Machine Learning Engineer Nanodegree

Capstone Proposal: Playing PacMan using Deep Reinforcement Learning

Prasoon Shukla
December 29, 2017

Domain Background

Reinforcement learning is a subfield of Machine Learning in which a machine agent learns the optimal behaviour in an environment in a trial-and-error manner. The agent is put in an environment and allowed to interact with it. Any action the agent performs results in a reward which can be either positive and negative and our intent is to train the agent to perform actions that result in the highest long term cumulative reward. As such, the agent learns on its own - there is no training data as is the case with supervised learning.

The problem of reinforcement learning can be framed mathematically as a Markov Decision Process (MDP) [1] where the environment - as perceived by the agent - is represented by the state space ***S*.** The agent, ***P***, can perform any action from the set of all actions, ***A***, to gain some reward, ***r***. Then, the objective is for the agent to learn the optimal action to perform in any given state that maximizes the total long term reward to the agent. One of the ways to achieve this is by using Q-Learning [2]. In Q-Learning, we calculate the Q function, *Q(s, a),* for each state-action pair in an iterative manner and figure out the optimal action using this Q-value.

In recent years, there has been an increased focus on 'general purpose AIs' - AIs that can learn much like a human would. Reinforcement learning is a good tool for building such a general purpose AI but unfortunately, the state space for any non-trivial task is prohibitively large. If we were to use the Q-learning iterative algorithm to teach our agent, it would take us, in some cases, thousands of years to converge to the optimal solution. This has led to the (very recent) development of the Deep Q Learning algorithm which uses deep neural networks to approximate the Q function[3][4]. The neural net helps to extract meaningful features from the data thereby drastically reducing the effective state space and also learns the Q-values by incorporating the reward into the loss function [5]. We will therefore use Deep Q-Learning to teach an agent to play PacMan.

Problem Statement

Given to us is the Ms. Pac-Man game. The game consists of a maze with 'pellets' scattered around and 'ghosts' which chase the main player (PacMan). The objective of the game is to acquire all pellets without running into the ghosts. Some special pills, called 'energizers', allow the main player to attack and kill 'ghosts' for a small duration. Our objective is to make a player bot that can successfully navigate this maze, collecting all pellets without getting killed more than three times thereby completing a level. We will try to complete at least one level.

Datasets and Inputs

We will be using the OpenAI Gym 'MsPacman-v0' environment [6] to collect the data from the simulation. OpenAI Gym is an easy way to simulate a game environment and to retrieve the state and reward. The environment is provided as thr RGB image of each frame of size (210, 160, 3). A typical frame from the game contains maze walls, pellets, ghosts, number of lives remaining and the score. The agent can go in one of the 8 possible directions - left, right, up, down, left-up, left-down, right-up, right-down - or do nothing. The rewards are calculated based on the result of the agent's actions - eating a pellet, killing a ghost all result in positive rewards. Getting killed by the ghost results in a negative reward. Based on these, the weights of the neural net will adjust so that the NN can compute a good approximation of the optimal Q function [4].

Since we have the image of the frame, we have all the information needed by the neural net to construct the state. Just as in any other RL problem (like smartcab), the data isn't pregenerated - we simulate the data at training time. It should be noted here that while the dimensionality, and therefore the state space, appears quite large here, it's not actually as large as it looks. Much like in the MNIST data set, we can project the 210 * 160 * 3 = 100800 dimensional data to a much smaller subspace without losing any information. The reason for this is that a lot of data in the image is immovable and unchanging.
Once we have a frame, we preprocess the it (convert it to grayscale and scale down the size) to reduce the dimensionality without losing much information. This preprocessed image will then be passed to the neural net to decide the optimal action. Once the agent decides on an action and passes it to the environment, the action is executed on the next 2, 3 or 4 frames randomly. Once the execution is completed, the agent is provided with a reward. We use the reward and other Q-values to calculate loss and train the neural net.

Solution Statement

The problem is a regression problem (since Q-values are real numbers) and is solved by the use of Deep Q-Networks (DQN). In the paper by the DeepBrain team[3], the DQN is a convolutional network having following architecture:

Input -> Conv1 (8x8x32 filter) -> ReLU -> Conv2 (4x4x64 filter) -> ReLU -> Conv3 (3x3x64 filter) -> ReLU -> FC4 (512 neurons) -> ReLU -> FC5 (18 neurons) -> Output Q-values

The reason for using a CNN here is to preserve space locality. Since we are essentially 'reading' from an image it makes sense to use a convolutional network. Notice that we are not using Pooling layers . This is because we do **not** want space invariance - we want the action to be **dependent** of the spatial properties. Also note that last last layer is made up of 18 neurons which correspond to 18 different Q-values for each of the 18 possible actions in the original paper. Since our set of actions is smaller, we will modify this network architecture slightly to have smaller output of 9 units, one for each action.
Given a transition *<s, a, s', r>,* network will learn through backpropagation using the following loss function [5]:

$$L = \frac{1}{2}[r + \gamma(max_{a'}Q(s', a')) - Q(s, a)]^2$$

Based on this loss, we adjust the network weights using backpropagation until the loss tapers off - this would be the point where we would have converged to a good approximation of the Q-function. We will measure the effectiveness of the bot using the total reward accumulated by the agent until the game ends (more on this in the metrics section). Training the net with the same architecture on the game should result in similar performance numbers.

Benchmark Model

On the OpenAI Gym, several solutions to this problem have been submitted. We will use the best submission as the benchmark to compare against our solution[7]. The current best solution has a 100 episode mean of 6323 reward points that was achieved in 38 minutes of total playtime.

This benchmark model uses asynchronous advantage actor-critic based LSTM (A3C LSTM) network to achieve this high score. This new network model was released by Google DeepMind in 2016 [8] and is a giant improvement over the Deep Q-Network based models. This new network - while being applied to the same kinds of tasks - is completely different in its functioning than DQN, provides much better results and is simpler and faster than DQN. We don't expect our results to be very good compared to A3C LSTM but we should still see good results compared to human performance.

Evaluation Metrics

The underlying game doesn't have a specific threshold where it is considered solved - the aim is simply to score as high as possible. In this vein, we accumulate the rewards that the player receives until all lives are exhausted. The cumulative score at the end of the game is considered the "total score" for the player and it is this number that is used as the evaluation metric for measuring how well the model performed. This same metric is also used for the benchmark model for comparison. The simulation is run multiple time and the top 100 scores are taken to evaluate the mean score.

Project Design

As discussed earlier, we are going to use a Deep Q-Network to solve this problem. We follow these steps:

- Set up the development environment.
  - Installing OpenAI Gym.
  - Ensuring that the MsPacMan-v0 gym environment runs without issues and can provide state data, action list etc. after each frame.
  - Installing Anaconda, Tensorflow and Baselines
- Create a convolutional neural network using baselines (which is a wrapper over tensorflow provided by OpenAI). As discussed earlier, we will use the original model (shown below) used by the DeepMind team [3] as a base and make modifications to it depending on the requirement of solving PacMan and on the limitations of the machine we're training the model on. We may have to reduce the number of parameters for the data to fit in the GPU's memory.
  - *Input -> Conv1 (8x8x32 filter) -> ReLU -> Conv2 (4x4x64 filter) -> ReLU -> Conv3 (3x3x64 filter) -> ReLU -> FC4 (512 neurons) -> ReLU -> FC5 (18 neurons) -> Output Q-values*
- Set up the replay memory. This is a piece of memory that stores past experiences (transitions). It is necessary to use replay when training DQNs so that the estimated Q-values are somewhat stable. Otherwise the Q-values fluctuate wildly and the network never converges. We keep a limit on the replay memory size - it's a queue with a fixed memory where newer transitions replace older transitions.
- Set up the hyperparameters initially - we have:
  - α : The learning rate of the neural network.
  - ε : The exploration-exploitation parameter. This parameter decays from 1 to 0 with training.
  - γ : The reward decay factor.
  - batch_size: The minibatch size that we construct by randomly sampling the replay memory
  - epochs: Number of epochs/games to train the agent for.
- Run the training on a small number of games and tweak the hyperparameters/network architecture until we see somewhat good results. We follow these steps for training:
  - Initialize the network with some small random weights.
  - Given a state *s,* choose a random action with probability ε or choose $argmax_{a'}(Q(s, a'))$ with probability 1 - ε .
  - Observe the transition *<s, a, s', r>* and store it in the replay memory.
  - Get a random minibatch of transition states from the replay.
  - Forward propagate the state data through the network and calculate loss as described in the loss function above.
  - Calculate the gradients and backpropagate them to adjust the weights.
  - Repeat until epochs end/loss tapers off.
- Use the adjusted hyperparameters and model to train on a much longer simulation. We will train until either the number of epochs are completed or until the loss tapers off, whichever is earlier. We will also save the model checkpoints every time the loss decreases by some significant amount and whenever the model achieves the minimum loss observed so far.

- When done training, test the network in ‘human’ mode to see how well the bot plays. Submit the model to OpenAI for evaluation and see whether the metrics are satisying or not. Tweak and train the network for a longer duration to improve results.

References

[1] van Otterlo M., Wiering M. (2012) Reinforcement Learning and Markov Decision Processes. In: Wiering M., van Otterlo M. (eds) Reinforcement Learning. Adaptation, Learning, and Optimization, vol 12. Springer, Berlin, Heidelberg
[2] Watkins, C.J.C.H. & Dayan, P. Mach Learn (1992) 8: 279. https://doi.org/10.1007/BF00992698
[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).
[4] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.
[5] Tambet Matiisen, Demystifying Deep Reinforcement Learning, https://www.intelnervana.com/demystifying-deep-reinforcement-learning/
[6] https://gym.openai.com/envs/MsPacman-v0/
[7] dgriff777's algorithm, https://gym.openai.com/evaluations/eval_8Wwndzd8R62np8CxVQWEeg/
[8] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." *International Conference on Machine Learning*. 2016.