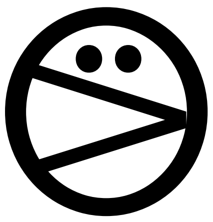
The Pac-Man Report.



Question 1:

Describe the algorithm:

The task is to use value iteration to solve a Markov Decision Process (MDP) for guiding Pac-Man's actions in a grid layout with multiple terminal locations. The objective is to develop an agent that maximizes its reward by finding an optimal policy — moving towards food while avoiding ghosts and slipping 10% of the time.

Key Components of Value Iteration

- **Value Function Initialization (`registerInitialState` method)**: Each state in the Pac-Man grid is initialized to zero, which represents its initial value. During value iteration, these values are updated based on the expected future rewards Pac-Man can obtain by reaching each state.
- **Iterative Updates Using the Bellman Equation (`createQValues` method)**: Call this methods inside the Iterate `registerInitialState`, iterate a given number of times (`self.iterations`) and in each iteration update the value of each state is based on the Bellman equation:

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right)$$

where:

  - P(s' | s, a) is the probability of reaching state s' from state s by taking action a.
  - R(s, a, s') is the reward for transitioning to state s'.
  - γ is the discount factor, which controls how future rewards (*V(s')*) influence the current state's value.

- **`computeQValueFromValues`**: This function calculates the Q-value of a state-action pair by iterating over the list of result of that action and its probabilities, then using the bellman for equation and `self.values` to accumulate the q-values.
- **`computePolicyFromValues`**: This has the purpose of returning the optimal action from the given state. You iterate over each action, then compute its q value from the stored values, then collect the actions with the highest q value in an array, after which you return a random action from the list. We add this element of randomness to prevent Pac-Man from walking in cycles. We can further emphasize this randomness by adding noise in the q value, this is in the **`noiseComputePolicyFromValues`** which will be discussed later.

Assumption:

- Static ghosts
- Reward structure: +510 for eating food. -500 for stepping on ghosts and -1 for time penalty.
- Terminal states are the ghosts and food pellets.
- 10% chance of slipping.

Experiment:

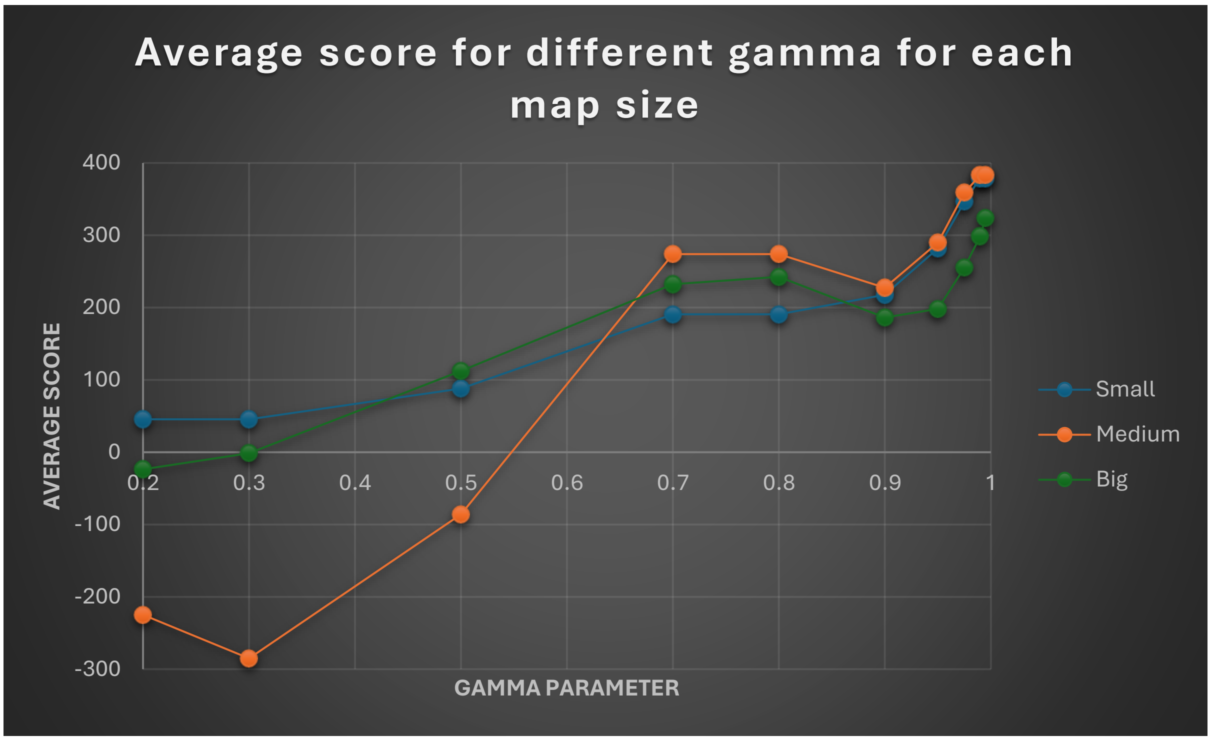


Figure 1. Maps the average score for different gamma values and for different map sizes.

Gamma	Small	Medium	Big
0.2	45.61	-224.754	-23.563
0.3	45.61	-285.048	-1.373
0.5	88.167	-85.992	112.571
0.7	190.748	273.973	232.215
0.8	190.748	273.973	242.744
0.9	217.675	227.448	186.517
0.95	281.819	289.948	198.221
0.975	346.683	358.792	255.215
0.99	378.904	383.352	298.789
0.995	378.904	383.352	323.8875

Table 1. The raw data corresponding to figure 1.

Optimizing parameters [Theory & Experiment]:

During **`createQValues`** you'll have slowly updated the `` `self.values` `` array, who's value of each state diverge ***from*** 0 and converge ***to*** values which represents the usefulness of that state. These values can be skewed in certain directions depending on the choice of discount factor, γ.

A clear way to find the optimal γ parameter, which reduces the value of long-term decisions, we can link it to Pac-Man's personality.

- Case of γ > 1: This would mean Pac-Man would value eating capsule in the future more than eating it sooner in some cases because the value of future rewards, V(s') is multiple by γ, this also doesn't correspond to the fact that Pac-Man loses points each second. Hence $\gamma > 1$ will not be considered.

- Case of $\gamma = 1$: it causes Pac-Man to value all future rewards equally, which can lead to indecisive behavior and infinite loops in some environments. Furthermore, it doesn't reflect the decaying points from the time penalty in the game.
- Case of $0 < \gamma < 1$: As γ increases the more Pac-Man values long-term decisions and the value for short term risks decreases. This is because γ multiplies with future rewards, V(s'), in Bellman's equation. For this reason, γ will also be referred to as Pac-Man's planning horizon

Theoretically: Due to the large disparity of cost between the time_penalty (-1) and aiming for pellets (+510) or slipping onto a ghost (-500) it means that it is more advantageous for Pac-Man to walk longer distances (at most 510 steps!) then to end the game early. This is an argument to choose a γ parameter with a very large value because then Pac-Man is more likely to explore longer paths, with less short-term risks, and more long-term gain since the time penalty is so relatively small.

Furthermore, for sparsely placed pellets and a small γ the influence of those pellets may not be felt on Pac-Man hence he might not ever consider it over walking in circles far away from ghosts. This is another argument for a larger γ parameter.

This is also experimentally visible, since in figure 1 and its corresponding table 1 it can be observed that larger γ parameters outperform smaller ones generally. Hence the choice for γ is 0.995.

A peculiarity in the table 1 is that sometimes the average reward plateaus for some γ, this is seen only in small and medium size maps, like when $0.7 < \gamma < 0.8$ or when $0.99 < \gamma$, this is because a barrier is reached for Pac-Man where much more iterations or a higher γ value is needed for Pac-Man to discover a more unintuitive or longer path which is more optimal.

This is not seen on larger maps because there are still multiple different avenues, and corners Pac-Man could turn to investigate. But I am certain that:

- For very small changes in γ the average score would plateau large maps (i.e. increasing Pac-Man's planning horizon a little bit may not give a change in its policy)
- Also, for $\gamma > \gamma`$ when for a large enough $\gamma`$ (i.e. When Pac-Man's planning horizon is so large, extending it further wouldn't make a difference in his score) the average score would plateau, this will be proven in question 2.

The final peculiarity is when $\gamma = 0.9$ there is a dip in Pac-Man's performance as seen in Figure 1. This is since Pac-Man's planning horizon is extended, by an increase in γ, but not maximized. This setup can sometimes result in Pac-Man taking slightly riskier paths, as it doesn't fully discount shorter routes that may bring it closer to ghosts, even if they eventually lead to food.

This balance causes Pac-Man to take actions that may result in higher penalties (like running into a ghost) compared to higher γ values, where Pac-Man would focus more on safe, long-term paths.

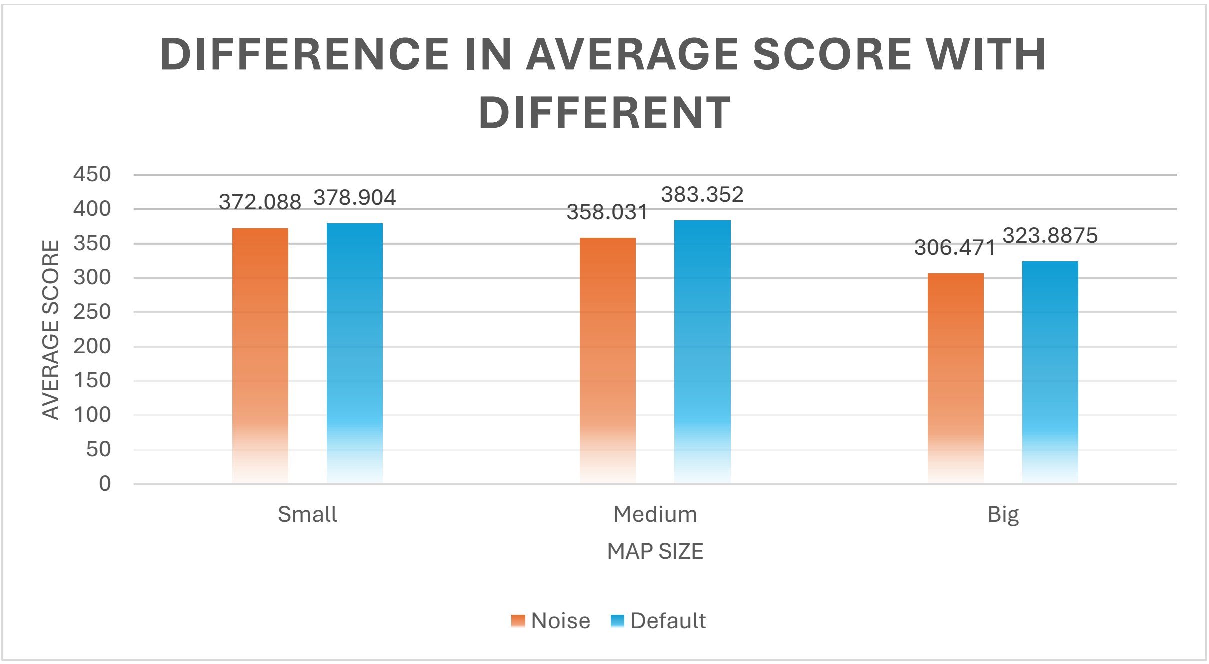


Figure 2. Comparing the difference in average scores when using `computePolicyFromValues` and `noiseComputePolicyFromValues`

Tie-Breaking with Noise:

In the **`noiseComputePolicyFromValues`** method, when finding the optimal action from a state with a given values and current state (like in **`computePolicyFromValues`**) adding small random noise in the q_values of each possible action helps Pac-Man choose between actions with equal Q-values, preventing it from oscillating between identical choices and promoting exploration in ambiguous scenarios. However, as evident from figure 2, these results were the same as in **`computePolicyFromValues`** because in **`noiseComputePolicyFromValues`** very small noise is added to the q_values, this only effects q_values which are the same but in **`computePolicyFromValues`** you randomly pick from an array of actions with the same q_value.

Conclusion:

Optimizing the discount factor γ shapes Pac-Man's strategy by setting the balance between immediate and long-term rewards. High γ values (e.g., 0.995) encourage strategic, long-term paths, while lower values lead to more immediate, potentially risky decisions. Experiments show that high γ values yield higher average rewards, especially on larger maps, because the cost of the time penalty is very small relative to the reward of a pellet so Pac-Man can afford to move around. Hence γ = 0.995 for big maps and γ = 0.99 otherwise. Adding noise to actions with the highest noise helps Pac-Man avoid repetitive choices in states with similar Q-values.

Future changes:

In the future I would experiment with finding a way to make Pac-Man stop early when needed. This could done through a DFS and see if the policy makes Pac-Man walk in circles. In theory this makes sense because Pac-Man wouldn’t pursue pellets when the expected penalty of ghosts is higher than the expected reward of a pellet, so he would walk in circles far away from ghosts. Then by detecting these cycles you can make Pac-Man end early.

Question 2:

Describe the Algorithm

The task for Question 2 is to implement a Q-learning agent with epsilon-greedy action selection to allow Pac-Man to learn an optimal policy through trial and error in a Markov Decision Process (MDP). Unlike value iteration, which precomputes an optimal policy, Q-learning enables the agent to learn directly from interactions with the environment, adjusting its strategy as it encounters new states and rewards. This method adapts Pac-Man's behavior over time to maximize rewards while navigating a grid with food pellets and stationary ghosts as terminal states (assumptions).

Key Components of Q-Learning

- **Q-Value Function Initialization**:
    - The Q-value table (Q_values) is initialized to zero across all states and actions. This table is iteratively updated as the agent explores the environment and learns the value of each state-action pair through experience.
    - The table has the format of Q_value[x_coord][y_coord][action_index], in python this is done with `getQValue` method which returns `Q_values[x][y][action_idx]`
- **Exploration-Exploitation Strategy (`epsilonGreedyActionSelection` method)**:
    - The agent follows an epsilon-greedy strategy, balancing exploration (trying new actions) and exploitation (selecting actions with known high Q-values). With a probability of epsilon, the agent chooses a random action to explore, and otherwise, it chooses the action with the highest Q-value in the current state, as calculated by `computeActionFromQValues`.
        - `computeActionFromQValues`: works by iterating over the legal action and returning the one with the highest Q_value.
- **Q-Learning Update Rule (`update` method)**:
    - This method updates the Q-value for each state-action pair whenever a state transition is observed. The Q-learning update rule is applied as follows:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(R + \gamma \max_{a'} Q(s',a')\right)$$

    - where:
        - Q(s,a) is the current Q-value of the state-action pair (s,a). In python you get this from `self.Q_values[x][y][action_idx]`
        - α is the learning rate, which controls how much new information overrides the old Q-value,
        - R is the observed reward for the transition, and

        - γ is the discount factor, which influences the importance of future rewards. It multiplies to the maximum Q-value of future states and action, hence reduces it's value if between 0 and 1.

- **Optimal Action Calculation (`computeActionFromQValues` method)**:
    - This method selects the best action to take in a given state based on the Q-values. By iterating over the available actions in the state, it identifies the action with the highest Q-value. This action is then selected during exploitation (non-exploratory) steps to maximize Pac-Man's expected rewards.

How This Approach Differs from Value Iteration

Unlike value iteration, which computes an optimal policy before interacting with the environment, Q-learning adjusts the policy **during the game** based on observed transitions and rewards. This allows Pac-Man to learn adaptively, updating its Q-values with each interaction. With epsilon-greedy selection, Pac-Man occasionally explores suboptimal moves, which helps prevent it from settling into non-optimal cycles, especially in dynamic or unknown environments.

Adjusting the γ parameter [Theory & Experiment]:

Methodology of the experiments: We will first adjust the γ parameter without changing the other parameters, then use the best γ parameter to experiment with adjusting the other 2 parameters. We can argue that the choice of γ is mostly independent of the other 2 parameters because γ (the discount factor) fundamentally controls the agent's horizon of planning (as discussed in question 1). This setting determines how much future rewards influence the agent's current decision-making.

By finding the best gamma value first, you establish a foundation for the agent's strategic approach. Once gamma is optimized, you can fine-tune exploration and learning speed with epsilon and alpha parameters.

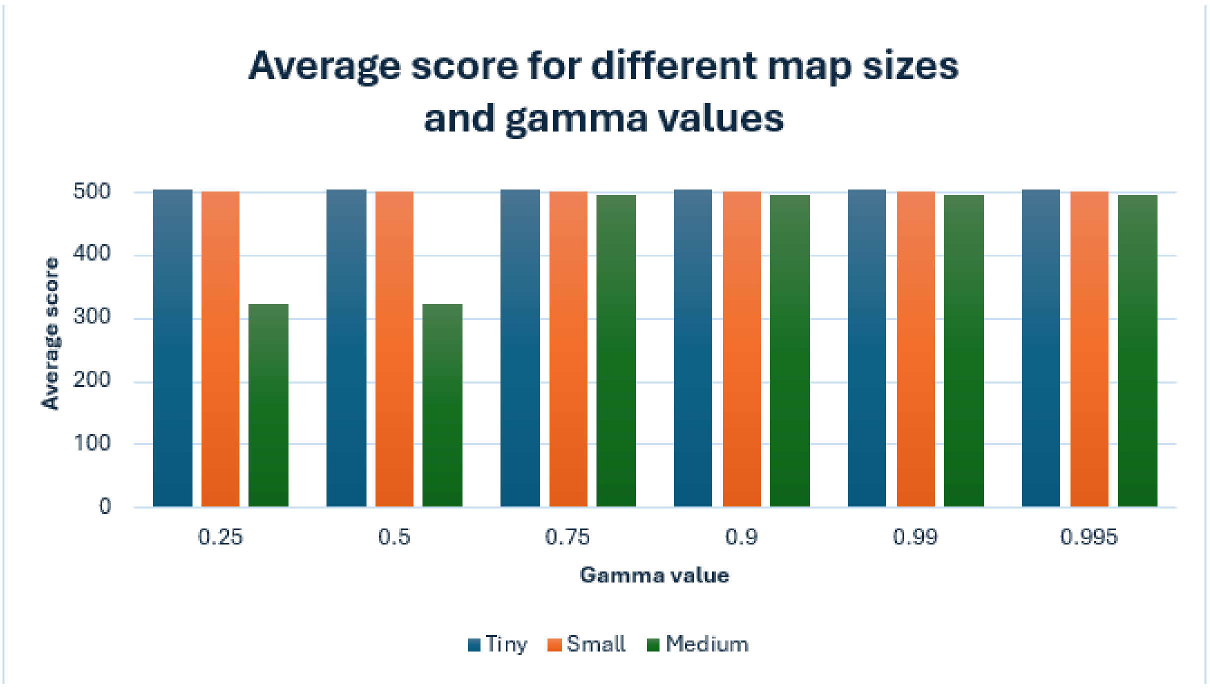


Figure 2. For the **tiny map**, the selected parameters are epsilon = 0.2 and alpha = 0.2. For the **small map**, the chosen parameters are epsilon = 0.3 and alpha = 0.2. Finally, for the **medium map**, the values are set to epsilon = 0.2 and alpha = 0.1.

For the same reasons as in question 1 we will consider when $0 < γ < 1$ and we γ values closer to 1 are more beneficial to Pac-Man. And Figure 2 shows this.

Something unique about figure 2 is the average reward plateaus for all γ values on small and tiny size maps. This occurs for the same reason in table 1 small and medium size maps plateaus for γ >= 0.99, however in figure 2 the plateau occurs early for γ >= 0.25 because of the effectiveness of Q-learning with the current epsilon and alpha parameters.

Furthermore, in Q1 I theoretically concluded that "for γ > γ` when for a large enough γ` the average score would plateau" because Pac-Man's planning horizon is large enough such that extending it further wouldn't increase he's score, this is proven by figure 2 where after a certain γ value the average score plateaus. For a tiny and small map size γ` = 0.25 and for a medium size map γ` = 0.75.

So, to conclude let γ = 0.9 for all map sizes, since according to figure 2 maximises the average score.

Adjusting the α and epsilon parameter [Theory & Experiment]:

The purpose of alpha, the learning rate, is a large alpha prioritizes recent experiences by making quick adjustments to Q-values, while a small alpha smooths learning by making gradual updates, preserving past knowledge.

Case of α > 0.2: Setting the learning rate alpha above 0.2 results in overly aggressive Q-value updates, where Pac-Man heavily weights recent experiences at the expense of stable, accumulated knowledge. This rapid adjustment causes the Q-values to fluctuate dramatically with each new reward, making it difficult for Pac-Man to converge on a stable policy. High alpha values lead to erratic behavior, where Pac-Man may keep changing its preferred actions without settling on a reliable path.

By limiting alpha to 0.1 or 0.2, learning becomes smoother and more controlled, allowing the Q-values to converge steadily, which results in a more reliable and consistent policy over time.

This is reflected in table 2, where average scores are close to perfect (the scores are near 500) when alpha is 0.1 or 0.2.

Epsilon decides how often Pac-Man takes risks and explores outside of his comfort zone.

Case of epsilon > 0.3: When epsilon is set above 0.3, Pac-Man engages in excessive exploration, meaning it takes random actions too frequently, even after identifying effective strategies. This high level of exploration hinders Pac-Man from consistently exploiting the optimal paths it has learned, as it keeps testing alternative actions rather than settling on proven routes. Consequently, Pac-Man may repeatedly take suboptimal paths or encounter ghosts, lowering its performance and stability.

By keeping epsilon between 0.1 and 0.3, Pac-Man can explore early in training but increasingly exploit learned strategies, achieving a balance that promotes both learning and stability.

This is reflected in table 2, where average scores are close to perfect (the scores are near 500) when epsilon is between 0.1 and 0.3.

There are 3 peculiarities in table 1, for (epsilon, α) = (0.1, 0.1), (0.2, 0.1), and (0.3, 0.1) the average score is lower. I noticed that the peculiarities stemmed from Pac-Man performing near perfectly on all the maps except he failed to avoid the ghost terminal state at least once in the testing, this decreased his average score. There are a 2 possible of reasons for this:

- Due to the randomness relating to the epsilon, maybe Pac-Man just got unlucky. However I re-ran the experiment, and the average scores were the exact same. So, this cannot be the problem.
- All these data set have α = 0.1 in common, this decreases Pac-Man's learning rate could *theoretically* imply since Pac-Man doesn't adjust his q-values quickly enough in response to rewards and penalties. So, if near the final iterations of his training and he lands on a ghost terminal, he could be too slow to learn that is wrong and change his path.

Epsilon	Alpha	Fixed Gamma	Tiny	Small	Medium
0.1	0.1	0.9	419	500.333	496.083
0.1	0.2	0.9	503.417	500.333	496.083
0.2	0.1	0.9	419	500.333	496.083
0.2	0.2	0.9	503.417	500.333	496.083
0.3	0.1	0.9	503.417	500.333	409.8333
0.3	0.2	0.9	503.417	500.333	496.083

Table 2. The average score after experiment with reasonable epsilon and alpha values.

Conclusion:

In summary, the Q-learning algorithm with epsilon-greedy action selection enables Pac-Man to learn an optimal policy through adaptive, trial-and-error interactions with the environment. By carefully tuning the discount factor γ, learning rate α, and exploration rate ϵ, we can control Pac-Man's planning horizon, learning stability, and balance between exploration and exploitation respectively. Experiments demonstrated that a high γ (0.9) maximizes long-term rewards, while a moderate α (0.1 or 0.2) and ϵ (0.1 to 0.3) provide stable learning and effective strategy exploration. These settings allow Pac-Man to achieve near-optimal scores, navigating the grid efficiently while avoiding pitfalls.

The optimal parameters chosen for each map size are as follows:

- Tiny map: epsilon is set to 0.2, alpha to 0.2, and gamma to 0.9
- Small map, epsilon is 0.3, alpha is 0.2, and gamma is 0.9
- Medium map, epsilon is set to 0.2, alpha to 0.1, and gamma to 0.9.

Future changes:

In the future I would experiment with larger epsilon and alpha values to see if I can find any unique outliers and peculiarities.

Question 3:

I started with a simple perceptron model comprising a single layer with just one neuron to predict optimal actions for Pac-Man (found in `perceptronPacman_simple.py`). This initial model focused on classifying each action as either desirable or undesirable based on features extracted from the game state, using a sigmoid activation function. This basic setup provided a clear foundation, helping to establish how the features contributed to predicting optimal moves without introducing unnecessary complexity.

After evaluating the single-layer model, I experimented by adding a hidden layer to create a multi-layer perceptron (MLP). This change allowed the model to learn more complex patterns in the data, potentially improving accuracy by capturing relationships that a single neuron might miss.

Single Layer Perceptron:

Model Architecture

For the initial model, I used a **single-layer perceptron** with only one neuron. This setup provided a straightforward approach to classifying actions as either optimal or suboptimal based on the features derived from the Pac-Man game state.

Describe the algorithm:

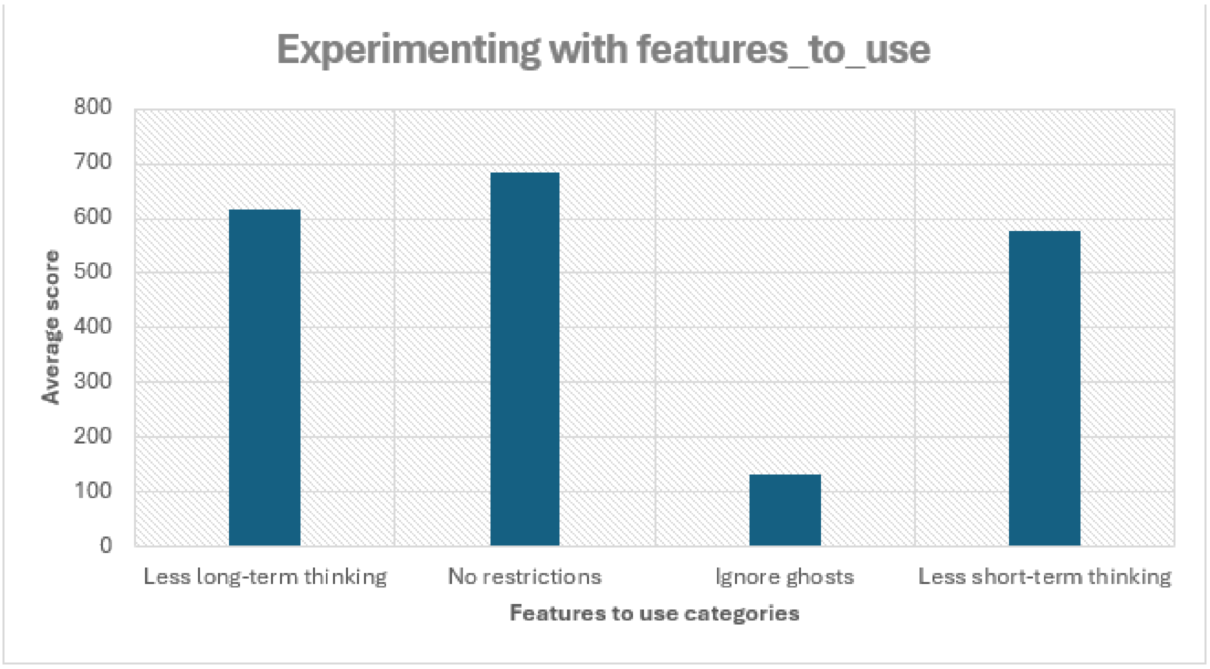
- `__init__`: initialize the weights to start at 0.
- `predict`: calculates the perceptron's output for a given feature vector. This is done by a dot product between the weights vector and the feature vector, then the answer is applied into the `activationOutput`.
- `activationHidden` and `activationOutput`: The sigmoid activation function is used for the reasons given above.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- `evaluate`: Measure the model's performance by calculating accuracy on a provided dataset. It iterates over the data set and see if the predication (use `predict`) matches the label within a reasonable threshold.
    - **Prediction Threshold**: A threshold of 0.5 is applied to classify predictions as 0 or 1.
    - **Accuracy Calculation**: Compares predicted labels to actual labels and calculates accuracy = Number of correct predictions / Total number of predictions.
- `train`: Trains the perceptron using a simple update rule based on the difference between predicted and actual labels (errors).
    - **Key formulas**:
        - **Error calculation**: `Error = y - y*` where y is the actual label and y* is the predicted probability (from `predict` method)

        - **Weight Update Rule**: `wᵢ` ← `wᵢ + learning rate * error * X_train[i]` where `X_train[i]` is the ith feature. This rule adjusts each weight based on the input feature and error, pushing the model towards correct predictions.
    - In the end of the algorithm, you can also `evaluate` your new weights on your validation data.
- `save_weights` & `load_weights`: writes and reads the weights to the `q3_weights.models` text file.

Experimenting with different features to use



	Less long-term thinking	No restrictions	Ignore ghosts	Less short-term thinking
Validation Accuracy	91.89	92.68	89.06	91.64

Figure 3. and table 3. Experimenting with which features should be used. For "Less long-term thinking" I ignored food withing five and nine spaces. For "No restrictions" there were no features blocked. For "Ignore ghosts" I blocked all features relating to ghosts. For "Less short-term thinking" I blocked all features that begin with "closest". With 100 iterations and 0.01 learning rate.

In table 3. Validation accuracy is calculated from the evaluate method.

I first experimented with feature selection, as the results will inform optimal inputs for models with a hidden layer by highlighting the most impactful features.

This experiment demonstrates the impact of different feature selection strategies on Pac-Man's performance. When using all features without restrictions, Pac-Man achieves the highest score, as shown in figure 3, indicating that a balanced approach, considering both short-term and long-term elements, yields effective decision-making. However, removing key features disrupts this balance.

For instance, ignoring ghost-related features leads to the lowest score, showing that these features are essential for survival as they help Pac-Man avoid dangerous situations.

Hence, I will assume enabling all features will give the greatest results in future experiments.

Mutli-Layer Perceptron:

Model Architecture

To add 1 hidden layer to my 1 neuron model I only had to make a few changes to the methods for the previous section.

- `__init__` method: Initializes number of hidden neurons, the weight matrices for connection between the input layer and the hidden layer (`weights_input_hidden`) and between the hidden layer and the output layer (`weights_hidden_output`).
    - These weights are initialized with `np.random.rand` to break the symmetry of all weights being 0 like with a single neuron.
- `predict` pseudocode:
    - `hidden_output = self.activationHidden(np.dot(feature_vector, self.weights_input_hidden))`
    - `# Compute final output`
    - `return self.activationOutput(np.dot(hidden_output, self.weights_hidden_output))`
- `activationHidden` & `activationOutput`: this method stays the same.
- `evaluate`: this method stays the same.
- `train`: Now you must update the weights of 2 matrices:
    - `self.weights_hidden_output += learning_rate * Error * hidden_output`
    - `δ_hidden=Error * self.weights_hidden_output * hidden_output * (1-hidden_output)`
    - `self.weights_input_hidden += learning_rate *` x^T `* δ_hidden`
        - x is the input feature vector.
        - `hidden_output = self.activationHidden(np.dot(X_train[i], self.weights_input_hidden))`
        - `Error = y-y*` where y is true label and y* is the predicted output.
- `save_weights` & load`_weights:` now there are two arrays to save and load, to distinguish between them in the text file they are separated by a line stating "array", so I take that into consideration when loading & saving the arrays.

Partial derivative: $\mathrm{d}L/\mathrm{d}w = x(\sigma(z)-y)$

Approach to train method:

The equations chosen for the train method are designed to iteratively adjust weights using gradient descent based on the error between predicted and actual labels.

The output layer weight update equation directly scales the error by the learning rate and hidden layer activation, guiding the model to reduce prediction errors.

The backpropagation equations, in the train method, for the hidden layer enable error propagation back through the network, allowing the model to learn complex patterns by adjusting weights at each layer to minimize the overall loss.

Optimizing learning rate and iteration parameters

Two new methods were created to run experiments more easily:

- `run_experiments`: Has the purpose of preparing and experimenting with different combinations of number of neurons, learning rates and number of iterations. Pseudocode:
    - Do nothing if `self.do_experimenting` (initialized in `__init__`) is False.
    - Otherwise: For each combination of number of neurons, learning rates and number of iterations initialize a new `PerceptronPacman` and then run `experiment_train` method.
- `experiment_train`: This is the same as train method but for each epoch you calculate training and validation losses and store in an array. These are calculated with the following formulas:
    - The loss function is L = (sum of Error for each sample)/N where:
        - N = number of samples
        - `Error = y -y*` where y is actual label and y* is predicted label.

Theory on parameters:

Learning Rate: A large learning rate causes Pac-Man to make abrupt strategy changes, potentially leading to erratic decisions, while a small learning rate enables gradual learning and fine-tuning, though it may slow down adaptation to new strategies.

Iterations: A large number of iterations allows Pac-Man to refine its strategy thoroughly but risks overfitting, whereas a small number may result in an incomplete or simplistic strategy due to insufficient training time.

Learning rate	Iterations	Accuracy
0.01	20	92.24
0.01	50	92.91
0.01	100	92.82
0.1	20	91.61
0.1	50	93.33
0.1	100	93.1
0.5	20	90.89
0.5	50	91.06
0.5	100	92.21
1	20	90.56
1	50	90.89
1	100	90.93

5	20	70.41
5	50	70.41
5	100	70.41

Table 4. With 20 neurons I experimented with different learning rates and iterations, and returning the accuracy of the validation data set.

It is expected that for smaller learning rates that a smaller number of iterations is needed to reach peak accuracy. This is because with a smaller learning rate, the model makes precise, gradual adjustments, allowing it to converge smoothly and reach peak performance with fewer iterations, increasing the iterations too much can invite newer information that could misguide Pac-Man.

A larger learning rate, however, can cause overshooting, requiring more iterations to stabilize it.

These are 2 different approaches to finding the optimal accuracy: lower learning rate and lower iterations; higher learning rate and higher iterations. This theoretical conclusion is supported by table 4 where for a learning rate of 0.01 or 0.1 the maximum accuracy is reach within 50 iterations but with a larger learning rate you need at least 100 iterations to begin to see positive results.

Hence, I will choose the approach which requires smaller iterations since its faster, it is the one with a smaller learning rate. From table 4 we can get the optimal parameters, these are learning rate = 0.1 and iterations = 50.

With this you get an average score of 692.07, which is higher than the score from single layer perceptron.

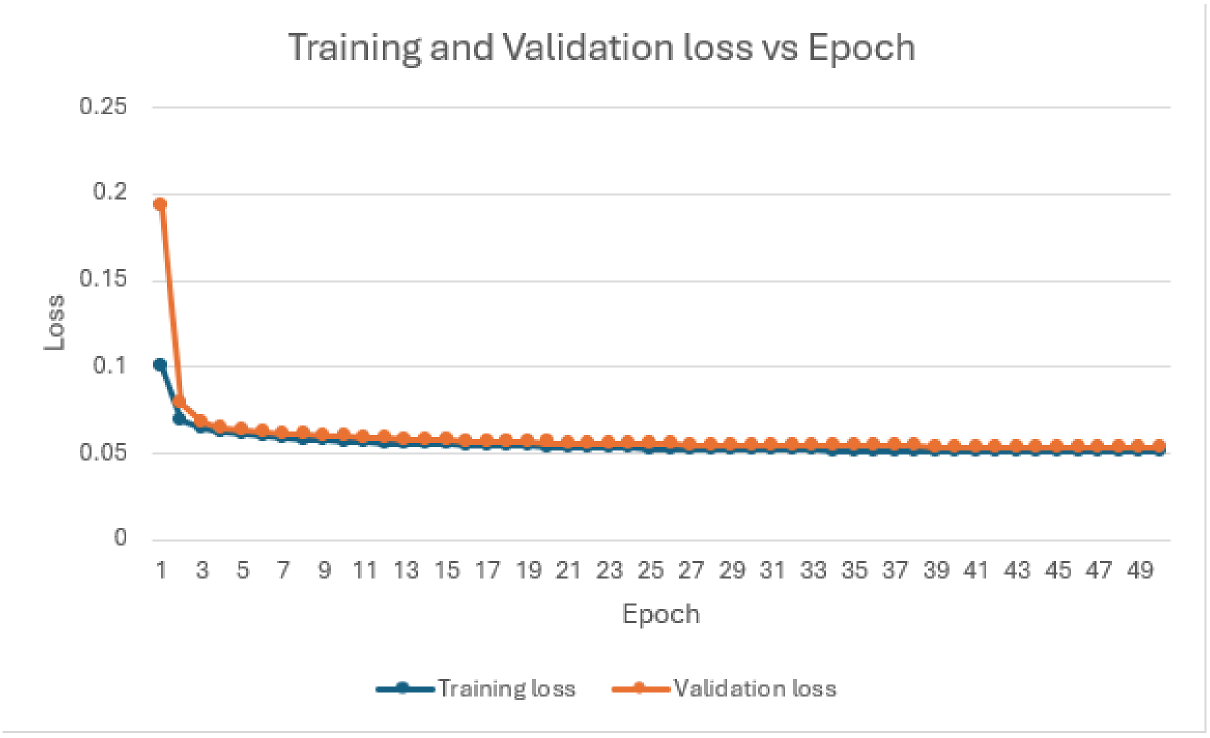


Figure 4. learning rate = 0.1 and iterations = 50 and 20 neurons in 1 hidden layer. This calculates the training and validation loss.

In theory, a well-trained model should show a rapid decrease in both training and validation losses during initial epochs, followed by stabilization as the model converges. A small gap between training and validation losses indicates good generalization, meaning the model is not overfitting and is likely to perform well on unseen data.

This is evident from Figure 4 because both training and validation losses decrease sharply in the first few epochs and then level off around the same low value. The close alignment between the two losses throughout training suggests that the model is learning effectively without overfitting, achieving stable and low errors on both datasets.

Optimizing number of neurons in the hidden layer.

Neurons	Accuracy
1	70.43
5	92.26
10	92.26
15	92.15
20	92.15
25	91.8

Table 5. learning rate = 0.1 and iterations = 50. Experiment with different kinds of neurons in 1 hidden layer.

Experimenting with the number of neurons in a single hidden layer allows us to adjust the model’s complexity to find the right balance between learning capacity and generalization.

Increasing the number of neurons gives the model more power to learn complex patterns, which can improve performance on intricate tasks by capturing detailed relationships in the data. However, too many neurons can lead to overfitting, where the model memorizes the training data instead of generalizing to new data.

Conversely, decreasing the number of neurons simplifies the model, which can improve generalization and reduce overfitting risk but may limit the model's ability to capture complex patterns if the task requires it. Thus, tuning the number of neurons is essential for achieving a model that learns effectively without sacrificing robustness.

This theoretical conclusion is reflected in Table 5 because the accuracy drops of quickly in too many neurons (25 neurons) and too little neuron (1 neuron). Therefore, I chose 20 neurons.

Conclusion:

With 20 neurons, a learning rate of 0.1, and 50 iterations, the model achieves high accuracy on the training data and the validation and training losses converges closely. This suggests that the model after being trained is flexible to newer scenarios. Overall, this configuration provides strong performance because the parameters are chosen with theoretical backing and proven by an abundance of figures and tables.

Future changes:

In the future I would also experiment with different activation functions such as Relu, and the `activationHidden` and `activationOutput` having different functions.