THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

EXTENDING THE TEMPORAL RANGE OF HIERARCHICAL DEEP REINFORCEMENT LEARNING

ANDRES DE LA FUENTE DURAN
SPRING 2019

A thesis submitted in partial fulfillment
of the requirements for baccalaureate degrees
in Computer Science and Philosophy
with honors in Computer Science

ABSTRACT

In Reinforcement Learning, a long-standing problem is that of extending the temporal range of an agent, or in other words the capability to learn effective behaviors in environments where positive feedback has a long delay. Conceptually, this is analogous to the notion of long-term strategy or planning. In this paper I propose an approach for improving temporal range which is built on a few recent developments. One such development is the successful creation of a Neural Network model for use in Reinforcement Learning, called the Deep Q Network. Despite the success of this model in achieving cutting edge performance, it still faces the problem of long-term delayed rewards. Recent work has proposed a hierarchical organization of Deep Q Networks in order to achieve a certain level of temporal abstraction. This has shown to be an improvement over a single Deep Q Network in the context of long-term delayed rewards. However, it still relies on the Deep Q Network, which has a weakness: its updates (which are when the weights of the model are altered as a response to new observations) are only based on limited information. More specifically, its architecture does not have a built-in persistence of information. Consider trying to read this paragraph while only being able to remember a few random words you have read so far. Recurrent Neural Networks are designed to have long term information persistence as a result of their structure. Here I propose a model which incorporates a Recurrent Neural Network called ‘Long Short Term Memory’ into the Hierarchical Deep Q Network approach with the goal of improving temporal range.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Vasant Honavar and Dr. John Hannan for their insightful and critical feedback throughout the process, helping me to find good literature and to formulate my project more thoughtfully.

I would also like to thank Google DeepMind for putting out accessible publications on their recent work; I would not have discovered the subject of this paper without them.

Introduction

Reinforcement Learning (RL) has historically posed a particular set of problems in application which have motivated much of the recent work that informs this paper. Among these are the problems of dealing with very high dimensional inputs, and that of learning temporal abstractions. Many approaches to both of these problems have been proposed in recent literature – this is covered in the literature review that follows, along with some explanation of RL in the context of this paper. In this introduction I present a little more background in order to give a sense of purpose to the entirety of the review.

First, the general class of problems RL deals with is that of 'agent control'. Specifically, this means the problem of achieving desirable decision-making behavior in a computer agent. Desirability is defined in terms of some kind of feedback from the agent's environment. Before getting into the details, this means that an RL agent must take in observations of its environment, and use that information to make decisions.

The problem of high dimensionality refers to cases in which these observations are simply very large. Consider an RL agent which is being trained using video data. This could be an agent learning to control a robot, or learning to play a videogame. Such an agent might be receiving hundreds or thousands of input values each time it makes an observation (think of all the pixels from a few frames of video, and then the different values each pixel represents). Huge amounts of input information such as this can be too computationally taxing on RL agents.

To deal with this issue, non-linear value function approximators, such as Neural Networks, have been suggested. [1] [2] [3] The value function is a core component of RL explained in the review. While other suggestions involve reducing the state space (another core component of RL) using abstractions such as ‘contingency awareness’, which refers to regions of observation that are most contingent on the agent’s actions [4], state space reduction is a path of inquiry I will not focus on here. The non-linear value function approximations are what is primarily of interest to this paper, in particular the Deep Q Network. Accordingly, the review begins by covering Neural Networks, Reinforcement Learning, and the Deep Q Network.

An approach to temporal abstraction which is based on this model is the Hierarchical Deep Q Network. This alteration organizes two Deep Q Networks into a hierarchical structure, with one network learning behavior over simple actions and the other learning goals which are used to ‘motivate’ the simple action selection. [8] This is the base for my proposed model.

Motivation

Even though HDQN is an improvement towards the problem of RL in delayed reward environments, it faces some challenges. Besides the problem of disentangling features from raw pixel data, there is room to introduce some notion of long term ‘memory’ to the model. As is, the model bases every decision only on a limited history, as it is built using the original DQN model. In this paper I propose and evaluate a model that attempts to give the HDQN a longer ‘memory’, so to speak. I introduce a type of neural network called LSTM to the internal structure of the DQN being used within this hierarchical model. The motivation is to improve the performance of the HDQN in delayed reward scenarios.

Acronyms

The subject of this paper requires the repeated use of cumbersome terminology, such as Hierarchical Deep Q Networks, Long Short Term Memory, Deep Convolutional Neural Networks, and so on. Below is a table of acronyms used for repeated terms throughout the paper. The full terms are used when they are introduced, but are otherwise replaced by the following.

Table 1: Acronym List

Term	**Acronym**
Reinforcement Learning	RL
Deep Q Network	DQN
Hierarchical Deep Q Network	HDQN
Neural Network	NN
Convolutional Neural Network	CNN
Deep Neural Network	DNN
Recurrent Neural Network	RNN
Long Short Term Memory	LSTM
Markov Decision Process	MDP
Rectified Linear Unit	ReLU

Literature Review

To provide background for the introduction of Recurrent Neural Networks into the Hierarchical Deep Q Network architecture, I review the following topics.

First, I cover the subject of Neural Networks, Convolutional Neural Networks, and Deep Neural Networks; all being key to understanding Deep Q Networks.

Second, I review some basics about Reinforcement Learning, and what formulation of it is being used here.

Third, I review the Deep Q Network itself, covering its design, motivation, successes, and shortcomings.

Fourth, I review some approaches to the temporal abstraction problem that are relevant background for the Hierarchical Deep Q Network.

Fifth I review some of the neuroscience and cognitive science behind the idea of hierarchical learning.

Sixth, I review the Hierarchical Deep Q Network, covering it in detail as it is the main foundation for this paper.

Finally, I review Recurrent Neural Networks and the Long Short Term Memory model.

Neural Networks

The Neural Network (NN) is a well-established and extensively documented model in machine learning. My intention here is not to provide a deep review of its history or development

– rather just the relevant definitions and background to discuss the Deep Q Network (DQN) model. Since the DQN is a ‘deep’ ‘convolutional’ Neural Network, I will briefly explain what an NN is, what a deep NN is, and what a convolutional NN is.

The basic premise of an NN is to (very loosely) mimic the sequential firing of neurons. Each ‘neuron’ is a node which takes many input values and produces one output value. Nodes are organized in layers, and each node receives the output values of all the nodes in the previous layer. How are these output values generated? There is a mathematical function which a given node in an NN uses to produce its output, referred to as the ‘activation function’. A commonly used activation function is the sigmoid, for example. In Figure 1 below, **z** is the weighted

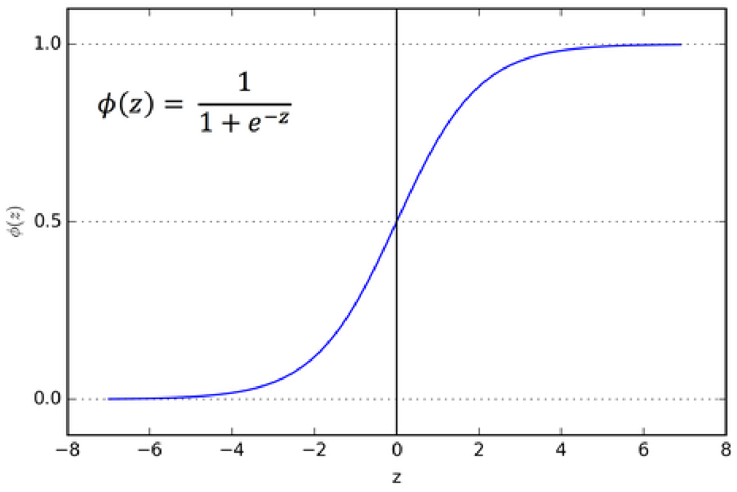


Figure 1: Sigmoid Activation Function

aggregation of the output values from the previous layer. This means that every node in a sigmoid layer (meaning all the nodes in such a layer use the sigmoid activation function), will produce a value between 0 and 1. If all the nodes in a layer use the same function, and receive every output from the previous layer, how come they do not all output the same value? The variation between each of their outputs is due to what are referred to as the weights. [9]

When each node is aggregating the outputs of all the previous layer's nodes to pass them through the activation function, each value is weighted. The weights assigned to each output from the previous layer are different for each node. In effect, each connection in the graph representation below (Figure 2) conveys the output value of the node it is coming from, and has a

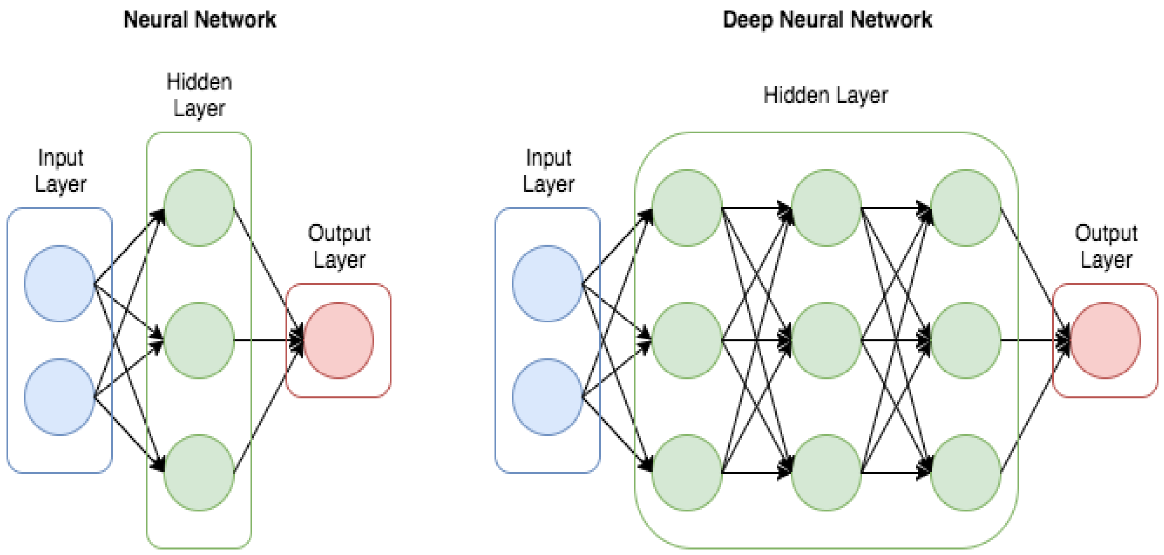


Figure 2: Simple and Deep Neural Networks

weight assigned to it. [9]

NN's are effective at capturing patterns, and have enjoyed particular success in problem domains such as clustering and recognition. The way this is achieved is that the weights are repeatedly altered in small amounts such that the output of the NN is increasingly desirable. This is usually referred to as 'updating' or performing 'updates'.

A Deep Neural Network (DNN) is simply an NN with more layers of nodes between the input and output. The idea is that the added depth allows for more abstraction of the input data, increasing the pattern recognizing power of the model.

One of the challenges for NN's in general has been the need to perform updates efficiently. In the case of the DNN's, the computational cost of doing this was prohibitive when they were introduced. Since that time, advances in design, update approaches, unsupervised learning, and computational power, have allowed DNN's to outperform other machine learning models in many pattern recognition tasks. [10]

Now I review the Convolutional Neural Network (CNN). A CNN is different from an NN in that each node is not connected to every node in the previous layer. Each node is only connected to a group of nodes which are next to each other in the previous layer. Figure 3 below visualizes this distinction.

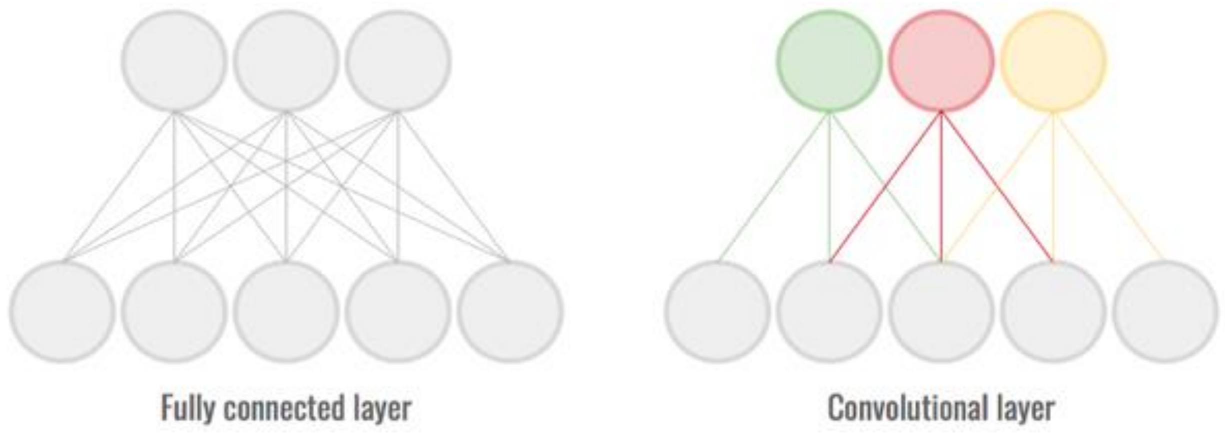


Figure 3: Convolutional Neural Networks

The 'tiling' of the CNN's layers allows it to take advantage of spatial information, which is especially useful when the input is an image or a stream of images (as is the case with an agent playing a computer game or navigating a real-world environment).

Deep CNN's are Deep Neural Networks which use convolutional layers rather than fully connected ones. They use these layers in a hierarchical structure to simulate the behavior of receptive fields, an important notion from research done on the visual cortex. [11] These Deep CNN's are fundamental to the design of the DQN, since they have been successful in both making use of spatial patterns and being more robust to natural transformations such as perspective shift.

Reinforcement Learning

Reinforcement learning is a formalization of the problem of agent control that usually defines a set of **states** the agent can be in, **actions** the agent can take, and **rewards** the agent can receive. The problem of control is turned into one of finding a policy which is used to select actions within an environment – the goal being to find a policy which would maximize expected future reward. [12]

This formulation is known as the Markov Decision Process (MDP). Formally, an MDP is a tuple (S, A, P_a, R_a). S is the set of all possible states. A is the set of all actions. P_a(s, s') = probability(s_{t+1} = s' | s_t = s, a_t = a), or in other words, P_a is the probability that a state s and action a at time t will lead to state s' at time t+1. R_a(s, s') is the reward for going from state s to s'. With these definitions in mind, the expected future reward mentioned above is based on P_a and R_a.

The formal goal is to find a policy π, which is a function that picks an action based on the current state (a = π (s)). The notion of expected future reward in RL is captured by what is called the value function. The value function essentially stores the estimated value of each state to the agent. This estimated value uses P_a and R_a to calculate the statistically expected reward of going to each state. The way the expected values are updated varies by approach.

To deal with environments where rewards can be delayed, an alteration has been proposed which adds goal states to the value function (the domain of the value function was previously just the set of possible states). [13] Policies can be generated that end when the agent reaches a goal state. These different policies can then be unified in different ways.

One such way is to make use of temporal dynamics, which is to say that the different policies operate over different time scales. This, combined with a hierarchical structuring, can produce effective behavior in delayed reward environments. [14] This is relevant to the design of

HDQN, which uses DQN’s to approximate these value functions which are organized hierarchically and dynamically with respect to time. [8]

Deep Q Networks

The DQN has been introduced in recent literature as a way to use deep learning in the RL domain. It makes use of the Deep CNN architecture. A specially designed Deep CNN is used to approximate the value function mentioned above. [2]

The motivation for the particular design of the DQN involves three major points. First, RL models do not perform consistently when non-linear function approximators (such as NN’s) are used for the value function; they can actually diverge due to this. Second, correlations in sequences of observation can throw off the model. Third, small changes to the value function can dramatically influence policy generation. [15] These latter two problems are causes for first problem (that of divergence).

The DQN makes use of two strategies to address these two root issues. The first is dubbed ‘experience replay’, after the name of a phenomena observed in biology. [16] [17] The biological observation is that memory is not used ‘linearly’ when being used for learning. In sleep, the brain appears to flash through disparate episodes. In context of this reinforcement learning model, the application of this observation is to randomly sample from some ‘history’ of events when making updates. This randomization in the context of updating the value function helps avoid any effects from correlations in sequences of observation. [2]

The second innovation of the DQN is having a ‘target’ value function which is only updated periodically. These ‘target’ values are used when updating the value function directly in use by the DQN. The target values act as a kind of anchor for the current values, as the difference between the current values and these targets is used in calculation of new values. The effect of incorporating this separate set of values is to mitigate the influence that small updates to the value function can have on policy, since the target values are not updated every step. [2]

Figure 4 below (credit to [2]) shows the structure of the DQN. The convolutional layers are used to abstract information from the image input; as discussed previously, DCNN's are very effective with visual data due to the advantages of convolution. Traditional fully connected layers are then used to eventually output a value for each possible action.

This diagram is also useful to note where my proposed alteration will fit in. I replace one of the fully connected layers with a recurrent NN called an LSTM, which introduces a long-term memory to the internal structure of the DQN.

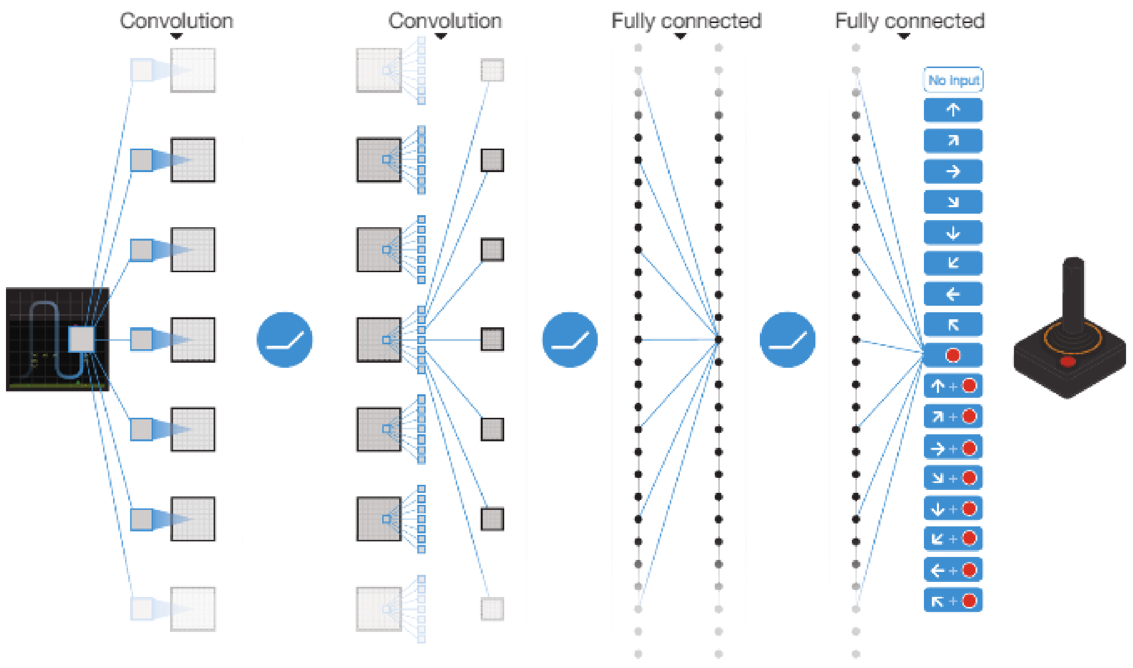


Figure 4: DQN Internal Architecture

The left part of this diagram shows the way each 'filter', or piece of the convolutional layer, relates to the input image. Each internal layer except for the last one is a Rectified Linear

Unit (known as ReLU). This simply means that the activation function (as discussed previously) is **max(0,x)**. The benefits of ReLU is that it avoids complicated math in the activation, and it ignores negative inputs, meaning that a given node is not activated every time. [18] These things make it very efficient versus other schemes, which is especially useful in this context of deep convolutional networks.

In the overall DQN there are three convolutional and two fully connected layers. The last is linear rather than ReLU, since it is meant to output the action selection. I am replacing the first of the fully connected layers with an LSTM network.

Temporal Abstraction

A conceptual subject of recent interest to researchers in RL is the idea of temporal abstraction. There is a kind of intuitive notion that humans are able to abstract action across time. For example, with the goal of going to the bathroom, we do not think in terms of taking individual steps that reduce our distance to the bathroom. We can abstract the thought of traversing a sequence of rooms or hallways. It makes sense that achieving something like this in machine learning might improve performance in situations where feedback is delayed.

Recent work has proposed an alternative to the typical Markov Decision Process (MDP) which can introduce a kind of temporal abstraction. Traditionally, MDP’s are the RL framework I described before, which includes a set of states, actions, transition probabilities, and rewards. Sutton et al. have suggested what they call options. Options are policies for action selection over a period of time. [14] These options can be used interchangeably with simple actions in the context of reinforcement learning. An agent can choose either a primitive action or an option at a given timestep. The importance of this relates to the problem of exploring the state space.

Being an old problem, the approaches to it are very diverse. Traditional ε - greedy approaches assign some probability to whether the agent will decide on the next action or pick randomly. Boltzmann exploration offers an approach that, although still favoring the optimal action when deciding, weights the others based on their estimated values. Unlike ε - greedy, the non-optimal actions are not treated equally. [5] Thomson sampling instantiates the agent's estimations randomly every time a selection is made. [6] It is interesting to note that these are all approaches to the Bernoulli Bandit problem which was formulated in the 1930's and which has resurfaced specifically in the context of RL. Some recent literature has elaborated upon these approaches in the RL domain. [7] A prevailing problem with such exploration is that it still only deals with 'primitive actions', or actions at the lowest level of granularity specified.

Options aid in exploration of the state space by mitigating the effects of increasingly large state spaces. In Figure 5 below (credit [19]), imagine the agent as being at the root of the tree. Each connection represents an action and consequent transition to a new state. In the figure, there is a goal state, and the arrows represent the sequence of actions needed to get there. Options turn the problem of learning seven decisions into a problem of two. Part A visualizes the

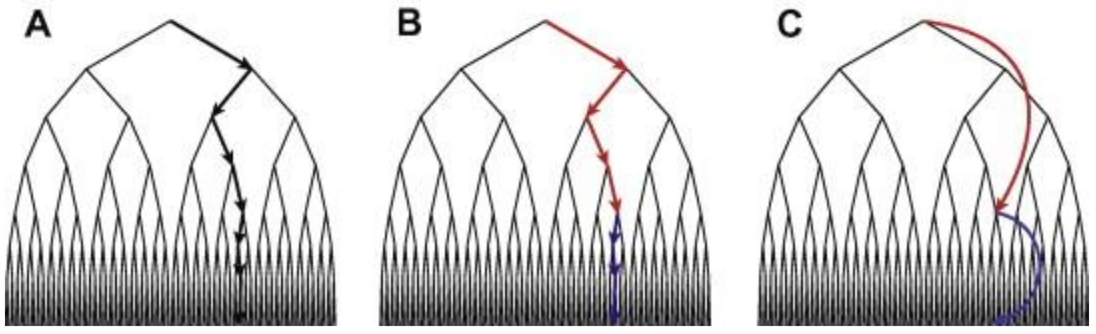


Figure 5: State Space Searching

‘primitive’ actions required. Part B visualizes the groupings by color of these actions under options. Part C visualizes what the model can now account for (simply two options). [19]

There have been various approaches proposed for learning options in real time. One idea is to introduce various reward functions (whereas traditionally there is only one). [20] Another proposition is to compose various options into new options in a way that does not detract from the original options’ usefulness. [21] Increased exploration to parts of the state space that have not been within reach before has also been suggested. [22]

The HDQN’s approach to temporal abstraction involves learning a policy for option selection as well as primitive action selection using DQN’s.

An interesting connection to this discussion of temporal abstraction is the traditional concept of time complexity in the theory of computation. Time complexity is defined asymptotically in terms of the repetition of what are taken to be primitive operations. For example, in the context of searching a list, comparing the value at an index in the list to some target value might be such a primitive action. If the search is performed by simply traversing the list from start to finish until a match is found, the repetition of primitive actions will scale linearly with the length of the list. In time complexity theory, this is referred to as O(n) time complexity, with n being the length of the input. This means that *at most* the operation will take as many repetitions of primitive actions as there are items in the list, n is the upper bound.

Time complexity is often presented in terms of searching or sorting information, such as the example of the list. If you will recall Figure 5, I was just discussing the search for a goal state in a tree representation of possible states. The difference between the search with and without the options is the manner of traversal through the tree. Options allow the search to essentially scale the time requirement to search the tree. In other words, temporal abstraction in this context can be thought of as attempting to interact with the time complexity of a particular search. Even though it will still be relative to the size of the tree, (therefore the O function will not be a different one), it is possible scale by some constant.

Cognitive Science and Neuroscience

In cognitive science, there has historically been a distinction between evolutionary psychology and enlightenment era ideas about human nature. The former, supposedly inspired by Darwin, suggests that the mind is a conglomerate of specialized mechanisms shaped by evolution. The latter view is of the mind as a single learning system which discovers regularities in experience. Recent literature has suggested an alternative: the mind has some ‘core knowledge’, and new skills and beliefs are constructed from these foundational blocks. [23]

This notion can be seen as related to the idea of options in RL. Options perhaps can be taken to be analogous to ‘core knowledge’. Taking this to be the case, what is left is the question of how these things are put together – the question of hierarchical learning. It is relevant to consider the ways in which hierarchical RL and cognitive science have influenced each other.

As early as 1951 in the realm of psychology, the assertion has been made that the selection of low-level actions requires a higher-level representation of goals. [24] Since then much work has been done regarding the possible nature of such representations. Recent publications propose a concept of ‘task representation’ which resembles options in a few ways: these representations can be selected, they remain active for some time, they cause the creation of a ‘policy’, and they can be hierarchically nested. [25] [26]

There have been some differences in the interests of psychology versus those of hierarchical reinforcement learning. In psychology, much attention has been given to the transition between tasks, the comparison of tasks, and the role of tasks as mechanisms for information persistence. The focus has not been so much their role in learning. [19]

Some works of observation have produced evidence for hierarchical learning in children. Essentially, it appears that throughout development, children exhibit behavior that incorporates previously known ‘simpler’ operations into larger wholes. [27] [28]

There are two relatively recent frameworks from psychology about the theory of cognition which relate to hierarchical reinforcement learning, Soar and ACT-R. These frameworks involve the idea of using ‘chunks’ (think of options) which can cause the execution of sequences of actions. [29] [30] The takeaway has been that these ‘chunks’ ease problem solving and speed it up. However, an important distinction is that reinforcement learning is ultimately only formulated in terms of reward maximization.

So cognitive science and RL have influenced each other in that the latter has motivated empirical observations (such as those of children’s hierarchical behavior), and ideas from cognitive science have inspired models in RL. In a similar way, neuroscience has been influenced by RL.

A formulation of RL called the ‘actor-critic’ model involves two eponymous parts: the *actor* learns a policy over actions and the *critic* maintains the value function. During execution, the *actor* takes actions, and the *critic* calculates prediction error. The prediction is simply taken to be the value of the state the actor just left by taking action added to the value of the new state. By comparing this to outcomes, the *critic* maintains the estimated values of each state.

In neuroscience, this actor-critic model has had an impact. For example, in the observation of neural structures, recent work has proposed identifying the actor and critic as parts of the brain, and the reward prediction error function as dopamine. [31] [32] [33] To read a more detailed summary of the relation of hierarchical RL to neuroscience, utilizing the more technical terminology from the field, see [19].

Hierarchical Deep Q Networks

To understand the HDQN model, first consider the traditional MDP formulation: states, actions, a transition function that maps states and actions to new states, and rewards. In the context of HDQN, this environment reward is now distinguished as the ‘extrinsic reward’. The stated goal of this model is to maximize the extrinsic reward over extended time. [8]

Goals are introduced to this traditional MDP formulation to capture the notion of ‘intrinsic motivation’. The agent’s purpose is to set and complete sequences of goals that maximize extrinsic reward. Options define policies for each goal, with the agent learning these ‘mini policies’ along with an optimal sequence of goals. A ‘critic’ (as in the actor-critic model described above) rewards the agent based on its successful completion of goals. [8]

There are two ‘controllers’ in the HDQN: the ‘controller’ and the ‘meta-controller’. The meta-controller takes the state as its input and produces a goal. The controller takes the state and

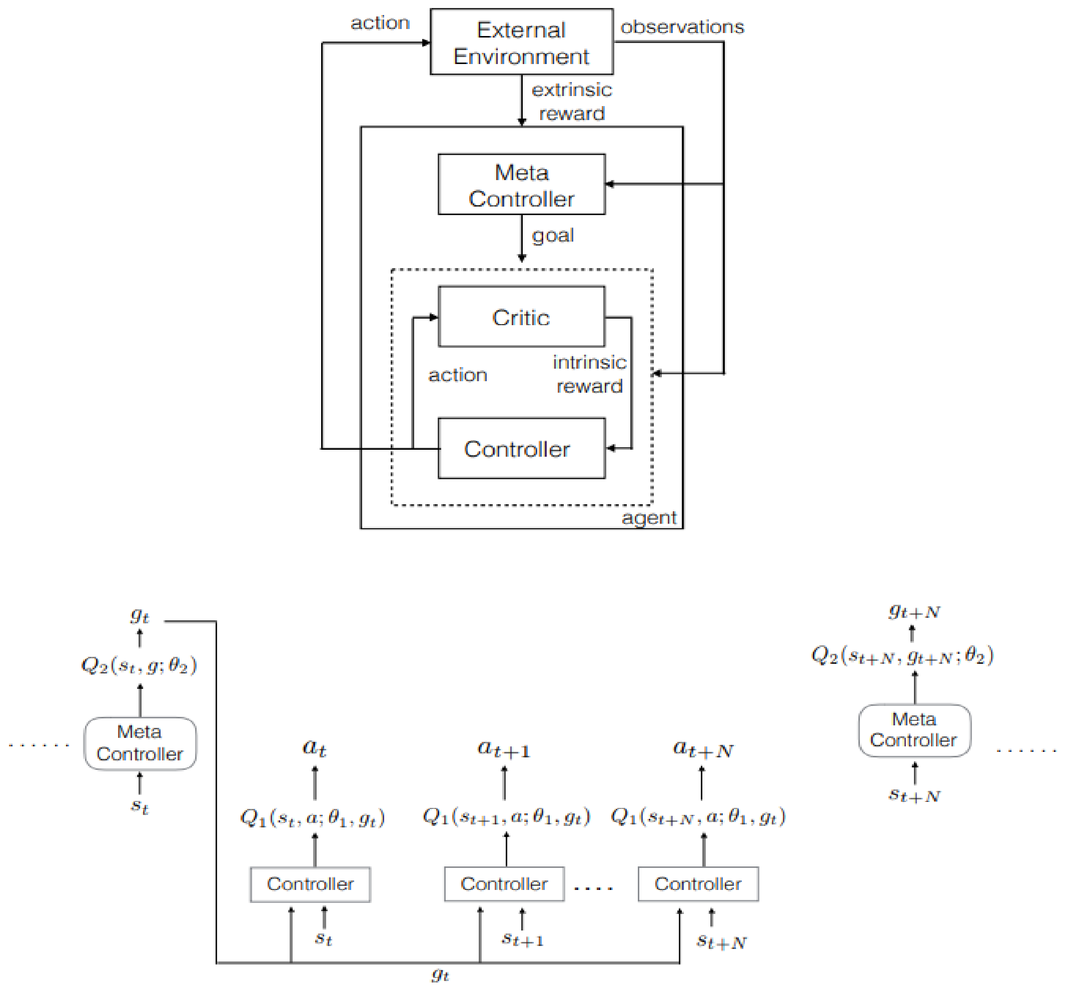


Figure 6: HDQN Overall Design

the goal as input and selects an action. The goal is not changed at every timestep; it remains for some previously specified time interval until it is achieved or the interval ends. The critic then provides a reward to the controller based on whether this goal was completed. The controller is attempting to maximize this reward, called ‘intrinsic reward’, over time. The meta-controller is attempting to maximize cumulative extrinsic reward. [8]

Figure 6 above (credit [8]) illustrates this hierarchy. The agent is ultimately taking actions and receiving feedback from the environment. This feedback comes in the form of extrinsic rewards and new states (observations). The new states are used by the critic to determine whether an intrinsic reward is earned. The extrinsic rewards and the new states are used by the meta-controller to learn its policy over goals. The meta-controller feeds its goal to the controller. The controller uses this goal and the new state observation to select a new action. The second half of the diagram illustrates the way a single goal is used across several time steps before being replaced, and how this is used as an input to the controller along with each successive state.

Moving on, the meta-controller and the controller are trying to maximize extrinsic and intrinsic rewards respectively. This is accomplished by using a DQN for each, with the Q value function being distinct. The controller uses the following (1) function:

$$\begin{aligned} Q_1^*(s, a; g) &= \max_{\pi_{ag}} \mathrm{E}[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} \mid s_t = s, a_t = a, g_t = g, \pi_{ag}] \\ &= \max_{\pi_{ag}} \mathrm{E}[r_t + \gamma \max_{a_{t+1}} Q_1^*(s_{t+1}, a_{t+1}; g) \mid s_t = s, a_t = a, g_t = g, \pi_{ag}] \end{aligned} \quad (1)$$

The meta-controller, alternatively, uses this (2) function:

$$Q_2^*(s, g) = \max_{\pi_g} \mathrm{E}[\sum_{t'=t}^{t+N} f_{t'} + \gamma \max_{g'} Q_2^*(s_{t+N}, g') \mid s_t = s, g_t = g, \pi_g] \quad (2)$$

The training requires the minimization of loss functions corresponding to each (1) and (2). Also, each controller has its own history of events to randomly sample from when updating. Stochastic gradient descent is the method employed, with the meta-controller only collecting experiences every N steps, and the controller collecting experiences every step. The meta controller selects new goals using an ε-greedy approach, with a modification that reduces the probability of exploration as learning is achieved. The controller's exploration takes into account the current success rate of reaching the particular goal g active at the time. [8]

Recurrent Neural Networks and LSTM

One of the problems faced by the HDQN is that as it relies on the DQN without major alteration, it is subject to the DQN's lack of information persistence. In fact, NN's in general face this issue, which has motivated the development of modified NN's which incorporate this notion of persistence. The Recurrent Neural Network (RNN) is a NN which, along with a usual output, produces a signal which is used as input, creating a loop. [34]

Figure 7 below (credit [34]) illustrates how this loop structure can be thought of as creating an array of copies of the NN. Each NN in the figure is actually just the original NN at each different time step of operation.

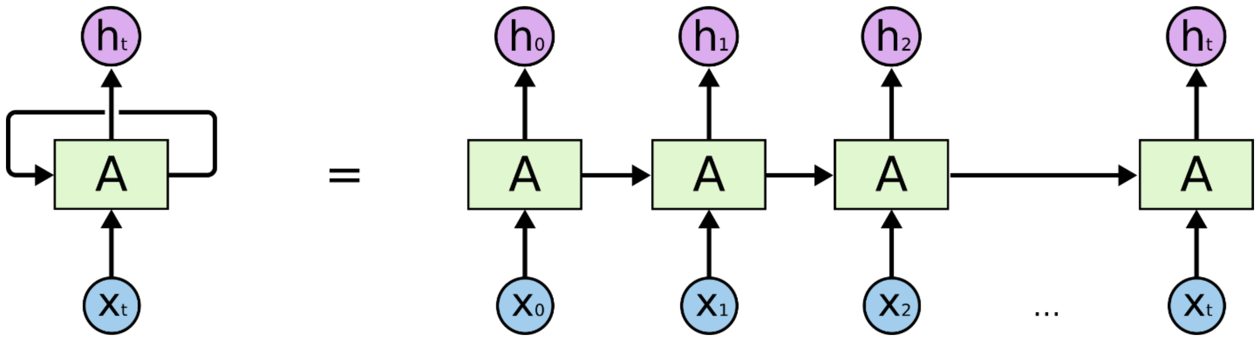


Figure 7: Recurrent Neural Network

To understand the way this is useful, consider the act of reading a sentence with the goal of predicting the last word. Depending on the sentence, different amounts of context might be necessary. For one sentence, the containing clause could be sufficient; for another sentence, it might be necessary to look all the way back to the first word. In Figure 7, the 'unrolled' display of the RNN can help visualize why this architecture would be expected to perform better in these contexts. The blue X circles are inputs to the RNN. It is possible that some of the information from the input at time 0 will be present in the RNN at time t, making it available for the output calculation at that timestep.

However, a flaw of the basic RNN is that it struggles to perform well as the duration for which information must be retained increases. In other words, an RNN does well if there are only a few time steps between the production of a certain piece of persistent information (which is placed on the looping signal) and *when* that piece of information is relevant. Long Short Term Memory (LSTM) is a more complex model of RNN which was created to address this issue; it has been very successful in doing so. [34]

One of the main innovations of the LSTM is that it controls interactions with the looping signal, referred to as ‘cell state’. In Figure 8 below (credit [34]), the cell state is the signal running along the top of each time step (the green rectangles). The red circles represent pointwise operations that combine signals, there are two points on the cell state. The yellow rectangles each represent a neural network layer. [34]

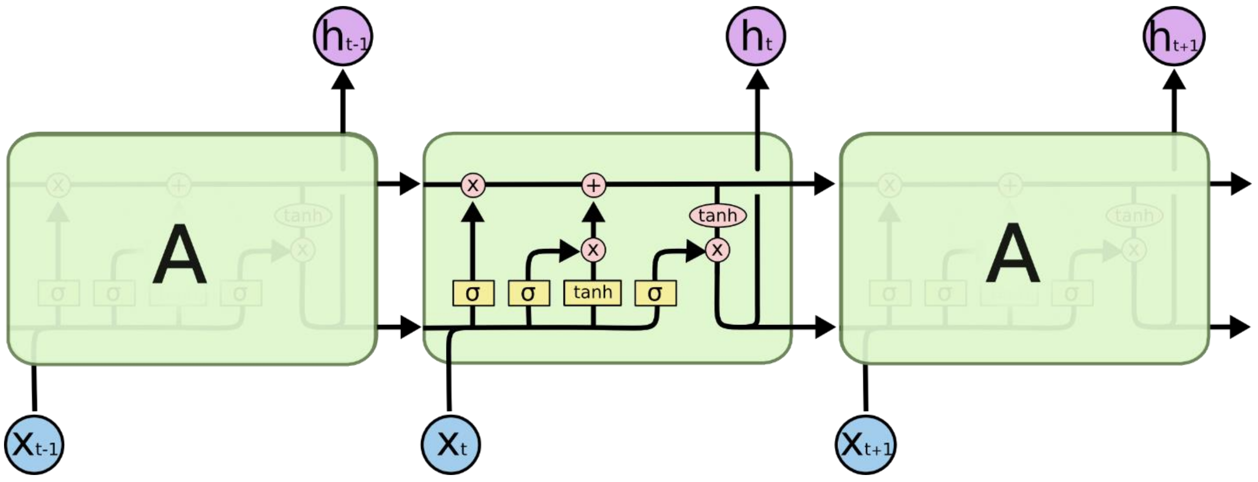


Figure 8: Inside an LSTM

The points on the cell state are key to the LSTM’s success, as they are what control what gets added or removed from the signal. There are three instances in the model of a sigmoid layer (the NN layers labeled σ) followed by a pointwise multiplication ‘x’. These can be thought of as gates that control what to let through. [34]

Recall that a sigmoid layer refers to a layer of NN nodes which use the sigmoid activation function. This means that the part of each gate that is the sigmoid layer will output a set of values between 0 and 1. The effect of this is that when pointwise multiplied with another vector, each output of the sigmoid layer is essentially deciding how much of each feature of that vector to keep, so to speak.

The first gate on the left is controlling what gets ‘forgotten’ from the cell state. It takes into account the input at this timestep and previous cell state (passed through a filter). Since it is a pointwise multiplication, each 1 the sigmoid layer produces ‘retains’ that part of the cell state, and each 0 ‘forgets’ that part of the cell state. Since a sigmoid activation is used, there can be values within this range, meaning that most parts of the cell state will not be entirely ‘kept’ or ‘forgotten. The other gates operate similarly.

The next gate is the one that controls what gets added to the cell state. It is right next to the only NN layer in the model that is not a sigmoid layer. The ‘tanh’ layer generates the values that might potentially be added to the cell state. The tanh activation function can be seen as similar to the sigmoid, in that it generates values between a certain range. In the case of tanh, the range of outputs is (–1, 1) rather than (0, 1).

The second gate, which is located to the left of the tanh layer in the diagram, determines how much each of these potential values will ‘go through’ to the cell state. Both of these parts also use the input and previous cell state at this time step to generate their outputs. The third gate, which also takes these two things as input, decides what parts of the cell state will be the output for the LSTM, as well as be passed to the next iteration. [34]

Mathematically, the output of the first gate looks like this:

$\text{Gate1} = \sigma(W_{g1} * [h_{t-1}, x_t] + b_f)$

Where W_{g1} are the weights of the sigmoid layer, * is a pointwise multiplication, $[h_{t-1}, x_t]$ is the output from the previous timestep concatenated with the input at this timestep, and b_f is a constant modifier. The output of each other gate has an analogous calculation, with the only difference being in the weights of each NN layer and the constant modifiers. The tanh layer follows the same pattern, except the sigmoid function is simply replaced by the tanh function. The output of the LSTM at timestep t, which is also passed as input to the next timestep of the gates' operations, is as follows:

$h_t = o_t * \tanh(C_t)$

Where o_t is the output of the third sigmoid layer, and C_t is the cell state at this timestep, and the * is a pointwise multiplication.

The innovations of LSTM have produced impressive results, clearly improving upon the RNN in terms of long-range performance across time. [35] This model's ability to selectively retain information is highly relevant to the problem of temporal range in RL as a whole, and is very versatile. A DQN with an LSTM should theoretically fare better than the vanilla version, since one of the weaknesses of the DQN is lack of information persistence. With the information covered in this literature review, I can describe my proposed model in detail.

Model Design

Since the idea is to increase temporal range of the agent, and the HDQN model has two main parts, it is technically possible to introduce an LSTM into either. However, in the model I propose, I only use an LSTM inside the meta-controller. The reason is twofold.

First, the goal is to increase performance in delayed reward scenarios, and the HDQN architecture already has a controller setting goals for the lower level controller to use in selecting actions. In other words, the lower level controller is already making use of some temporal abstraction. Considering it operates towards the completion of each goal for some given amount of time before receiving a new goal, attempting to extend this controller's temporal range would be redundant. The idea is to increase the range of the model as a whole.

Second, the meta-controller does have use for some kind of temporal abstraction. Of course, it is learning a policy for the selection of options, which are a form of temporal abstraction, but unlike the lower level controller, this policy learning persists the entire time. In other words, there is effectively only one policy being learned. The meta-controller is not being 'reset' after every window of time given to complete each goal. For this reason, it might be beneficial for the meta-controller to have internal information persistence. It would also conceptually align with the notion of long-term strategy, as now the option selection will be made with consideration of information across a wider range of time.

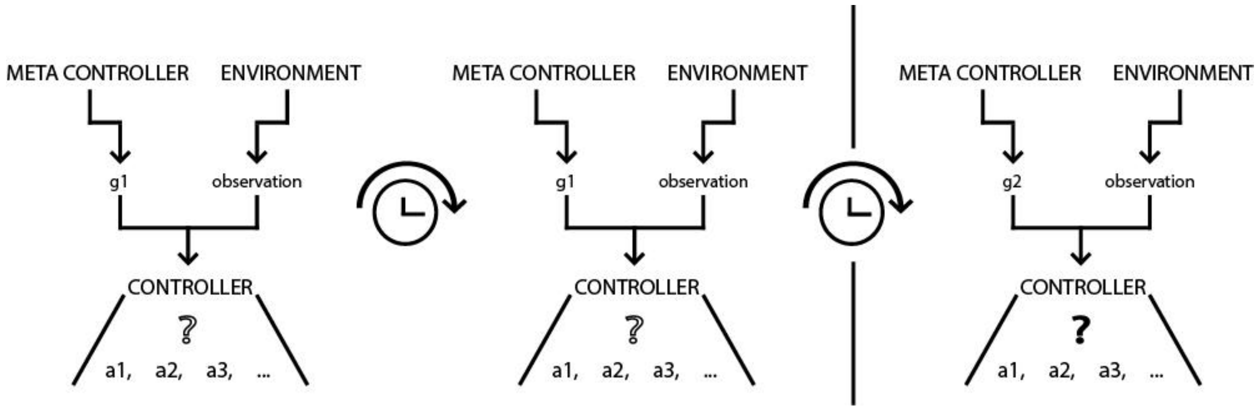


Figure 9: Controller Continuity

Figure 9 above illustrates, in a more abstract way, the break in continuity for the operation of the lower level controller. It receives a goal from the meta controller, an observation from the environment, and selects an action based on those things. Between some time steps, such as the first one in the diagram (symbolized by the clock with an arrow), the goal input remains the same, meaning that the action selection will be geared towards the same end (question marks are the same color). Between time steps when the meta controller selects a new goal, such as the second one in Figure 9, the low level controller will essentially face a break in continuity. Its action selection is now is according to a new goal (question mark is a different color). Since the controller's operation is being 'interrupted' in this way, information persistence might be less fruitful in its context.

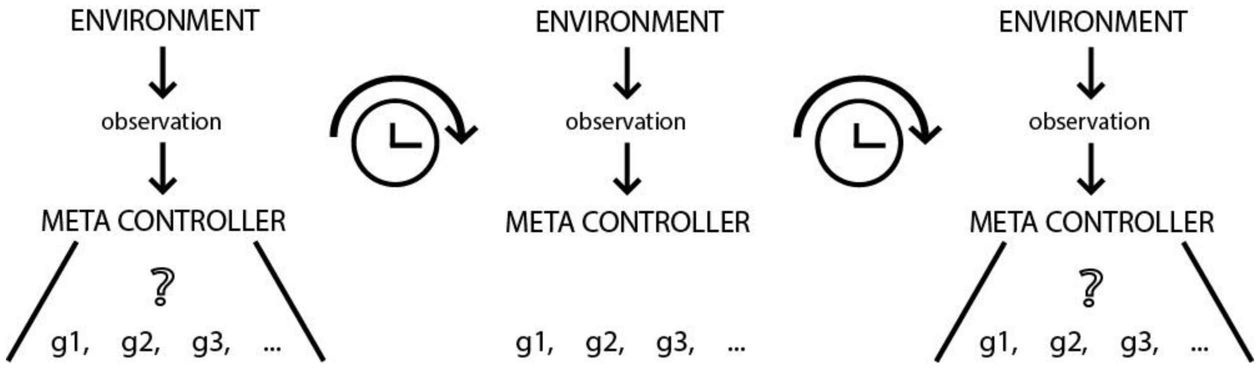


Figure 10: Meta Controller Continuity

Figure 10 above illustrates how the operation of the meta controller is different in a similarly abstract way. On some timesteps, such as the first one illustrated, the meta controller is not making any new selections. On others, such as the second one, the meta controller chooses a new goal to feed to the low level controller as part of its input. There is no break of continuity in the same way that there is for the low level controller. The meta controller is learning a policy to select goals continuously throughout its operation. For this reason, information persistence would appear to have more relevance to this higher level controller than the other.

Figure 11 is a 3-dimensional visualization of the architecture I propose for the DQN in use by the meta-controller. The input is a stack of 84 x 84 images. There are three convolutional layers, an LSTM, and a fully connected layer.

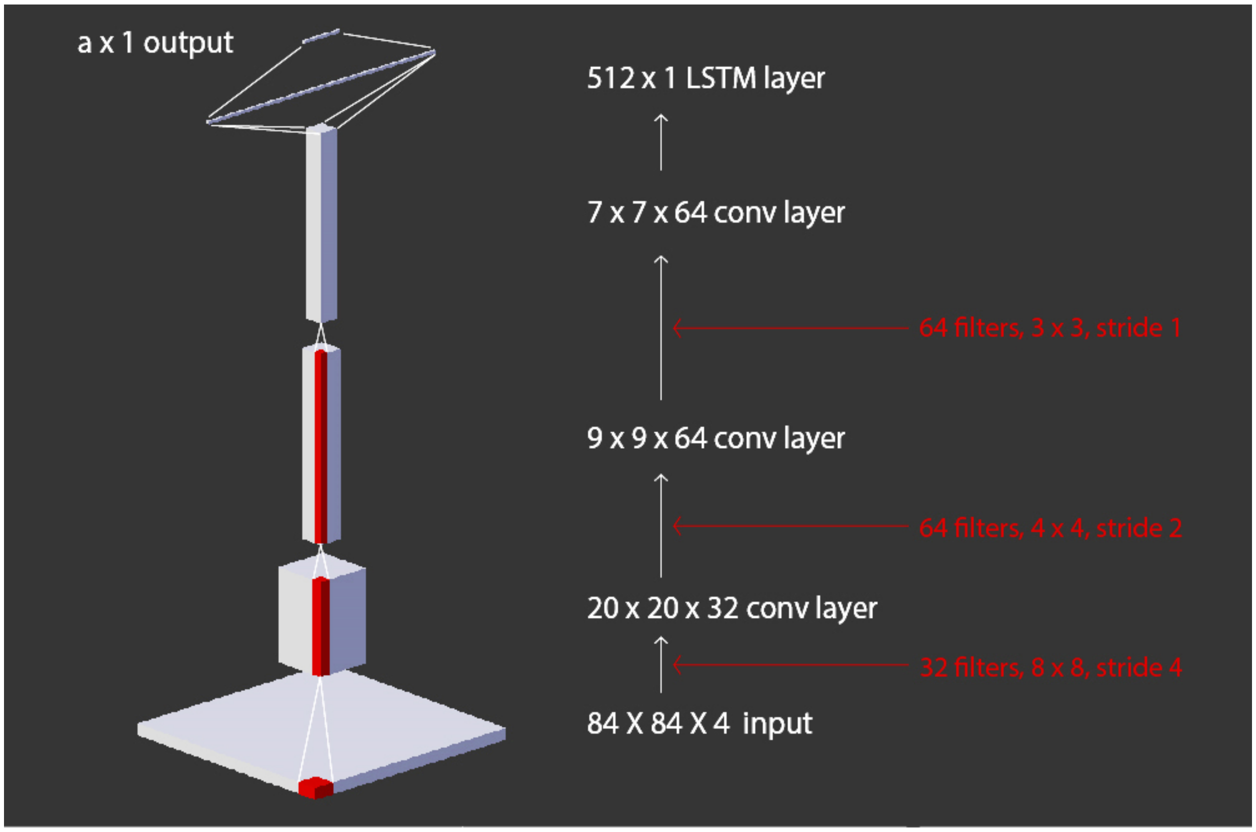


Figure 11: Proposed DQN Architecture

The first convolutional layer is generated by convolving 32 filters of size 8 x 8 with a stride of 4 across the input. This means that there is are 32 sets of 8 x 8 values which are multiplied with every possible set of adjacent 8 x 8 columns in the input. The filtering is visualized in red above, showing the relative size of the filter used for each volume.

Each value generated by convolving a filter across a volume is a single point in the next volume. Each filter is essentially generating a single 'sheet' of values in the next layer; 32 filters results in a volume with a depth of 32. After each convolution, ReLU is used as a processing step for these values. Then the next layer's filters are convolved through this volume, generating a new volume.

After the third convolution, the last volume generated is fully connected to the LSTM layer succeeding it. Each value of the volume can be thought of as the output of a node in the previous layer of the NN. A volume is like a 3-dimensionally organized set of nodes. The convolutions make use of the spatial arrangement, but in this fully connected step, the 3-dimensional structure is no longer used. In the LSTM, each NN layer is fully connected to this set of outputs from the last convolution. The LSTM also uses ReLU.

After the LSTM, there is a final fully connected layer with size **a** x 1, with **a** being the number of actions available in a given environment. This last layer uses a linear activation function rather then ReLU, and each of its outputs is indicative of how much each particular action is preferable to be taken a given time step. The maximum of these outputs is used to determine the action taken, or in the case of the meta-controller, the goal selected. In the context of the meta-controller, **a** actually refers to the number of available goals to choose from.

This is the model for the altered DQN I propose to use for the meta-controller in HDQN.

Evaluation

To test the success of my proposed model, the problem of a videogame which is known to require some long-term strategy for optimization is used. The results are compared to those of a regular DQN in the same environment.

Atari Simulation

In this experiment I trained an altered DQN on the Atari game called Breakout. I used the Open AI Gym interface for the Arcade Learning Environment to simulate the game. This tool was developed as a way to expedite RL research by providing a way to have agents interact with Atari games in the context of actions, observations, and rewards. Gym makes the Arcade Learning Environment accessible in Python (along with other kinds of environments such as physics simulations).

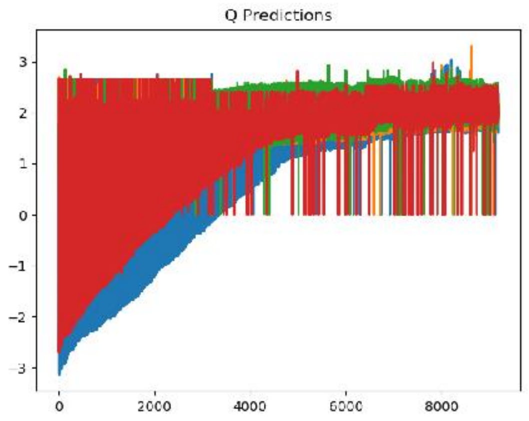
Breakout allows the player to move a small platform on the bottom of the screen either left or right. There is a set of blocks arranged at the top of the screen, and there is a ball that can bounce between the blocks and the player’s platform. The way points are earned is to use the platform to cause the ball to hit blocks. If the ball bounces past the bottom of the screen rather than hitting the player’s platform, it is lost, and a new ball must be used. The player has a limited number of balls to use before the game is over.

This game has room for long term strategy to pay off, making it an interesting experiment for my model. Human players have found that the optimal strategy is to break all the way past the layers as soon as possible, allowing the ball to then bounce above the blocks, taking out many with no risk of being lost. Breaking through the layer in this way is a deliberate strategy that would take some time to pay off in the context of the game, especially in comparison to simply playing ‘normally’, which would result in essentially having to bounce the ball more frequently.

The agent received a stack of observation frames generated by Gym from the game. Each observation from Gym is given in the form of an RGB image of the screen at that time. If taken directly, these images come in the form of 210 x 160 x 3 array of values. As an input for NN training, this is undesirably large. Using some preprocessing (described in the Appendix), these observations from the game are reduced to an 84 x 84 x 1 image. Other details about training and model parameters are provided in the Appendix.

The model’s training statistics are visualized below. Data for the graphs was not collected during the initial period of each experiment, during which the agent was allowed to perform random actions without learning. This is done for the purpose of populating the replay memory with experiences before the learning process begins.

The Figure below visualizes Q predictions along with respect to each update to the network. The multicolor graph shows the predicted value of each action (there are 4 in Breakout) at the time of each update. The single-color graph shows the maximum of these values at each update. The trend is to predict increasingly high values for all of the actions at each time step, and for an increasingly high maximum possible value for each state. The problem that was



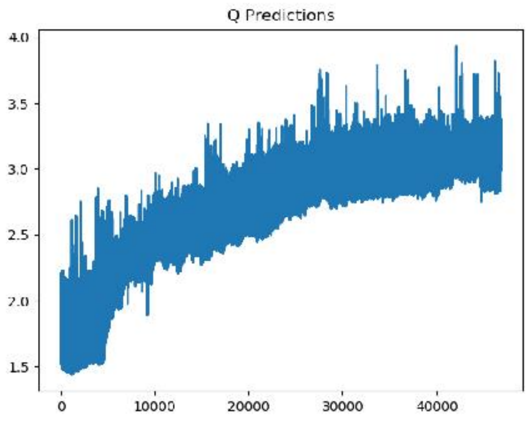


Figure 12: Q Predictions Plateauing

encountered by this agent can be seen in part through these graphs as well: even in relatively early stages of training, the predictions begin to plateau.

When observing the agent's actual performance in the game during training, these figures make more sense. The behavior of the agent is to learn to consistently hit a few blocks during each episode, which corresponds to the value predictions plateauing around 3 to 4.

Take a look at the screenshots from a player's episode during training. This is the behavior it appears to settle on, not finding the higher potential scores. Perhaps the reward structure of this environment has this effect on training. Conceptually, the reward is only earned when the ball is hitting a block. Whatever the observations look like at that point will be what is associated with the reward. However, the base DQN agent has been able to learn to outperform human players in this environment.

These observations bear some of the signs of unstable training. In response, I performed smaller experiments to understand the nature of the instability: ranges of hyperparameters, different weight initialization techniques, and different games.

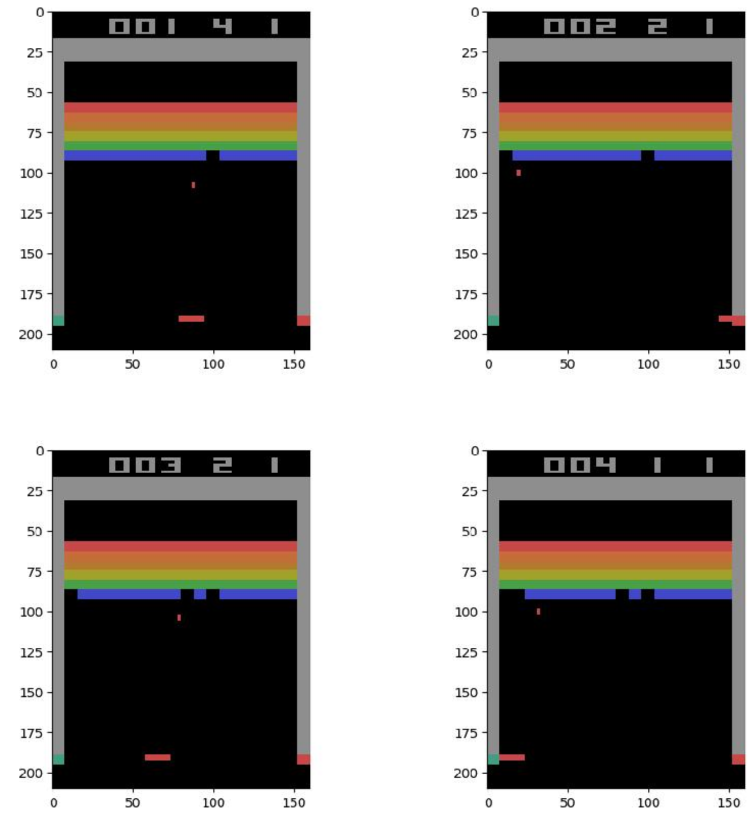


Figure 13: Breakout Performance

First, the hyperparameter experiments. I most broadly tested the learning rate, since the issue appears to be an undesirable behavior of the loss over time. The agent seems to settle at the same local minimum of the loss function every run, regardless of learning rates. The loss calculated during each optimization step drops very quickly at the beginning of training and then does not change outside of some small variation for the remaining time. The graphs of loss below show this behavior.

Each of these graphs visualizes the loss with different learning rates and over different intervals. The one on the left shows the loss with the lowest learning rate used (0.0000001) within a relatively early range of updates. The one on the right shows the loss with the highest learning rate used (0.00025) over a longer training period. Regardless of learning rate, the loss converges to the same place.

Perhaps part of the problem is that the agent is learning to associate most observations with no reward, as the reward is limited to only a sparse few frames – the ones in which the ball

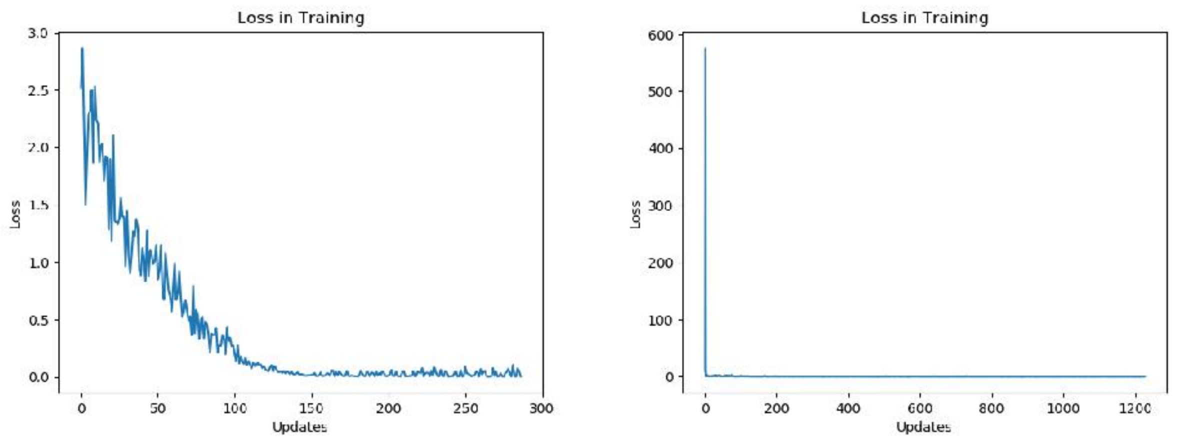


Figure 14: Losses Converging

hits a block. However, the graphs of Q predictions above would conflict with this, as the agent is predicting increasing potential values for each state it finds itself in.

In the next set of experiments, different initialization strategies were used for the model network’s weights. Random uniform initializations as well as random normal initializations were used for the weights. The resulting graphs for loss exhibited the same behavior as those from the learning rate experiments.

The final set of experiments was to try the training on a different Atari game, Space Invaders. A few sets of hyperparameters from the previous experiments as well as uniform random initialization were used. The loss visualizations resulted similarly as well.

Overall, my model’s performance showed signs of trouble with the training process. It was unable to outperform the original model it is based on.

Implementation Issues

The original intent was to compare the performance of my model on this game to that of the original as per the publication's evaluation. For reference, the results of the original were presented in a couple of ways, including a percentage relative to human performance. The values for human performance were calculated using various human players over many episodes of each game. With respect to Breakout, the original model's performance was over 1000 % (more than 10 times as good as the human performance). Below is a table for a comparison of performance in terms of average score.

Table 2: Score Comparison

Model	**Score**
DQN	401.2
My Model	3.4
Random Baseline	1.7

The interesting point is that the model learns something at least slightly better than purely random action (Random Baseline in the table), but nowhere near the original. To verify whether this had something to do with my implementation of the base model, I removed my proposed alterations from the model and performed some experiments. The behavior of the loss function and the average scores are similar to those from my altered model, suggesting a flaw within my implementation of the original paper. Specifically, based on an informal code review, the implementation of the updating procedure is in some way incorrect, given that the original publication bears no mistakes in its algorithm description.

This means that the value of the model proposed here cannot be truly evaluated here.

Conclusion

Overall the effectiveness of recurrence within the DQN is inconclusive in the experiments of this work. Due to the behavior of loss and predictions, it is apparent that training the model with the added complication of recurrence requires an alteration of the whole process. It is notable that in the original experiments carried out on DQN, there were many small modifications that had to be made to reduce instability in training, such as the reward clipping and the exploration policy decay. Clearly the internal structure of the LSTM must be taken into account in designing particular training procedures for any model similar to the one proposed here. It is still reasonable to predict that the LSTM would introduce a kind of persistence different from what the convolutions already provide, especially in the context of the findings of the original DQN publication. [2]

Further work could look into several areas. There could be more experimentation as to the specifics of the model's layout. Can the LSTM be placed after a fully connected layer rather than in place of one? How does the size of the LSTM affect learning? Another avenue of research is that of generating goals automatically to be selected from by the meta-controller. This is an open problem, as the nature of goals is still conceptually open for definition. It might be possible, for example, to define goals as based on objects in a visual input, and then the problem of automating the goal space generation turns into one of computer vision. It is also interesting to consider using *convolutional* LSTM's in place of one or more of the convolutional layers. Perhaps the more low level information is more valuable to keep in circulation.

Appendix

Here you can find some of the specific details about how the experiments were set up and what was used to implement my model.

Image Preprocessing

The image preprocessing necessary to make the training feasible is done in a few steps. First, grayscale values are generated from the RGB values. Then, the 210 x 160 images are scaled to 84 x 84. These things are accomplished using Open CV, an open source library for different computer vision tools. The scaling is actually from 210 x 160 to 84 x 110. The top 26 rows of pixels are removed to get a square image 84 x 84. The idea is that the top row of pixels in Atari games generally only display the score, so they are not necessary for the observations we want to train on (the game itself).

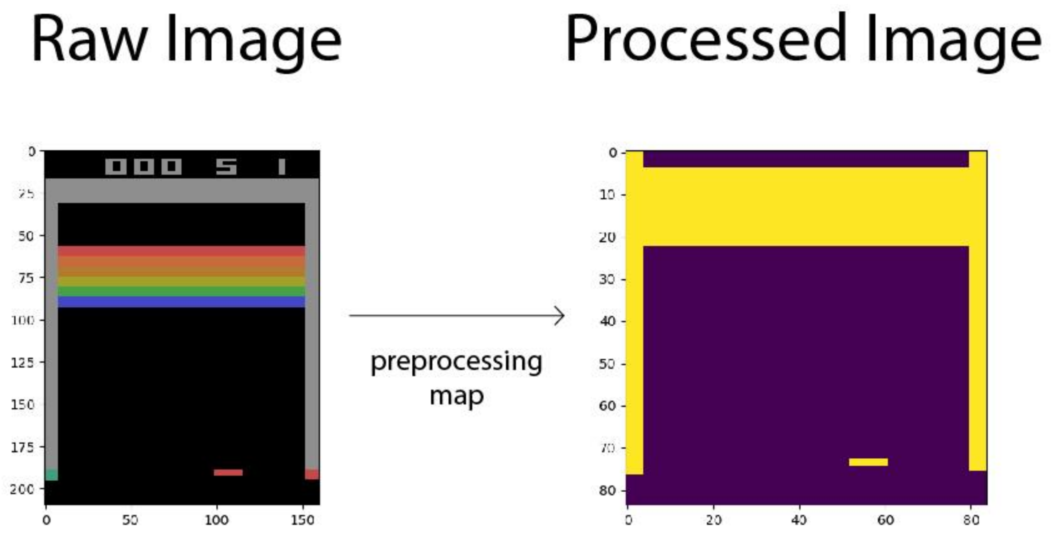


Figure 15: Image Processing Visualization

The Figure above visualizes the results of preprocessing on the image inputs to the model. In context of training, the input to the model is actually a stack of such preprocessed images. The previous 4 observations are used.

Programming

The following table gives short descriptions for the tools used to implement the model and run the experiments.

Table 3: Programming Tools

Tool	**Description**
Python	Programming language
Tensor Flow	Python framework that abstracts the implementation details of neural networks, providing functions to build and interact with them
Keras	Interface for Tensor Flow that is useful for generating models
OpenCV	Open source library used for the image preprocessing steps (scaling, extracting grayscale from RGB)
Open AI Gym	Python interface for the Arcade Learning Environment, used to simulate Atari games and receive observations and rewards
Matplotlib	Python library used for generating visualizations of data

Keras's Squential model functionality is used to create the model as visualized in the Model Design section. The RMSprop algorithm implemented within Keras is used for model optimization, with a batch size of 32.

The input to the model was implemented as a queue of size 4, with a frame of observation being pushed into the queue after every 4th action. This is because the agent as implemented repeats an action 4 times into the environment. Effectively, it only receives feedback every 4th frame. This is done for efficiency in training.

Rewards during training are clipped between -1 and 1. This allows for the learning rate to be more versatile and makes certain computations less expensive. The drawback to this is that conceptually the model is not learning differently based on the magnitude of each reward.

Parameters

As alluded to above, this implementation involves several parameters which affect training. They were not optimized in an automated way for these experiments, as that would require significant computational resources and my experiments here are limited in scope. The table below summarizes the settings used for each parameter and what that parameter does.

Table 4: Parameter Settings

Parameter	**Setting**	**Description**
Gamma	0.99	The discount factor used in calculating updates
Epsilon	1.00	Starting value for the e-greedy exploration policy
Epsilon Minimum	0.01	Lowest value epsilon can reach
Epsilon Decay	10000	How many frames epsilon will be reduced for
Learning Rate	0.0025	The learning rate used by RMSProp

Batch Size	32	Number of transitions sampled from memory to perform updates
Action Repeat	4	How many times each action will be repeated, or effectively how many frames are skipped
Final Exploration Frame	40000	How many frames will pass before epsilon begins to get reduced (how long to act random)
Target Update Frequency	10000	How many frames pass between resetting of the target value function
Max Frames	500000	How many frames are used in training

As further explanation for the Final Exploration Frame parameter: in training the model is allowed to act purely randomly for some time in order to accumulate a mass of transitions in memory which can be used for learning. This parameter sets how long this will go on.

The way epsilon is reduced is that after every update, a constant amount is subtracted from the current epsilon. This goes on until it reaches its lower limit specified by Epsilon Minimum. The amount subtracted is the result of dividing the difference between starting epsilon and Epsilon Minimum by the Epsilon Decay parameter.

The action repeat works by causing the agent to submit the same action to the environment that many times. After the nth time, an observation is received from the environment and updates performed as usual.

BIBLIOGRAPHY

[1] J. Koutnik, J. Shmidhuber and F. Gomez, "Evolving deep unsupervised convolutional networks," *Proceedings of the 2014 conference on Genetic and evolutionary computation,* p. 541–548, 2014.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, M. G. Bellemare, A. A. Rusu, A. Graves and et al., "Human-level control through deep reinforcement learning," *Nature,* p. 529–533, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot and et al., "Mastering the game of go with deep neural networks and tree search," *Nature,* pp. 484-489, 2016.

[4] M. G. Bellemare, J. Veness and M. Bowling, "Investigating Contingency Awareness Using Atari 2600 Games," *AAAI,* 2012.

[5] A. Juliani, "Simple Reinforcement Learning with Tensorflow Part 7: Action-Selection Strategies for Exploration," 14 November 2016. [Online]. Available: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf.

[6] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband and Z. Wen, " A tutorial on Thompson sampling," *Foundations and Trends in Machine Learning,* pp. 1-96, 2018.

[7] B. C. Stadie, S. Levine and P. Abbeel, "Incentivizing exploration in reinforcement learning with deep predictive models," ArXiv, 2015.

[8] T. D. Kulkarni, K. Narasimhan, A. Saeedi and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," *Advances in neural information processing systems,* pp. 3675-3683, 2016.

[9] "A Beginner's Guide to Neural Networks and Deep Learning," Skymind, [Online]. Available: https://skymind.ai/wiki/neural-network.

[10] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, pp. 85-117, 2015.

[11] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," *IEEE,* p. 2278–2324, 1998.

[12] R. Sutton and A. Barto, "Introduction to reinforcement learning," *MIT Press,* vol. 135, 1998.

[13] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White and D. Precup, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," *The 10th International Conference on Autonomous Agents and Multiagent Systems*, vol. II, p. 761–768, 2011.

[14] R. S. Sutton, D. Precup and S. Singh, " Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence,* p. 181–211, 1999.

[15] J. Tsitsiklis and B. V. Roy, "An analysis of temporal-difference learning with function approximation," *IEEE,* 1997.

[16] J. L. McClelland, B. L. McNaughton and R. C. O'Reilly, "Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory," *Psychol. Rev.,* p. 419–457, 1995.

[17] J. O'Neill, B. Pleydell-Bouverie, D. Dupret and J. Csicsvari, "Play it again: reactivation of waking experience and memory," *Trends Neurosci,* p. 220–229, 2010.

[18] D. Liu, "A Practical Guide to ReLU," 19 11 2017. [Online]. Available: https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7.

[19] M. M. Botvinick, Y. Niv and A. C. Barto, "Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective," *Cognition,* pp. 262-280, 2009.

[20] C. Szepesvari, R. S. Sutton, J. Modayil, S. Bhatnagar and et al, "Universal option models," *Advances in Neural Information Processing Systems,* pp. 990-998, 2014.

[21] J. Sorg and S. Singh, "Linear options," *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, pp. 31-38, 2010.

[22] M. C. Machado and M. Bowling, " Learning purposeful behaviour in the absence of rewards," *arXiv*, 2016.

[23] E. S. Spelke and K. D. Kinzler, "Core knowledge," *Developmental science,* pp. 89-96, 2007.

[24] K. S. Lashley, The problem of serial order in behavior, Bobbs Merrill, 1951.

[25] J. D. Cohen, K. Dunbar and J. L. McClelland, "On the control of automatic processes: A parallel distributed processing account of the Stroop effect," *Psychological Review,* pp. 332-361, 1990.

[26] R. Cooper and T. Shallice, "Contention scheduling and the control of routine activities," *Cognitive Neuropsychology,* pp. 297-338, 2000.

[27] J. Bruner, "Organization of early skilled action," *Child Development*, pp. 1-11, 1973.

[28] P. M. Greenfield, "Building a tree structure: The development of hierarchical complexity and interrupted strategies in children's construction activity," *Developmental Psychology,* pp. 299-313, 1977.

[29] J. F. Lehman, J. Laird and P. Rosenbloom, "A gentle introduction to soar, an architecture for human cognition," *Invitation to cognitive science*, pp. 212-249, 1996.

[30] J. R. Anderson, "An integrated theory of mind," *Psychological Review,* pp. 1036-1060, 2004.

[31] N. D. Daw, Y. Niv and P. Dayan, "Actions, policies, values and the basal ganglia," *Recent breakthroughs in basal ganglia research,* pp. 1214-1221, 2006.

[32] J. O'Doherty, P. Dayan, P. Shultz, J. Deischmann, K. Friston and R. J. Dolan, "Dissociable roles of ventral and dorsal striatum in instrumental conditioning," *Science,* p. 452–454, 2004.

[33] A. G. Barto, "Adaptive critics and the basal ganglia," *Models of information processing in the basal ganglia,* pp. 215-232, 1995.

[34] C. Olah, "Understanding LSTM Networks -- colah's blog," 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[35] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation,* pp. 1735-1780, 1997.

ACADEMIC VITA

Andres De La Fuente

EDUCATION

The Pennsylvania State University — **University Park, PA**

B.S. in Computer Science + B.A. in Philosophy — *2015 –2019*

- Schreyer Honors College

WORK AND LEADERSHIP EXPERIENCE

Air Products *(IT Intern, Applications Team)* — **Allentown, PA**

Summer 2017

- Developed a proof of concept cross platform mobile app to perform object recognition, character recognition, and barcode scanning to manage inventory
- Independently communicated with people from across the company to manage the project

PSU E-Design *(Student Researcher, Developer)* — **University Park, PA**

Fall 2018

- Led a group of diverse academic backgrounds in development of a 3D point cloud de-noising solution, implementing and enhancing methods in recent academic literature from scratch
- Performed a literature review based on a variety of mathematical fields

SELECTED ACADEMIC PROJECTS

Honors Thesis *(Machine Learning, with Dr. Vasant Honavar)* — *Spring 2018 – Spring 2019*

- Researched a wide range of literature, with a focus on Reinforcement Learning
- Designed and developed a novel approach to the problem of temporal abstraction using a hierarchical grouping of deep convolutional neural network models
- Built this model from cutting edge research in deep reinforcement learning and models from other domains

SELECTED INDEPENDENT PROJECTS

Maestro — *Fall 2018 – present*

- 2D game in development, technical achievements so far include: 2D custom gesture recognition, custom development tools, 2D physics
- Directed rehearsals and recording sessions with voice actors and musicians

Zillow Prize Competition — *Spring 2018*

- Performed exploratory data analysis on Zillow's competition dataset on Kaggle.com
- Put together a successful machine learning ensemble based on the above
- Testing involved a few types of models: support vector machines, random forests, and boosting

SKILLS & INTERESTS

Languages: Fluent in Spanish and English, Intermediate in German

Programming Languages: Python, C, C++, C#, Java, JavaScript, HTML, CSS
Development: Git, Unity GDB, Bash, Azure, Xamarin Android / iOS
Machine Learning / Data Science: TensorFlow, SciKit Learn, SciPy, NumPy, MatPlot Lib, OpenAI Gym
Organizations: The GLOBE *(discussions with professors about international topics through different academic lenses),* Performing Magicians *(performer and mentor for four years)*