Final Report
28th November 2019

The Pacman Game

Using Deep Reinforcement Learning

Vishal S Rao - 01FB16ECS450
Vishnu V Singh - 01FB16ECS451
Aditya Lokesh - 01FB16ECS461

Under the guidance of
PROF. SHRIKANTH H R
August 2019 – December 2019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

100ft Ring Road, Bengaluru – 560 085, Karnataka, India

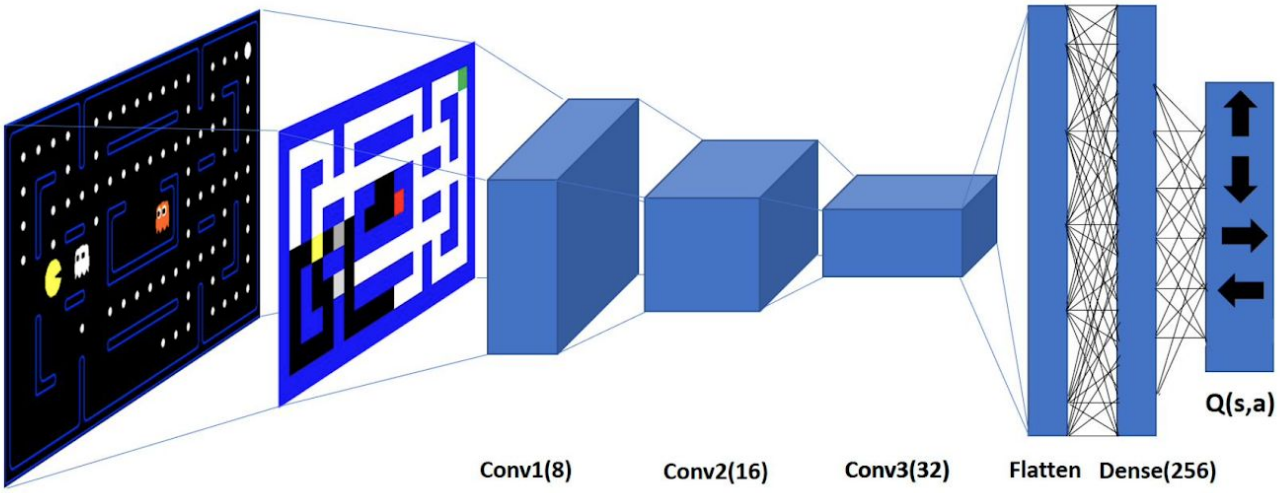
Abstract

The Pacman game is one of the oldest games we have played. We have tried to implement and optimize a version of this game where Pacman, learn how to win using Deep RL. We referred to a paper from Stanford[1] that has done this and a course from UC Berkeley[2] which provides the GUI for this purpose.

The GUI has different-sized mazes for the game. For the purpose of this project, we have used two - “SmallGrid” and “SmallClassic”.

Architecture

The Network Architecture is as follows -



We use TensorFlow to implement a Convolutional Neural Network (Q-network), to generate the Q-values corresponding to the four possible actions at each state: Up, Down, Left, Right. DQN architecture is shown in the architecture figure, and it is composed of three convolutional layers (3x3x8, 3x3x16 and 4x4x32), a flattening layer and a fully connected layer with 256 neurons. All layers except the output layer use a Leaky Relu activation function. The output layer does not contain any activation function as we require the output to give Q-scores (This is not a classification problem).

We use a CNN before the fully-connected network so that we do not have to manually provide the features, rather the CNN’s automatic feature detection does this job for us.

The Grid has feeds 6 input Channels to the CNN, i.e,

1. The Wall Matrix
2. Pacman Matrix
3. Ghosts Matrix

4. Scared Ghosts Matrix
5. Food Matrix
6. Capsules Matrix

Hence the input_shape to the CNN is (None, H, W, 6), where H is the number of pixel rows, and W is the number of pixel columns of the input image (Pacman Grid).

Layer Type	Output Shape	# of Parameters	Connected to
conv2D_1	(None, (H-2), (W-2), 8)	80	conv2D_2
conv2D_2	(None, (H-4), (W-4), 16)	160	conv2D_3
conv2D_3	(None, (H-7), (W-7), 32)	544	Flatten_1
Flatten_1	(None, (H-7)*(W-7)*32)	0	Dense_1
Dense_1	(None, 256)	(H-7)*(W-7)*32*256	Output
Output	(None, 4)	1024	OHE

Deep Q-Learning

The network architecture given above implements the Q network. The Q network takes as input a state of the Pacman game and outputs a vector containing the Q-scores for the possible actions (Up, Down, Left, Right) on the given state.

We choose deep reinforcement learning over vanilla reinforcement learning because our state-action matrix is too large to be stored as a table, hence, we use a neural network to infer the Q-scores of a state-action pair.

The cost function used for calculating error,

$$Error = (Q(s,a) - r(s,a) - \gamma * max\, Q(s',a'))^2$$

We can see that the error function used in MSE because this is not a classification problem.

Steps for Training (Epsilon Greedy Policy) -

1. Initially, initialize epsilon to 1.
2. The network is given the next state (s') as input with the q-scores initialized to 0. The output will contain the q-scores for all possibles actions (a').
3. A random number from 0 to 1 is chosen. If it is greater than epsilon, then we choose the highest q-score to be Q(s', a'), else we choose a random q-score to be Q(s', a').
4. This Q(s', a') is used to calculate the error of the current step.

5. The error is propagated backward and the weights are updated using Adam optimizer.
6. Update epsilon using - $\varepsilon = max(EpsilonEnd,\ 1.0\ -\ epochNumber/EpisilonStep)$

Steps for Testing -

1. At each state (s), pass the state to the deep network.
2. Choose the action (a) with the highest Q-score.
3. Move to state (s') by applying the action (a).

Hyperparameter Tuning

The hyperparameters used -

1. Learning Rate, $\alpha = 0.0002$
2. Batch Size = 25
3. Discount Factor, $\gamma = 0.95$
4. Replay Memory Size = 100000
5. Epsilon_Start = 1
6. Epsilon_End = 0.2
7. Epsilon_Step = 1000

Optimizations

The optimizations we have implemented as an attempt to improve the DQN are-

1. Leaky ReLU activation
    - We have used the Leaky ReLU instead of the normal ReLU to avoid the Vanishing Gradient Problem.

2. Batch Normalization
    - We have used Batch Normalization at each convolutional layer (between the output of the conv layer and the activation function) to solve the covariant shift problem.

3. Epsilon Greedy Policy
    - Gives a good mix of exploration vs exploitation

4. Replay Memory
    - Replay memory is normally used in DQNs. It is a method to store the 'N' most recent experiences and while training, we randomly sample an instance from this memory. We use replay memory while training to break the correlation between consecutive samples.

5. Prioritized Experience Replay
    - PER is used to learn more efficiently. Instead of sampling experiences randomly during training from the replay memory, we sample them based on the probability with which they occur across the N most recent episodes. This induces a human-like learning strategy into Pacman.

Hardware Specifications

We used a cloud service provider called PaperSpace[6]. The specifications of the server used-

- Machine Type - P4000
- RAM - 30GB
- vCPUs - 8
- GPU - 8GB
- HDD - 50GB
- Ubuntu 16.04

Results

Training Results

1. SmallClassic Layout

Training Episodes	Accuracy	Time (in seconds)
5000	21%	16646.19

2. SmallGrid Layout (With Optimizations)

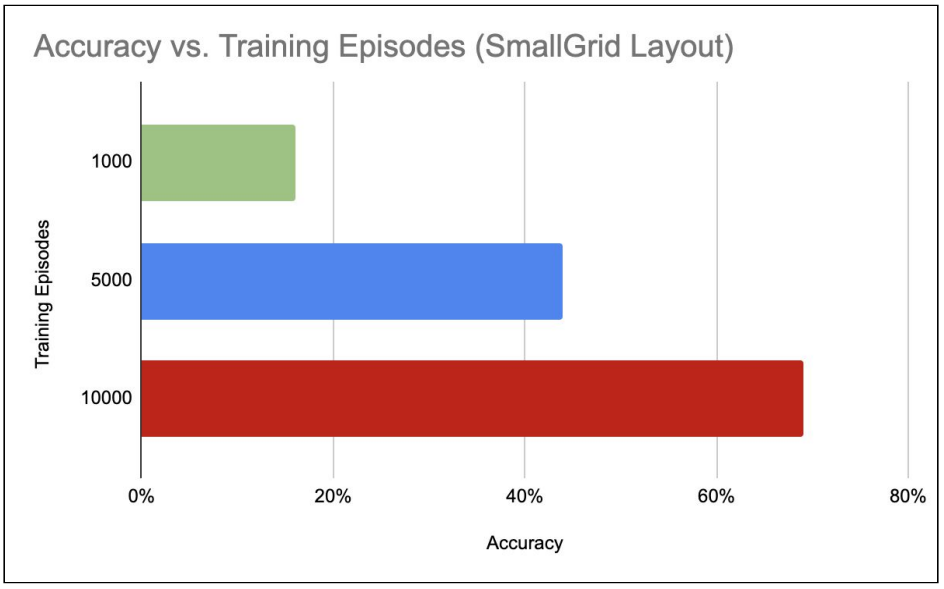
Training Episodes	Accuracy	Time (in seconds)
1000	16%	96.8
5000	44%	1441.03
10000	69%	2899.62

3. SmallGrid Layout (Without Optimizations - Leaky ReLU and Batch Normalization)

Training Episodes	Accuracy	Time (in seconds)
5000	34%	1953.54

4. MinimaxClassic Layout (Prioritized Experience Replay Analysis)

Optimizations	Training Episodes	Accuracy	Time (in seconds)
Without PER	4000	48%	595.54
With PER	4000	61%	19231.65



Conclusion

- We got an accuracy of 69% for the smallGrid Layout Pacman Game on training for 10000 episodes.
- We observed that not applying the optimizations reduces the accuracy by 10% and increases the training time significantly.
- We got an accuracy of 21% for the smallClassic layout on training for 5000 episodes. This would improve significantly if we train over a higher number of training episodes.
- Prioritized Experience Replay increased training time and increased accuracy significantly.

References

[1] http://cs229.stanford.edu/proj2017/final-reports/5241109.pdf
[2] http://ai.berkeley.edu/reinforcement.html
[3] https://adventuresinmachinelearning.com/reinforcement-learning-tensorflow/
[4] https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad
[5] https://github.com/mrkulk/deepQN_tensorflow
[6] https://www.paperspace.com/