CS230

Pacman – using Deep Networks for State evaluation

Manuel Anton Kohnen

manuelak@stanford.edu

Abstract

This project uses the Berkeley Pacman program with an Expectimax algorithm. With this context, I will try to get the best game state value estimation for the Expectimax, and this by comparing different Deep Network sizes and types. I will also touch on the temporal credit assignation issues and its impact on the learning process.

1. Introduction

The game of Pac-Man **Error! Reference source not found.** is a classic 80's arcade game illustrated in figure 1.

This project uses the Berkeley Pacman application [2] fixing on the Expectimax Reinforcement Learning algorithm. The aim of the project is to obtain the best possible evaluation function of a given state, and within the parameters set in the Stanford CS221 assignment [3]

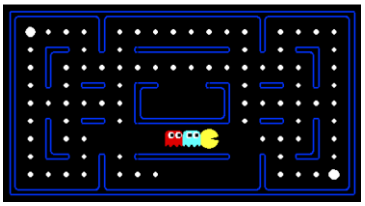

Fig 1 – Pacman game

Using the Pacman map represented as a string, plus other information on the state, I will try to find the best Deep Network architecture to steer the behavior of Pacman which would maximize Pacman's score at a game.

I discuss an attempt at Convolutional network, and which size of Fully Connected networks work best.

Finally, I will address the topic of temporal credit assignation, that is how do you weight the past vs the future in order to determine the best policy. As I found out, this was the most impactful of all parameters.

2. Framing of the problem

As mentioned during the introduction, this project is using a Stanford CS221 assignment as the basis for the work. In this problem, the objective is to obtain the best evaluation function for a given state. It is not comparing different Reinforcement learning algorithms but comparing the best deep neural network to achieve the evaluation of any given state of the game. The RL algorithm is fixed to an Expectimax search algorithm.

3. Related work

Work by Jake Grigsby [4] gives advanced techniques for Deep Q Networks and is very friendly to read. Note that this work differs in the sense that it starts with a screen of the game as input, so there is more emphasis on how to get this screen into logical data using CNNs. I am skipping this step and getting the logical data from the Pacman application right away. Jake Grigsby work also discusses different learning algorithms, which is a fixed parameter in my project. In then explores dueling architectures which are relevant for Minimax algorithms, while for this project we fixed on Expectimax, i.e. the ghosts are behaving randomly, so it is not applicable.

Work by Abeynaya et al. [5], is very close to this project in the sense that they use the same Berkeley Pacman framework. The paper however differs in the sense that it seems to analyze the results of a given fixed network (figure 3 of the paper) with different Pacman layouts. Whereas I am fixing the layout and optimizing the network for it. The scores obtained by Abeynaya and al. for the same layout seem to converge to a 200 points mark. My approach was to challenge the network architecture and the resulting network, which ends up much simpler, reaches the 1200 points on average. One main reason for discrepancy could be the score used against a given state for the learning. As we will see, this is the most impactful parameter of all, but I cannot compare as it is not mentioned in Abeynaya et al.

Generally, most papers look for what algorithm will perform the best to solve this problem, whereas I took a fixed algorithm, a fixed framework, and tried to find the best network for addressing this narrower problem. Note that this was done intentionally as I wanted to focus on deep learning rather than reinforcement learning.

4. Dataset and features

Features

The information on a state is limited to what the editable function of the assignment (betterEvaluationFunction()) can see in its scope.

The main piece of information available is a map of the maze with all relevant information like existence of a food pellet, position of Pacman, position of ghosts, etc… This map is a discrete map as follows (fig 2.) where '%' represents walls, '.' represents food pellets, 'G' are the ghosts, 'O' represents a power food, and '<' represents Pacman.

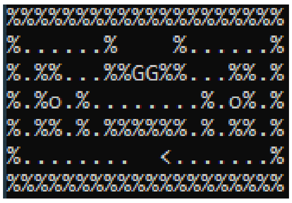

Fig 2 – Pacman map as a string

Along with this map, the information available which I store is:

- Scared timer (In case Ghosts are in scared mode (can be eaten), how many seconds left)
- Direction (last action taken (Up, Down, Right, Left)
- Turn number (this is stored additionally but not available in the function)

It can be noted that the different combinations of possible states are a combination of agent positions and food mapping. There are 53 food pellets in this game configuration. Each food position could have food or not, that is 2^{53} combinations. The agents (1 Pac-Man and 2 ghosts) can be in any of the 60 locations, giving 60^3 combinations. Multiplying this with all the possible scared timer values and directions, we are in the order of 10^{25} possible combinations.

Labels

The score at each state is stored. The reward system is as follows:

+10 for each food pellet eaten

+200 for eating a ghost

-1 for each turn

+500 for winning the game (eat all the food pellets)

-500 for losing the game (Pac-Man gets eaten by ghost)

Meta data on the overall game is also stored such as final score, win/lose flag.

Volumes

I created a dedicated MySQL database to store the data. At the end of the project, and which the results are based on, the data amounted to over 10,000 games that generated over 4,000,000 states.

Note however that, as suggested by my TA, I am not using some of the oldest data as the algorithms were improving. I was regularly pruning some of the earlier data (manually at regular interval)

5. Method

Algorithm

As mentioned earlier, the RL algorithm is fixed to an Expectimax search algorithm of depth 3. Ghosts are behaving randomly (hence the expected value), and the algorithm maximizes for Pacman. State values are calculated as follows, where *s* is the state, *d* is the depth, *a* is the action. Pacman is agent 0, i.e. a_0, is Pacman's action, other subscripts are ghosts.

$$V_{exptmax}(s,d) = \begin{cases} Utility(s) & if\ IsEnd(s) \\ Eval(s) & if\ d = 0 \\ max_{a \in Actions} V_{exptmax}(Succ(s,a),d) & if\ Player(s) = a_0 \\ \frac{1}{|Actions|} \sum_{a \in Actions} V_{exptmax}(Succ(s,a),d) & if\ Player\ in\ (a_1, \dots, a_{n-1}) \\ \frac{1}{|Actions|} \sum_{a \in Actions} V_{exptmax}(Succ(s,a),d-1) & if\ Player = a_n \end{cases}$$

Best Heuristic

The goal of the project is to obtain the best Eval(s) function as shown above. At any point in time, the current score is known. What is yet to determine is the heuristic to be added to the current score in order to determine what is the most desirable state, i.e. the state maximizing the end value.

$$Eval(s) = currentScore(s) + heuristic(s)$$

Throughout this document I will refer to the $currentScore(s)$ component also as the "baseline".

Architecture

1) Using the python str() function of the Pacman application which converts the map into a string as in figure 2, I save the states and the remaining features at each state.
2) I convert this string plus the additional features into a vector of floats. I strip the map of the walls, which will never change, and keep only the map features with information. Once adding the scared timer and direction, I end up with a vector of length 73.
3) I feed this 73 length vector to a 3 hidden-layer Fully Connected network, using ReLU as the activation function.
4) The last layer is a standard linear layer meant to return the Y value corresponding to this state.

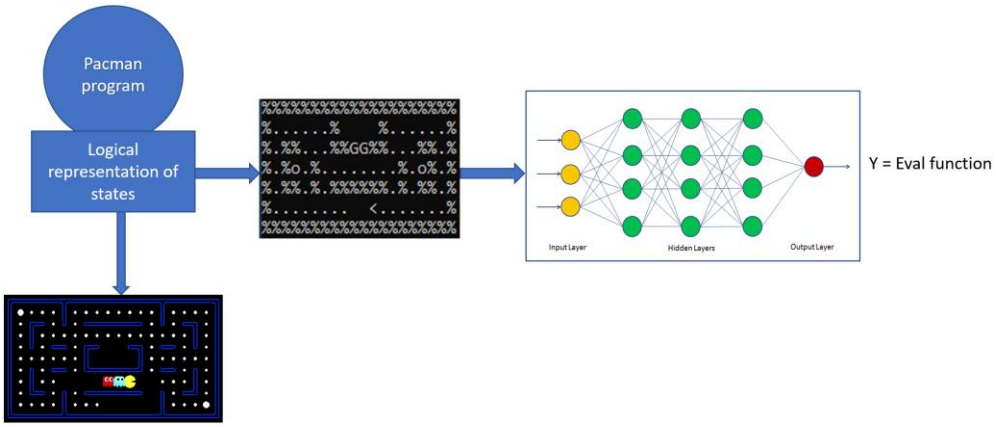

Fig 3 – Overall architecture

The loss function is a standard sum of squares. i.e. $\frac{1}{m}(Yhat - Y)^2$

Why Fully Connected and not Convolutional

As the features contains a game map but also logical features such as "scared timer", "last direction", I would have had to combine a CNN with a FC network, by concatenating the logical features with the flattened CNN output, as illustrated in figure 4. This is best explained in Adrian Rosebock's article [6].

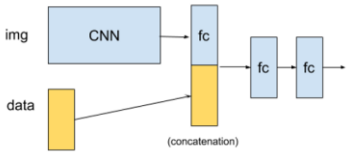

Fig 4 – Overall architecture

During an attempt to build the CNN part, the CNN layer was clearly degrading performance, compared to Fully Connected, to the point of getting close to baseline + randomness. I suspect that the map of characters (18x5 once excluding the edges) is too small for the CNN to add value. Since the whole relevant data can be turned into a 72 long vector already without any loss of information, the CNN did not help. Of course, if the input was a screen image of a Pacman game, with let's say 500x300 pixels, then a CNN would have probably been necessary. Here, it hindered performance and I concentrated on Fully Connected layers.

Experience buffer and replay

In order to gather data and train the network, training at each game would have been to lengthy. I have therefore used experience buffer technique. On other words, I would run a certain number of games (usually around 20), save the data, then retrain the network adding those 20 games and repeat.

Comparison of different network variants

In order to compare the different networks, I randomly choose a seed, then reset the random.seed() to this chosen seed every time I run a batch of games with a different network. This allowed me to compare networks with the same seed.

Exploration

For Reinforcement Learning, an exploration strategy is required in order to avoid getting stuck with a subpar policy. Often it is implemented in the form of taking a random action at an Epsilon probability. In this project, first it is to be noted that Ghosts are random. This in itself will ensure diversity in scenario. Secondly, I was running different networks in parallel to compare them and storing all resulting game states in a common repository, which was then exploited in the learning process. I believe that this comfortably replaced an exploration strategy and did not implement an explicit one.

6. Experiment

Mini-batch size

I have used a Windows Desktop PC which I bought for the occasion. This Desktop has a "Nvidia GeForce GTX 1660 Super" GPU with 6 GB memory. I initially started without mini-batches until I hit a GPU memory overflow at around 1.5M states to process. I have then implemented mini-batch to contain around 1M states in each batch (i.e. 4 mini-batches currently)

Learning rate

I have used 0.001 arbitrarily from the start. This value proved robust. The cost was diminishing rapidly before flattening after around 1000 epochs, and never did I notice overshooting (i.e. Cost never increases from one epoch to another). So, I stayed with 0.001

Number of nodes per layer

Early on, a three hidden-layers architecture seemed to score best (see next section). Fixing on a three hidden-layers architecture, I have compared different number of nodes per layers as follows:

- FC1 : 50, 25, 10
- FC2 : 80, 50, 10
- FC3 : 100, 100, 10
- FC4 : 100, 150, 10
- FC5 : 100, 150, 20

(as a reminder, the input layer is 72 and the output layer is 1)

The average scores after 100 games is shown in figure 5.

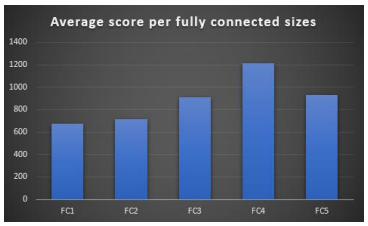

Fig 5 – Score per network size

As one can see, there seem to be a diminishing return after FC4, especially the fact of having 20 nodes at the last hidden layer (instead of 10), made this network score more poorly. So the champion for these different combinations seem to be FC4. Adding more nodes showed diminishing return at some point and even hindered the performance when increasing the last layer.

Number of layers

Early on, a three hidden-layers architecture scored the best. Using the champion of the 3-layers architecture below, I have compared adding or removing a layer to measure the relative performance. The scores in figure 6 confirmed the early findings that 3 layers seemed to perform best.

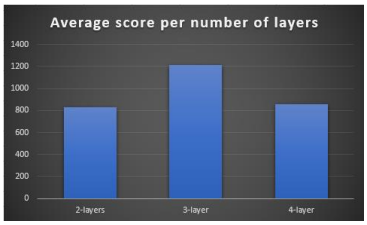

Fig 6 – Score per network number of hidden layers

Overfitting

Overfitting did not seem to be a concern. As I was experimenting, to get quick results on a new combination of parameters, I would do an initial training with a limited number of epochs, i.e. having an equivalent of early stopping. Then run some games to feed in the experience buffer and run training again. Sometimes this process would run overnight. Comparing the early stopped results with the increasing number of epochs and reducing loss, I have found no pattern of diminishing performance hence no overfitting. Another reason preventing overfitting I believe is the turn number described in the next section.

Turn number adding noise

As described in the reward policy of the game, each turn subtracts one point from the score. The framework given for the assignment and which I tried to stick to does not provide for the turn number in the evaluation function. Unless changing the call of the function which was not allowed. So although I collected it with the data, it was not used in the learning process as not available later when making predictions. Because of the negative points associated to turn numbers, we could easily imagine that if Pacman did back and forth, a given state (position of food pellets, ghosts, etc…) could be exactly the same twice, but with a different score. Just because of the number of turns. This means that a good portion of the score is not explained in the model, hence adding substantial noise.

7. Best heuristics and credit assignation issue

Far more impactful that the input was the correct labelling/scoring of states.

In order to estimate the value of a state, we have on one side the facts, i.e. the actual observation of the past, represented in the $currentScore(s)$. What we need now is an estimation of what additional future gains this state is likely to provide.

For any state, the additional gain, that is the *heuristic* term in the equation $Eval(s) = currentScore(s) + heuristic(s)$, is measured by:

$$heuristic(Stored\_Sate) = finalScore(Stored\_Game) - currentScore(Stored\_State)$$

The above in words means that the remaining gain expected for a state is measured by the final score achieved in that game minus the current score of that state.

The above gave very poor results. Pacman died at almost every game when training with the above labels. I believe that the future signal which is very noisy due for example to the issue of the turn number, was overwhelming the short-term signal of how well the state was doing so far. Especially early on in the game, it was too early to steer Pacman on such long-term goals. With the above, we are basically weighting the noisy prediction as much as the hard facts. So the above heuristic may work asymptotically, but with the current data volumes, noise, etc… It did not work at all.

To be clear, the above heuristic performed worse than the baseline, *currentScore()* alone. I had to recourse to shorter views. I have tried different methods and one that worked the best is as follows.

I labeled the states according to the additional score performed on the next three turns. I have used weights for each period. So, the label/scoring used, i.e. the reward at time *t* as:

$$label\ (s_t) = \beta_1\big(score(s_{t+1}) - score(s_t)\big) + \beta_2\big(score(s_{t+2}) - score(s_{t+1})\big) + \beta_3(score(s_{t+3}) - score(s_{t+2}))$$

After trial and error, good values for $\beta_1, \beta_2, \beta_3$, seemed to be 80%, 40%, 20%.

In short, what seemed to work the best was to make the model myopic to three turns ahead only, and decreasing the weight of information as we guessed further into the future. I did not have time to make the parameters β learnable but it would certainly impact the performance greatly to fine tune these. Could be future work.

Note that another option which I did not have time to test would have been to use the very first heuristic presented, the one using the final score minus current score, and give a discount on the heuristic. Ideally, one could modify the discount on the heuristic as the game progresses. I.e. a high discount early in the game, making the future prediction less relevant, and reducing the discount as the game progresses. I had no time to test this.

8. Comparison with handcrafted features

The initial Stanford CS221 suggested to handcraft features to obtain the best heuristic. During this earlier assignment [3], I handcrafted features such as:

- Distance to nearest food pellet (taking into account the walls)
- Going backwards compared to last move (0 or 1)
- Distance to a scared ghost
- Distance to a bad ghost

For this CS221 assignment I scored an average of 1100, compared to 1215 for the best Fully Connected model in this project. So, this model performs already better than with the handcrafted features defined above.

The objective of my project was to compare hand crafted features with a neural network which is given only the raw data. It would be interesting to see if adding the hand-crafted features above into the input of the network would improve performance. I.e. if we spell out a few important features, even if redundant to what is already there, would the network perform better? I leave this to further work.

Note that some students scored at 1700 on the leaderboard during the course. So Human level performance with handcrafted features seem to be at around 1700

9. Conclusion and further work

We have seen that there seem to be an optimum size of network for a given task. Both in terms of layer as well as number of nodes. We could observe that adding layers or nodes sometimes would be detrimental to the performance. And not only through overfitting it seems.

We have seen that the key issue is how to correctly balance future observation and current score at a given state. The Expectimax algorithm is exploring at depth 3, and choosing which leaf scored best. We can improve the actual score by a heuristic, but at some point, the uncertainty of the heuristic overwhelms the signal and becomes counterproductive. This is actually more impactful than network architecture. It would be interesting to get to the bottom of this and perhaps learning through parameters this careful balance between observed facts and promises when evaluating a state. i.e. the beta parameters.

We have seen that for this amount of data, a Convolutional Network did not bring benefits. Most likely due to the small size of the input features.

We were able to outperform basic handcrafted features as I defined them, although the score of the handcrafted features was not the optimum achievable with this method. It would be interesting to push further to reach the 1700 mark reached by some students and hence the seemingly achievable human performance.

It would also be worth exploring the mix of handcrafted features and raw data to see if human guidance to highlight what is clearly relevant, even if redundant to the raw data, enhances performance or is quickly matched by the network anyway.

10. Contribution

I was alone in this project. Not that I necessarily like to work alone. I have a demanding 100% job as Managing director at a leading consulting firm and my job leaves me little time to work on the course. And almost exclusively on weekends. I did not want to be liability for a group with different paces.

I would like to acknowledge the contribution of Jo Chuang my TA for the guidance.

References

[1] Game of Pacman on Wikipedia: https://en.wikipedia.org/wiki/Pac-Man

[2] Berkeley Pacman Code: http : ==ai:berkeley:edu=projectoverview:html

[3] Link to Stanford CS221 assignment. Class of spring 2020: https://stanford-cs221.github.io/spring2020/assignments/pacman/index.html

[4] Jake Grigsby , Advanced DQNs: Playing Pac-man with Deep Reinforcement Learning : https://towardsdatascience.com/advanced-dqns-playing-pac-man-with-deep-reinforcement-learning-3ffbd99e0814

[5] Abeynaya Gnanasekaran, Jordi Feliu Faba, Jing An, Reinforcement Learning in Pacman: http://cs229.stanford.edu/proj2017/final-reports/5241109.pdf

[6] Adrian Rosebrock, Keras, multiple inputs and mixed data : https://www.pyimagesearch.com/2019/02/04/keras-multiple-inputs-and-mixed-data/