cases. Since there are ghosts and simple searches cannot win the game, hence the demand for such dynamic real time algorithms exist.

- Minimax and other adversarial techniques use foresight to predict the next action.
- In reinforcement learning, Pacman actually learns to play the game, without any prior knowledge of where and when to move. It learns by trying to maximise its reward.
- Bayesian inference allows Pacman to infer where the ghost would probably be at current time, so it can hunt it down (when it has the powerup enabled). When this inference becomes too expensive to compute, we switch to particles that approximate the probability distribution based on where majority of the particles are.
- Classification allows Pacman to make decisions based on weight vector updates. These updates happen in the case of wrong actions, so that next time it makes a better decision.

BIBLIOGRAPHY –

1. http://ai.berkeley.edu/home.html
2. http://ai.berkeley.edu/lecture_videos.html
3. http://ai.berkeley.edu/lecture_slides.html
4. https://en.wikipedia.org/wiki/Q-learning
5. https://en.wikipedia.org/wiki/Minimax
6. https://en.wikipedia.org/wiki/A*_search_algorithm
7. https://en.wikipedia.org/wiki/Statistical_classification
8. https://en.wikipedia.org/wiki/Bayesian_inference