Coordination Based Deep Reinforcement Learning

Trevor Morton
trevorcmort@gmail.com
Rose-Hulman Institute of Technology

Abstract

In this paper, we will present a deep reinforcement learning system architecture that uses multiple reinforcement learning agents working in tandem. These agents are comprised of fully connected layers within a larger neural network whose decision making and training algorithms combine aspects of multi agent systems design and existing deep reinforcement learning methods. This design is aimed at lowering the convergence time for scenarios where multiple actions can operate at once. We apply our design to the game Super Smash Bros. Melee running in the Dolphin Game Cube emulator.

Introduction

Recent advancements in Deep Reinforcement Learning have shown great promise for developing algorithms that can learn policy representations based on both high and low dimensional input. These application cases span from control systems to playing video games, and with the generality of what it takes to formulate a task as a reinforcement learning problem research in this field has high promise for numerous application domains. Many of these architectures are based on the premise that each possible action that can be chosen for a given state should be directly correlated to an output on a neural network. While this strategy has shown to be effective for many situations, it does not extend well to some more complicated domains. Take for example the deep reinforcement learning architecture presented by Mnih (Mnih et al. 2013). In their work with Atari 2600 games they make the assumption that the number of actions that can be preformed at any time is singular, or that for any given state only one action can be chosen with which to procede to the next state. As such their network that interacts with the game has one distict output for each action possible in game. This model does not extend efficiently to scenarios where for a given state a set of actions can be chosen with which to proceed to the next state with. As an example take the scenario of controlling a robotic arm on an assembly line that has two motors. In the architecture presented by Mnih, each combination of motor movements would need to be given an output on the neural network and for any given state input a single motor movement combination would be chosen based upon a single network output. Although this can be an effective model, it creates the necessity for a combinatorial number of outputs to be created on the network. When we say there must be a combinatorial number of outputs, we mean that there must be an output relating to every combination of moves the motors can make. Stepper motors that move most robotic arms can have hundreds of positions on each of them, and even a robot arm with two motors and a hundred positions on each motor would need to encode ten thousand outputs on a network, a number that is entirely infeasible.

In scenarios such as the robotic arm where multiple actions can be chosen simultaneously, multi agent systems have been shown to preform very well. By delegating the decisions for each exclusive set of actions to be managed by individual agents, reinforcement learning systems can be constructed that more efficiently model scenarios with multiple simultaneous controls. These categories of multi agent systems have been shown capable of learning effective policies from both high and low dimensional input, and so it makes sense to consider the use of these types of systems when extending deep reinforcement learning to such a scenario. These systems also have a rich research background, with many fields intersecting in their research including game theory and dynamical systems.

By combining aspects of both deep reinforcement learning architectures and algorithms used to train multi agent reinforcement learning systems, we aim to construct a system architecture that can learn, from high dimensional input, policy representations for scenarios that require simultaneous actions to be chosen.

Related Work

One of the most major recent successes in deep reinforcement learning is Playing Atari Games With Deep Reinforcement Learning (Mnih et al. 2013). Using a basic convolutional network design, they show that an effective gameplay policy can be learned from high dimensional image input and that superhuman levels of play can be reached on some games for the Atari 2600. They further expand upon this work, showing that by using a target network design and only periodically updating the gameplay network the convergence time can be greatly reduced (Mnih et al. 2015). They also show their designs to be very flexible in terms of application, with the same training algorithms and net-

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

work architectures being effective at learning over 50 different Atari 2600 games. The generality of this design leaves a lot of promise in extending the work and expanding its use case.

We argue that multi agent system design can be applied to deep reinforcement learning systems, which we can see is true from the work of Tampuu (Tampuu et al. 2015). Tampuu takes two deep reinforcement learning agents and pits them against each other with various reward schemes and shows that the networks predictably react to being part of a system with a game theory oriented reward scheme. In this case, it was pitting two deep reinforcement learning agents against each other in a game of Pong. By modifying the reward scheme in various ways, Tampuu was able to teach the networks to coordinate with each other to keep the ball in play forever, and was also able to teach them to play as competitively against each other as possible. The resultant play styles of the agents were predicted by the basic game theory principles that governed the choice of their reward schemes. This shows that deep reinforcement learning agents can react and learn gameplay policies in a fashion predicted by game theory principles. This is important, as without this finding it would be unknown whether incorporating multi agent system aspects would alter the properties or stability of these deep reinforcement learning algorithms. Many of these algorithms are already known to be very difficult to train to convergence and the knowledge that these systems won't ruin the convergence of our algorithms is key.

Much of our design for the multi agent system itself is taken from Guestrin (Guestrin, Lagoudakis, and Parr 2002) and Kok (Kok, Spaan, and Vlassis 2003) who present the idea of coordination graphs for organizing agents that need to communicate, and the algorithms for training these agents in reinforcement learning scenarios. Coordination graphs detail which agents can communicate with each other and simplify the decision making process by organizing the agents in a structured fashion. Most importantly Guestrin's extension of q-learning to scenarios where simultaneous actions are performed is very much needed to extend deep reinforcement learning to this same scenario. By taking advantage of this coordination graph setup, agent decisions can effectively propogate to other agents and coordination can be formed through q-learning.

Communication is a major component of multi agent systems, as it enables game theory concepts to be applied and for more complex agent architectures to be implemented. Much research has been put into communication schemes for agent communication, but we will use the work of Foerster (Foerster et al. 2016) to design communication between our neural network agents. Foerster shows that in deep reinforcement learning scenarios by feeding both the environmental inputs and the decisions made by other agents into a neural network agent, agents can learn to base their actions off the agents before them and develop coordination based policies. Although Foerster also presents another communication scheme where the networks share full differentiable connections with each other, we found that this scheme disrupted the q-learning algorithm's exploration component. By making it difficult for the dependent agents to decide what action the agent before it had taken in cases where a non-optimal action was chosen by the algorithm. Alhough this may limit the depth of the communication channels agents can take advantage of, it showed effective enough in our experiments.

Cordination Based Deep Reinforcement Learning

We present in this paper a deep reinforcement learning architecture that is capable of efficiently modeling scenarios where simultaneous actions can be taken. We will use the video game Super Smash Bros. Melee (SSBM) as our game of choice. This game is ideal as for every frame there are serveral sets of controls that can be operated in tandem with any other, giving us several exclusive sets of actions. Although a scenario like this can be modeled in the same way as Mnih does, it would require either limiting the model to only using one control at a time or creating an output for each possible combination of controls in the game. This is clearly not ideal as in the former scenario the skill level of the model is limited by not being able to use more than one control at once in a game designed to have mutiple controls used simultaneously. While the latter scenario is in theory capable of reaching a high performance level, in the scenario of SSBM it would require 1296 outputs to encode all the basic controller combinations, and that is in a simplified scenario where the joystick controls are locked to the 8 cardinal directions and only 6 of the most relevant controls are considered.

This game has been studied already by (Firoiu, Whitney, and Tenenbaum 2017), and they have shown it is possible for deep reinforcement learning agents to reach professional levels of play when fed the internal game data as input. They use a simple network design of two hidden layers of size 128 and 54 outputs, corresponding to the fact that their agent was only capable of choosing from 54 possible actions. These possible actions were chosen by people with intimate knowledge of the game and although it allowed their agent to play using most of the major move combinations a professional player would it leaves many more subtle move combinations out. Despite the success of their system, interacting with an evnironment based off the internal game data is a very different problem from playing off visual input.

In order to more efficiently model this scenario, we will extend the methods Mnih et al present and create a multi agent system. In this system, we will have an agent for each distinct set of actions in the game that can be chosen at the same time. We will link the agents together in a directed acyclic coordination graph based on their inherent dependencies from the game. From this coordination graph we can construct a single neural network whose structure reflects that of our coordination graph. We can then combine the training algorithms outlined by Mnih and Guestrin to train a single neural network to converge on a policy.

The coordination graph we will make differs from that Guestrin outlines (Guestrin, Lagoudakis, and Parr 2002), as ours will be directed and acyclic. This difference is necessary as Guestrins algorithm is presented in a more abstract

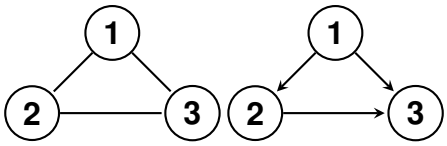

Figure 1: Two versions of the same coordination graph. The graph on the left is representative of three agents that can all communicate with each other via Guestrin's design. The graph on the right is the same graph converted to a directed acyclic graph as we will use in our design.

way where each time we evaluate our agents decisions there does not always need to be a definitive order in which the agents are evaluated. We will use the control of a robotic arm as an example again. Let's say there are three motors on the arm that need to be contolled. Controlling them with a multi-agent model, we can create three agents one for each motor. Linking them together in a coordination graph we make undirected links between each agent saying they can all communicate about what action to perform as seen in figure 1. While this may work with Guestrin's algorithm, in order to make a feed forward network based on this graph we must make it acyclic as our neural network is not recurrent. When converting our graph, there must always be at least one agent who's decision can be made without knowing what other agents will choose, and for that reason we need a directed acyclic graph so we can always have a root node as seen in figure 1. The directed component is important as well as it will allow us to more explicitly define which actions we want to depend on other actions from our coordination graph.

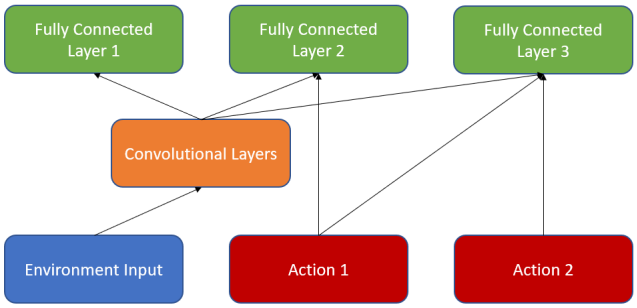

Figure 2: Network architecture as we propose. Action 1 and action 2 are boolean arrays correlating to which action we have chosen from agents 1 and 2. By setting these network inputs relative to the chosen action we can train network components to evaluate action reward based on actions other agents are taking.

When there is only one set of exclusive actions that can be considered, the neural network architecture is identical to that of Mnih (Mnih et al. 2013). This means that when there is only one node in our coordination graph, we will construct a neural network with several convolutional layers at the beggining and a single set of fully connected layers on the end outputing the predicted reward for each possible action. Careful attention has been payed to ensure that in the case only one set of actions is being chosen from the work of Mnih can be used as a base case. There have been numerous works based on these methods, and by staying as close to them as possible we can use many of the optimizations made for them.

In the case where there are multiple independent sets of actions to be chosen from and multiple nodes in our graph, each node corresponds to a set of fully connected layers on the end of the network. The inputs to each nodes fully connected layers are both the outputs of the convolutional layers and a boolean array relating to the actions chosen by the agents that our current agent is dependent on in our coordination graph. An example of this can be seen in figure 2, where a network created from the coordination graph in figure 1 has been created. With the method we architect this network, there is always a clear order to how these agents outputs should be interpreted. With the actions chosen by previous agents being network inputs and not passed internally to the network, during the exploration phase of our algorithm where we choose non-ideal actions to explore our environment more we train our network by feeding as input the action we have chosen instead of that the agent thought best. We found this to be necessary, as without it our network catastrophically diverged as the different agent components could not learn which actions agents were going to choose during exploration using network outputs directly as inputs.

We then make our decisions by using a modified form of the q-learning algorithm presented by Guestrin (Guestrin, Lagoudakis, and Parr 2002). Modification is necessary to the algorithm presented, as in the case of Guestrin each agent has a seperate decision function. This is not the case for our architecture where all agents will be a part of the same larger neural network. Our algorithm will then mirror Guestrins, except instead of setting an agents inputs and then evaluating it to find the decision it makes we will set our global neural networks relevant inputs taking into account any decisions made so far, evaluate the entire network, and interpret the outputs relating to the agent who's decision we are currently trying to find. With these changes in mind, we have the following algorithm that is used in order to make a single decision.

$G = CoordinationGraph$
$N = topoSort(G)$
$A = \emptyset$
$X = currentstate$
$p = choiceprob$
while $N \neq \emptyset$ **do**
    $n = N.poll()$
    $an_{max} = max_{an} Q(X, A), an \in n.actions()$
    if $random() > p$ **then**
        $an_{max} = random(a)$
    end if
    $A.add(an_{max})$
end while

The complexity of this algorithm is dependent on the

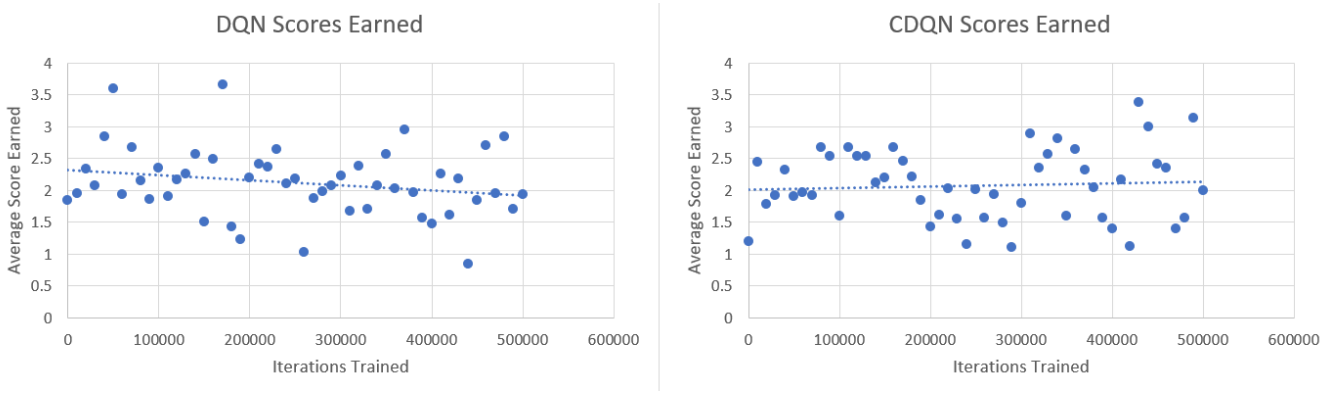

Figure 3: Average positive score earned per iterations trained at batch size 32. Notice the high amount of noise in both sets of performance statistics, as even our baseline DQN failed to converge for Super Smash Bros. Melee. Although our algorithm, CDQN, performs slightly better and appears to have an upward trend when compared to DQN in this environment neither indicate any trend that gives strong evidence for which performs stronger.

number of nodes in our graph. While not ideal, because we are potentially making multiple evaluations of our neural network inside this algorithm the number of times we must evaluate our neural network is the true efficiency of this algorithm. The effective runtime of this algorithm can be cut down by storing the network outputs whenever it is evaluated and only actually re-evaluating the network once a node is reached that requires a re-evaluation. For example, if we were to have two nodes only dependent on our root node there is no need to evaluate the network three times when making their decisions, as the required inputs for the dependent nodes are identical. While this doesn't help the worst case efficiency, the coordination graph can be designed in such a way to minimize the number of dependencies and as such minimize the number of times the network must be evaluated.

It is important to note that when saving data to our experience replay, only the boolean arrays that relate to the final set of actions chosen are kept and used as inputs. No other version of this array is necessary due to our network being designed off of our coordination graph. At any given point in our decision making process we are only making decisions for agents that have had all agents they are dependent on make their choices. This guarantees that we can safely interpret the relevant agent's network outputs as all inputs that connect to them will have been set.

We can take this algorithm and use it to replace the step in Mnih's algorithm where either a random action or the singular maximum action is chosen (Mnih et al. 2013). With this being the only change, integrating performance optimizations such as target networks and prioritized experience replay becomes trivial. With both the ability to choose an optimal set of actions for a given state and update our q-value approximators to more accurately predict from observed results, we have a novel neural network architecture capable of learning and selecting actions in a scenario where there are multiple exclusive sets of actions.

Experiments

In our experiments, we have trained agents for SSBM giving them control of two of the main in game controls, the main movement stick and the A button. These controls were chosen because movement is necessary for any of the most basic navigation in the game, and with the A button and movement stick used in combination the highest damage dealing attacks in the game can be triggered. We found this to be important, as when we gave our agents only controls with little damage dealt or little potential to KO the oponent (which had the highest reward) the agent struggled to find correlation between action and reward.

Baselining with traditional DQN, we found that trained agents performed very poorly. Fully "trained" agents were lucky to survive a matter of seconds before they found suicide to be the most rewarding action to perform, and they showed no reaction at all to enemy movement on screen. Plots of the in game score earned relative to how long the network had trained showed virtually no upward trend in the agents learning, and could not beat even the score earned of an agent that did not react to anything and allowed itself to be pushed off the edge of the map. After many trials with diferent network and input configurations we found that the baseline DQN could only begin to learn strategies once the resolution the game was scaled to was doubled when compared to the input sizing of the agents from Mnih et al. This was not a surprising finding as Atari 2600 games, which Mnih's network is optimized for, are played at a resolution of 160x192 pixels. These screen images are then downscaled to 84x84 when input into the neural network. This is a downscaling of about 2.28 times relative to the images largest dimension. As such it can be said that roughly half of the image data is lost in the downscaling. SSBM is played at a resolution of 640x480, and so downscaling this image to 84x84 gives a downscaling factor of 7.61. This is a significantly higher amount of information loss, and so it is a logical conclusion that with Mnih's convolutional architecture agents wouldn't be able to learn an effective policy.

By doubling the network resolution and doubling the filter size of the first convolutional layer to 16x16 we were able to train individual agents whose gameplay strategies began to approach that of what a human observer might think was a child trying to play the game. While this is encouraging, the data in figure 3 still indicates no upward trend in training and only outliers that can perform at basic levels. Although much of the literature indicates that a longer training time would lead to higher rates of convergence, we observed no emerging trend even when training for upwards of 5 million iterations, or ten times as long as the data gathered above. Several different network architectures were attempted in order to improve these figures, including switching the first convolutional layer to 16x16 filters and using several separate channels of convolutions. We found that using several channels improved our agents ability to react to unique features in the game, but did not raise the performance enough to justify the several magnitudes more of parameters in the network.

When we run our algorithm using this configuration we found that it reaches aproximately equal levels of play to our DQN implementation. Despite the high amount of noise in the training, these results indicate that our algorithm at the very least does not have catastrophic behavior and converge to nothing. More importantly, it indicates that our algorithm for selecting the maximal action is viable, as despite low levels of performance we observed our agents capable of exhibiting diverse sets of actions during gameplay. Both sets of trained agents had networks that could play at basic levels but the data gathered also clearly indicated a global struggle to converge to a cohesive policy in this game.

Conclusion

The literature on DQN has never tried to hide that convergence is delicate at best and for some games non-existent. It is known that not all algorithms work as well as others on reinforcement learning tasks due to the extremely nonlinear nature of their function spaces. Although work has been done on SSBM in the past, it has potentialy an entirely different problem landscape when turned into a computer vision problem. As such, our game of choice may have been doomed from the start with no way to have known ahead of time that our algorithm was innapropriate for it. Applying our algorithm to Atari games like many other algorithms benchmark themselves on may have been a better choice of domain, but we feared that choosing a domain wih too simple of control combinations would not highlight the promise of our algorithm.

We believe that the main struggle of our algorithm lied mainly in our network architecture as it was meant for a game with a significantly lower resolution. All of our combinations of convolutional networks we tried failed to converge to a high level of game play and although we in the end had agents that could competitively play against the lowest leveled in game AI's they were encountered effectively through a random search of the problem space. With no positive trends in nearly any of the gathered data for even benchline versions of DQN we believe that learning SSBM from visual input is a significantly more challenging problem than we initially thought.

Another reason that SSBM could be more difficult than we originally thought is due to a feature of the game where the in game perspective changes. This means that as different players get closer to each other, the camera zooms in on them. A result of this is all the in game features that a convolutional network would learn to pick out change scale timestep to timestep, effectively changing what features the network should be attempting to identify. While to a human that is used to viewing objects from different distances this is trivial, all Atari games that are traditionally benchmarked against have a fixed camera position. While this may simply be fixed by a different convolutional architecture, we did not have the computational resources or expertise to determine how this could be solved. There are some resources indicating that through in game hacks the camera position could be fixed, but due to the already fragile state of our emulation environment we could not get these hacks to work. Further reason we have to believe that this could be one of our issues is by observing agent gameplay patterns. Many trained agents tended to excel at playing their opponent and navigating the stage when the camera was fairly zoomed in, but once the camera zoomed out would commit in game suicide and run off a stage edge. In observed game play, self destruction was in fact the only source of agent deaths. While not all these self destructions resulted from the camera zooming out, a large portion of them did and as such we believe this to be a strong source of the blame for convergence instability.

Further work in this will require a change in problem domain, a task we did not have the time to explore as our environment was designed for GameCube emulation and we could not easily pivot to a more reasonable domain. Even if a pivot was feasible, very few video games used for reinforcement learning benchmarking have complicated combinations of controls they take advantage of due to the fact that the older systems that can be quickly emulated have simpler controller inputs. Eventualy once we can emulate more advanced gaming consoles at higher speeds this problem may be possible to tackle again, but simulations that will fully take advantage of what we hope our algorithm can provide are not currently available in video game form.

References

[Firoiu, Whitney, and Tenenbaum 2017] Firoiu, V.; Whitney, W. F.; and Tenenbaum, J. B. 2017. Beating the world's best at super smash bros. with deep reinforcement learning. *CoRR* abs/1702.06230.

[Foerster et al. 2016] Foerster, J. N.; Assael, Y. M.; de Freitas, N.; and Whiteson, S. 2016. Learning to communicate with deep multi-agent reinforcement learning. *CoRR* abs/1605.06676.

[Guestrin, Lagoudakis, and Parr 2002] Guestrin, C.; Lagoudakis, M. G.; and Parr, R. 2002. Coordinated reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, 227–234. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Kok, Spaan, and Vlassis 2003] Kok, J. R.; Spaan, M. T. J.; and Vlassis, N. 2003. Multi-robot decision making using coordination graphs.

[Mnih et al. 2013] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing atari with deep reinforcement learning. *CoRR* abs/1312.5602.

[Mnih et al. 2015] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518:529 EP –.

[Tampuu et al. 2015] Tampuu, A.; Matiisen, T.; Kodelja, D.; Kuzovkin, I.; Korjus, K.; Aru, J.; Aru, J.; and Vicente, R. 2015. Multiagent cooperation and competition with deep reinforcement learning. *CoRR* abs/1511.08779.