Differentiable Neural Planners with Temporally Extended Actions

Valliappa Chockalingam
Supervised By: Satinder Singh Baveja
Computer Science & Engineering, University of Michigan

Abstract

Deep Reinforcement Learning has brought about various advances in game or puzzle-like benchmarks that other traditional AI agents long had difficulty with. However, the policies learned by the Deep Neural Networks have usually been reactive policies using primitive actions of the environment at hand. While certain components that aid in generalization have been in recent use, like external memory, other have largely been missing like explicit long-term planning computations. Additionally, a general (robust and "well-generalizing") artificial intelligence is likely to need temporally extended actions with a planner that can decide when, which and for how long to execute them. In this paper, Value Iteration Networks, deep neural networks with explicit planning components, are replicated and analyzed. Then, hierarchies of planning computations are considered through temporally extended options and simple experiments to demonstrate the viability of such Differentiable Neural Planners with Temporally Extended Actions are described with interpretable results. Finally, some possible directions for future work are noted.

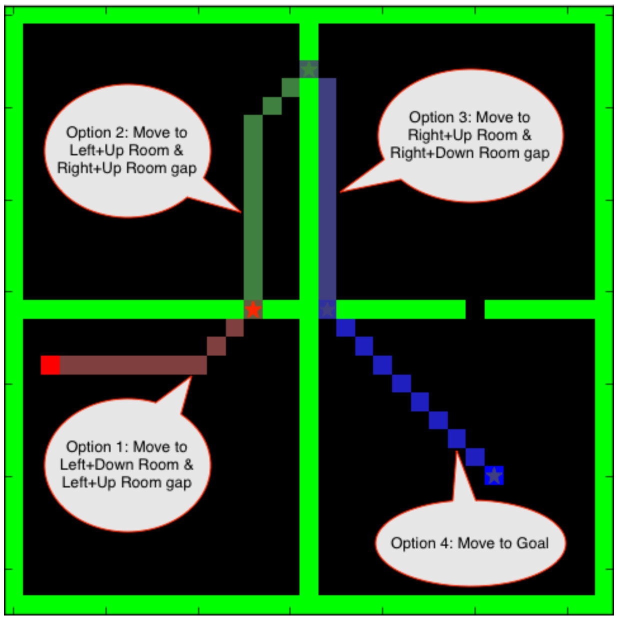
Option 2: Move to Left+Up Room & Right+Up Room gap
Option 3: Move to Right+Up Room & Right+Down Room gap
Option 1: Move to Left+Down Room & Left+Up Room gap
Option 4: Move to Goal

Figure 1: A Learned Path With Temporally Extended Actions

1 Introduction and Related Work

Deep Reinforcement Learning (Deep RL or DRL), an area in which researchers have become increasingly interested in the past few years, has brought about various successes in challenging puzzle or game-related benchmarks [12, 13, 16] and related "real-world" problems as well [6, 22]; it now allows us to solve various problems that would otherwise pose obstacles for traditional AI agents. In particular, Deep RL allows us to create agents that perform well in domains with the "extremes" of the constituent parts of a general RL problem [18]: high-dimensional inputs [12, 13], sparse rewards requiring deep exploration [2, 17], and with some specific architectures and algorithms, large (possibly continuous) action spaces [4, 5, 11], and, partial observability [8, 13].

However, while it is clear from the aforementioned

Deep RL work that we have developed agents that can act in complex environments so as to maximize reward successfully, it is still not clear how much these agents understand the learned policies and can generalize. Moreover, most DRL work has generally led to reactive policies that depend only on some small portion of the recent states seen and possibly actions taken. For a general artificial intelligence, the ability to plan (and construct long term plans in particular) is most likely quite crucial and is something largely unexplored in the field.

Recently “Value Iteration Networks” (henceforth abbreviated VIN at times) were introduced in [20], dealing with this problem, particularly in the context of learning to plan and generalizing this process across unseen environments. The main contribution of the paper is quite unquestionably in its introduction of an architecture that explicitly includes a differentiable planning computation. More precisely, the authors show that using Convolutional Neural Networks [9] in a certain way can be viewed as Value Iteration [18].

Considering multiple such “differentiable neural planners” or having some other combination of shallow and deep looking planners, we naturally arrive at temporal abstractions and move towards “long-term” planning. The abstraction of temporally extended actions, also known as options [19], allows for more robust agents due to the more coarse-grained nature of the actors and planners at higher levels of a hierarchy which don't have to worry about working with low-level primitive actions or even, possibly, the high dimensional “raw” or somewhat preprocessed states. Meanwhile, the planning and acting at the primitive level can be handled in the lower levels of the planning and acting hierarchy. Moreover, using certain ”non-confounding” implementations for the planners at different hierarchies, like Value Iteration Networks, allows for some nice and easily interpretable results (as shown in **??**). Here, ”non-confounding” loosely refers to the fact that figuring out what each neuron output means or refers to, in the RL setting, is simple. Thus, the motivation for having such “Differentiable Neural Planners with Temporally Extended Actions” becomes clear (The argument here is essentially along the lines of [14]; division of labour generally leads to productivity and economic growth.)

In the following sections, VINs are first introduced along with necessary preliminaries, following [20]. Then, the results of [20] are replicated on a Grid World domain. Next, some insightful analysis, in particular, a look at the inferred reward and value functions, and the plans is done; these analyses are not in [20]. Post that, steps are taken towards incorporating temporally extended actions into VINs that result in more robust agents that can construct useful high-level plans while abstracting away the unnecessary fine-grained primitive-action based control; namely, these steps include weakening the assumption that the transition probabilities take into account locality of actions and adding subcontrollers that execute temporally-extended actions comprising of the primitive actions while the ”main” or high level controller works with the temporally extended actions. Finally, some possible future steps are put forth.

2 Methods

2.1 Preliminaries

Following the standard model for a sequential decision process, first a Markov Decision Process (MDP) M is defined by a tuple $M = (S, \mathcal{A}, R, P, \gamma)$. Here, S is the set of states in the MDP, $\mathcal{A}$ is the set of actions available to an agent, $R(s, a, s')$ is the reward function which specifies the immediate reward an agent obtains by taking action $a \in \mathcal{A}$ at state $s \in S$ and reaching state $s' \in S$, $P(s'|s, a)$ is the transition probability function which specifies the probability of reaching state s' by taking action a at state s, and, finally, $\gamma \in (0, 1)$ is a discount factor [3].

A policy $\pi(a|s)$, which makes use of the Markov assumption [3], specifies an action distribution for each state, and, the goal in an MDP is to find policies that obtain the highest (discounted-)reward in the long term.

To find such a policy, the notion of the *value* of a particular state, s, under some policy, π, is useful. Formally, the value of a state s is the expected discounted sum of rewards starting from the state in question and following policy π:

$$V^{\pi}(s) = \mathbb{E}^{\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) | s_0 = s] \quad (1)$$

Here, the expectation is over trajectories of states and actions, $(s_0, a_0, s_1, a_1, ...)$, generated by following policy π.

Similarly, we can define a state-action value function, the expected discounted sum of rewards starting from state s, executing action a, and thereafter following policy π:

$$Q^{\pi}(s) = \mathbb{E}^{\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a] \quad (2)$$

The optimal value function, $V^*(s)$, is given by $V^*(s) = \max_{\pi} V^{\pi}(s)$, i.e., a value function assigning the maximal discounted reward obtainable from each state to each corresponding state-value. A policy π^* is said to be optimal if $V^{\pi^*}(s) = V^*(s) \;\; \forall s$.

To calculate both V^* and π^*, Value Iteration (VI) [18, 3], a popular Dynamic Programming algorithm, can be used. After initializing V_0 arbitrarily and choosing some policy π_0, we compute a state-action value function induced by the policy and current state-value function. Then, we update our state-value function by taking the max over actions of the state-action value function for each state.

$$V_{i+1}(s) = \max_a Q_i(s,a) \;\; \forall s \text{ where}$$
$$Q_i(s,a) = \sum_{s'} R(s,a,s') + \gamma P(s'|s,a) V_i(s')$$

Finally, we can calculate an optimal policy by taking the greedy action at each state, i.e., the action corresponding to the max state-action value for each state.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [9] are neural networks that have proved to be very useful in computer vision and also as part of policy or value function approximators in Deep RL, one of the first examples of which was [12].

A convolutional layer generally takes in a 4-dimensional input where one dimension corresponds to the batch size, b, another two for the number of vertical pixels, m, and number of horizontal pixels, n, and finally the remaining dimension for the number of channels, c.

The output of a c'-channel convolutional layer with kernels $W^1, ..., W^{c'}$ is given by $o_{i',j',d'} = \sigma\left(\sum_{i,j,d} W^{d'}_{i,j,d} X_{i'-i,j'-j,d}\right)$, where σ is a scalar activation function, usually a Rectified Linear Unit (ReLU) given by $f(x) = \max(0, x)$ with derivative $f'(x) = \mathbb{1}_{x \geq 0}$ where $\mathbb{1}_p$ is the indicator function giving 1 when predicate p is true and 0 if false.

Another type of convolutional layer, which allows for exponential increases in receptive fields (the size of the patches associated with each neuron) without any increase in the number of parameters, is briefly in the experiments. A convolution with holes, also known as a Dilated Convolution or a convolution trous, is useful in scenarios where we would like to keep the number of parameters low, but enlarge the receptive field, as described before. To achieve this, the main idea is to introduce a "rate" parameter, say r. When $r > 1$, this convolution samples the input values every r pixels in the height and width dimensions. This is equivalent to convolving the input with a set of upsampled filters, produced by inserting $r - 1$ zeros between two consecutive values of the filters along the height and width dimensions [1]. [21] provides a more detailed description of such convolutional layers with figures.

2.3 Value Iteration Module

The VI Module is the planning component of Value Iteration Networks that implements Value Iteration (illustrated in Figure 2). What follows is a brief explanation of this module and the insight behind it. The following section titled "Value Iteration Network Model" will go into more detail and describe how the VI module is used in Value Iteration Networks.

The useful insight is that each iteration of VI may

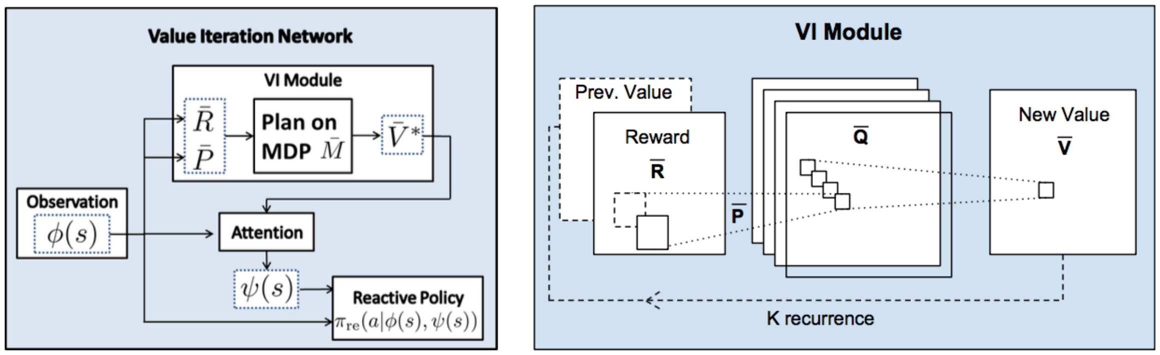

Figure 2: Value Iteration Module and Network, from [20]

be seen as passing a reward function image R and previous value function image V_k through a convolutional layer followed by a max-pooling layer. Passing V_k and R through a convolutional layer where the kernels correspond to discounted transition probabilities, we can get a volume where the channels correspond to the Q-function for each action. Max-pooling depth-wise and then taking that as the new value function results in an iteration of VI. Now, taking this new Value Function image with the reward function image, we can then perform the next iteration of VI. By repeating this process K times, we effectively get K iterations of VI.

2.4 Value Iteration Network Model

Following the MDP formulation, let M denote the MDP of the domain we design policies in. Assume there is some unknown MDP $\bar{M}$ such that the optimal plan in M contains some useful information in about the optimal plan in $\bar{M}$. This idea is to equip policies with the ability to learn and solve $\bar{M}$, and to add the solution of $\overline{M}$ as an element in the policy π. This leads to a policy that automatically learns a useful $\bar{M}$ to plan on.

Denote $\bar{s} \in \bar{S}, \bar{a} \in \bar{A}, \overline{R}(\bar{s}, \bar{a}, \bar{s'})$ and $\bar{P}(\bar{s'}|\bar{s}, \bar{a})$ as the states, action, rewards and transition probabilities in $\bar{M}$. To allow for a connection between M and $\bar{M}$, $\bar{R}$ and $\bar{P}$ can depend on observations in M, say $\phi(s)$ giving $\bar{R} = f_R(\phi(s))$ and $\bar{P} = f_P(\phi(s))$.

Once an MDP $\bar{M}$ has been specified, any standard planning algorithm can be used to obtain the value function V^*. Continuing the discussion of the previous section, we describe the VI module in more detail.

The inputs to a VI module is a "reward image" $\bar{R}$ of dimensions m, n, c, where here, for the purpose of clarity, the CNN formulation is used and it is assumed that the state space S maps to a 2-dimensional grid though this assumption can be weakened and, or otherwise, general discrete state spaces can be formulated this way. The reward is then fed into a convolutional layer $\bar{Q}$ with $\bar{A}$ channels and a linear activation function. Each channel in this layer corresponds to $\bar{Q}(\bar{s}, \bar{a})$ for a particular action $\bar{a}$. This layer is then max-pooled along the actions channel to produce the next-iteration value function layer $\bar{V}$ where $\bar{V}_{i,j} = \max_a \bar{Q}((i,j), \bar{a})$ and (i, j) indicates the state corresponding to the row i, column j. The next-iteration value function layer $\bar{V}$ is then stacked with the reward $\bar{R}$ and fed into the convolutional layer and max-pooling layer I times for I iterations of value iteration.

Now, the vector of values $\bar{V}^*(s)\ \ \forall s$ encodes all the information about the optimal plan in $\bar{M}$. Thus, adding the vector $\bar{V}^*$ as features to the policy π is sufficient for extracting information about the optimal plan in $\bar{M}$. However, an additional property of $\bar{V}^*$ is that the optimal decision $\bar{\pi}^*(\bar{s})$ at a state $\bar{s}$ can depend only on a subset of the values in $\bar{V}^*$ because

$$\bar{\pi}^*(\bar{s}) = \arg\max_{\bar{a}} \sum_{\bar{s}'} \bar{R}(\bar{s},\bar{a},\bar{s}') + \gamma \bar{P}(\bar{s}'|\bar{s},\bar{a})\bar{V}^*(\bar{s}').$$

Note further that if the MDP has some local connectivity structure, the states for which $P(\bar{s}'|\bar{s},\bar{a}) > 0$ is a small subset of S.

Thus, as far as neural networks are concerned, we are interested in a form of attention, in the sense that for a given label prediction (action), only a subset of the input features (value function) is relevant. Attention is known to improve learning performance by reducing the effective number of network parameters during learning. Therefore, the second element in VINs is an attention module that outputs a vector of (attention modulated) values $\psi(s)$.

2.5 Domains

2.5.1 Grid World Domain

The experiments begin with a simple traditional Grid World domain, similar to the one in [20], that allows for easy testing of RL agents. In this section, the domains and specificially the state spaces are described. The action spaces and training regimes are described in forthcoming sections.

The state observations are three-channel images, with height and width dimensions 20 x 20, as illustrated in Figure 3. Along one channel, the agent position is specified (the red pixel), the goal is given along another channel (the blue pixel), and finally, the obstacles (which agents cannot pass into or through) are given along the remaining channel (the green pixels.) The number of obstacles is chosen to be between 0 and 50 and it is ensure that there is always a path between the agent starting location and the goal.

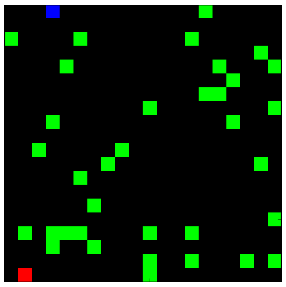

Figure 3: Illustration of State Observation

2.5.2 Four Room Domain

While the Grid World domain allows for some easy analysis and is a good starting step for testing RL agents, given that temporally extended actions are a focus of this work, another domain that yields more naturally to testing agents with options is also experimented with.

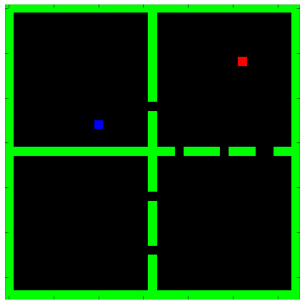

Figure 4: Illustration of State Observation

As shown in Figure 4, the state observations are very similar to that of the Grid World domain (33 x 33, 3 channel images.) More specifically, there are 4 rooms in this domain with different openings or gaps allowing for moving between the rooms. The number of openings or gaps between any two rooms is chosen between 0 and 5 (and it is ensured that there is a path between the starting location and the goal.) The state observations are the same as those in the Grid World Domain (three channels corresponding to the agent position, goal and obstacles respectively).

2.6 Action Spaces and overview of Experiments

The primitive actions in both the Grid World Domain and the Four Room domains are given by movement by 1 space in the unblocked (i.e., obstacle-less) 4 cardinal and 4 ordinal directions around the agent's current location.

The first experiment tests for a working VIN implementation. In the grid-world domain, we can let $\bar{M}$ have the same state and action spaces as the true grid-world M. However, as will be shown in the results, slightly increasing the "internal" number of actions can help in better performance. The reward function f_R ideally maps an image of the domain to a large positive reward at the goal, and a large negative reward near an obstacle, while f_P can simply encode the deterministic movements of the grid-world domain. While these rewards and transitions are not necessarily the true rewards and transitions we might use in defining an MDP for the Grid World, an optimal plan in $\bar{M}$ will still follow a trajectory that avoids obstacles and reaches the goal, similar to the optimal plan in M.

Next, in the Grid World domain, an experiment is done with both the 8 primitive actions (mentioned before) and an additional 16 options given as all permutations of primitive actions with length 2.

Finally, another experiment is perfomed in the Four Room domain with temporally extended actions. In this case, the options are specifically trained to be ones that learn how to move the agent from any given room to the nearest gap, or the goal, as shown in **??** .

2.7 Metacontrollers and Experiments

As described in the previous section, the first experiment with primitive actions is done on the Grid World domain to test for a working implementation of VIN. A few more details about this experiment follow.

For the attention module, we can simply look at the values of the states corresponding to the states that are neighboring the state where the agent is located. However, an even simpler mechanism is possible and is what is used. We can just look at the state-action values for the state corresponding to where the agent is located (and choose actions defined by this distribution.)

The architecture for the VIN for this experiment uses 150 kernels of size 3×3 with stride 1 followed by a single kernel of size 3×3 to get $\bar{R}$. The transitions $\bar{P}$ are defined as 3×3 convolutions. We then get $\bar{Q}$ with $\bar{\mathcal{A}}$ channels. After K iterations of VI are done, an MLP is then used to the (8) primitive actions if $|\bar{\mathcal{A}}| > |\mathcal{A}|$.

As for the K recurrence of the VIN, enough iterations are done to ensure that information about the goal flows to the state corresponding to where the agent is located (i.e., at least 1 larger than the floor of the diagonal.) Finally, a curriculum according to the distance needed to "find" the goal through the VI iterations is also tried. As for baselines, a two-layer CNN (first layer with 3 x 3 kernels and 250 filters, second layer with a single 3 x 3 kernel) with tied weights and the same K recurrence is used followed by a flattening and MLP to the (8) primitive actions.

The first experiment that is done with temporally extended actions is a simple one, to test how a VIN can learn to plan with given options of a fixed length. In particular, in addition to the 8 primitive actions (see the section titled Grid World Domain), 16 options corresponding to the locations reachable with a sequence of two primitive actions are added, as explained in the previous section.

Training for such an experiment can be done in different ways; however, the simplest way to do this is to train a single VIN with 24 or greater channels (corresponding to sum total of available primitive and temporally extended actions) in the state-action value layer. This controller is termed the base or metacontroller. Then, once a non-primitive action is chosen, we can look at separate subcontroller(s) for the subplan(s). For this Grid World domain experiment, all that is done is to execute a specific unique fixed set of instructions corresponding to each option chosen.

For the Four Room domain, this is a bit more complicated. The only differences from the Grid World domain is that the no primitive actions (only options) are used in the metacontroller and that the separate subcontrollers are other VINs, one corresponding to each option. The options in this domain are chosen to be the 8 "player location in room to opening" actions as well as one "go to goal" option for a total of 9 options. Thus, assuming the rooms are labeled Room 1, 2, 3, and 4 from the top-left clockwise, these are temporally extended actions that can be described as, for example, "Player located in Room 1 (this is to be found by the meta-controller and is an assumption to be made by the sub-controller), move to the Room 2 opening" meaning the action moves the player that is currently somewhere in Room 1 to the Room 2 opening (to the right of Room 1.) In this scenario, the base controller is to make complex decisions like which options to take, and in what order as well, in order to reach the goal the quickest. The sub-controllers on the other hand just have to do something very similar to the VIN in the Grid World domain, find the shortest path to the nearest opening corresponding to the option (i.e., if option 5 is one that moves an agent from Room 3 to the Room 3 / Room 4 gap, and there are say 3 such gaps, the sub-controller should both determine which one is the closest and find the shortest path to it.)

Next, the kernel sizes used for these experiments which correspond to the discounted transition probabilities are also of interest.

Options generally do not obey local connectivity (i.e., the discounted transition probability of reaching far by states is not necessarily 0 and, on the other hand, reaching nearby states is in fact less likely as option lengths increase.) Thus, we must change how the transition probabilities are calculated, and, in particular, weaken any local connectivity assumptions.

One obvious way to do this is to increase the size of the transition kernels. For the Grid World Domain experiment with the length 2 options, this is quite trivial. Increasing the kernel width and height to 5 (or using Atrous Convolutions to keep the number of parameters low, as described in the section titled "Convolutional Neural Networks") solves the issues. This is done and a comparison of each of the techniques can be seen in the results section.

For the Four Rooms domain as well, the metacontroller does something very similar. Using a kernel size such that, from every position in a room, all gaps between this room and another can be seen suffices. Thus, the 3×3 kernels were changed to 20×20 kernels. However, the subcontrollers continue to use 3×3 kernels (with the 8 primitive actions) as they are still working in that small temporal scale.

For this problem of non-local connectivity, using other techniques is also possible. For example, fully connected layers, while extreme in the number of additional parameters, would be a fine solution. Some others include, for example, Spatial Transformer Networks [7] and possibly some modified version of non-uniform dropout with Fully Connected layers like in [10].

2.8 Training

Value Iteration Networks are networks whose goal is to learn to plan. With this goal in mind, there are various ways to train such agents.

Firstly, there is a choice between using the traditional RL setting with rewards produced by the environment after each action, and a supervised setting where the training signal contains the desired actions themselves. Secondly, there is a choice between when to plan and when to act. Lastly, there is also a related question about how long or deep to plan and how many actions to take after each planning step.

Considering the focus of this research (into implementing, analyzing and extending VINs through temporally extended actions) and as a starting step, a simple training regime is used.

Namely, supervised imitation learning is used where the desired action from each state is given (from an expert who has computed a shortest path from the player location to the goal; in the experiments, a simple BFS algorithm is used). Since temporally-extended actions are just another action in the base or metacontroller, imitation learning also provides these action or rather option choices as the target when it is optimal to execute an option. The base controller and sub-controllers are thus trained with the appropriate imitation learning supervised signals.

Continuining on the description of the training regime, a reward image is first constructed from the state using convolutional layers as described in the previous section. Then, Value Iteration is done for some I steps where I is large enough such that each state gets reached by atleast one sequence of actions from the current state.

After these I iterations of VI, the agent is to produce just one action. This is sound because in the training process, as random starting locations and goal locations are chosen in the various environments and all state-target action pairs along the shortest paths are chosen (always keeping the obstacle and goal locations the same throughout), the training will lead to plans that are optimal throughout (not just in the first action or first few actions). So, as a related note, during training, a success simply denotes a correct choice of the next action to take.

As described before, when a curriculum is used, the agents get to train first on instances where the distance between the player location and the goal is 1 (thus requiring just a single action.) Then, after approximately 1 over the total number of different type of instances (i.e, the max shortest-path distance between any starting location and corresponding goal location), instances of the next difficulty are also introduced. While a curriculum could be considered based on the number of obstacles (chosen between 0 and 10 during training), this was not done as the policies are not necessarily affected just by the number of obstacles (the obstacles could be far off from the player and goal locations.)

In all experiments, 5000 unique training configurations of the obstacles are used and for each configuration of the obstacles, 10 random starting and goal locations are chosen to.

For evaluation, 5000 fixed configurations of the obstacles, without overlap with the training set were used, with 10 random starting and goal locations. Additionally, evaluation runs I iterations of VI just like during training. However, unlike training, a full plan is created until the agent either reaches the goal within the number of actions a shortest path plan would take or hits an obstacle before that. Only the former cases are counted as successes in the accuracy curves shown in the following section.

Note that in this training regime option termination maybe an issue with these choices and is left as a future work. Notably, during evalution, the subcontrollers in the two-primitive action terminate after exactly two actions. In the four room domain experiment, the subcontroller continues to execute actions until it reaches a state with maximal value. Option termination is left as a future work.

Now, it is noted here that this kind of training regime can certainly be tweaked and played around with (for example by interleaving multiple planning steps and multiple action or option executions in more complex domains.) The only non-trivial part is how to provide a loss for action sequences with target sequences in the supervised imitation learning setting. Using trajectory or policy optimization methods such as Trust Region Policy Optimization (TRPO, [15]) is one way to approach this.

3 Results

First, VINs are tested and compared with a baseline CNN as described at the end of the previous section. Figure 5 shows the evaluation or performance on the held out test set through the accuracy as defined in the previous section (full plans are constructed and said to be successful if the agent reaches the goal state without hitting any of the obstacles and with the number of actions exactly equal to that in a shortest path plan.)

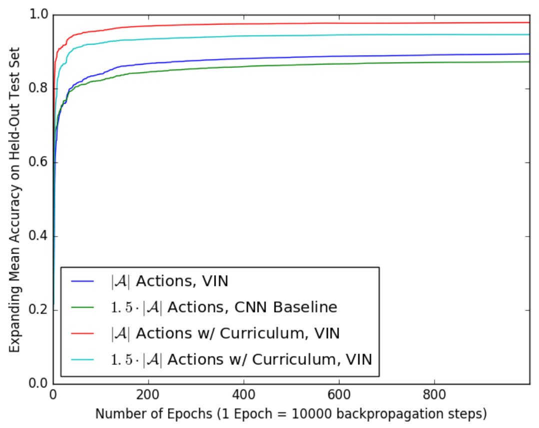

Figure 5: Expanding Mean Accuracy on Test Set

As shown in Figure 5, the baseline CNN does reasonably well (just around 83% accuracy in the plans.) However, VINs, especially when a curriculum is used and with the larger number of actions available at the Q-layer (see the previous section titled ”Training”), outperforms the baseline CNN.

Now, some analysis of the learned reward and value functions, with the plans is done.

We see that as intuition would lead us to believe, the value associated with the goal is set to be high (see Figure 3 for the starting state observation). Many of the other neighboring states also have some high value, but far off states have very small values in comparison.

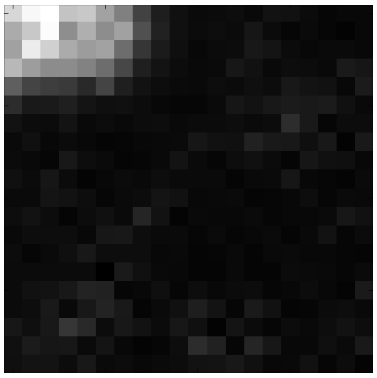

Figure 6: Value Function Image Example

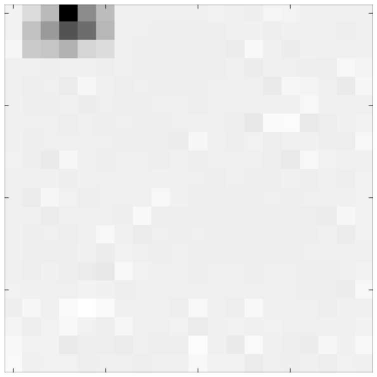

Figure 7: Reward Function Image Example

In Figure 7, there seems to be something interesting going on. Normally, for "good" or "desired" states, like the goal, we would assign high positive rewards. However, it seems that the VIN has assigned a small reward to the goal and in addition, the neighboring states also seem to have rewards shaped as such. However, there is a scaling that is applied to show the above images (to between 0 and 255 for grayscale pixel values), and the actual rewards may have very different magnitudes. Upon inspection of the kernel weights (discounted transition probabilities), it became clear that the negative of the reward is used. In other words, the VIN has learned to plan with a punishment or a negative reward based approach.

Finally, the plan shown in Figure 8 is indeed optimal as the player only ever uses one square of each row of pixels as it aims to go up to the goal and all actions (including the diagonal ones) are taken to be equally costly which induces a Manhattan metric (and not a Euclidean one.)

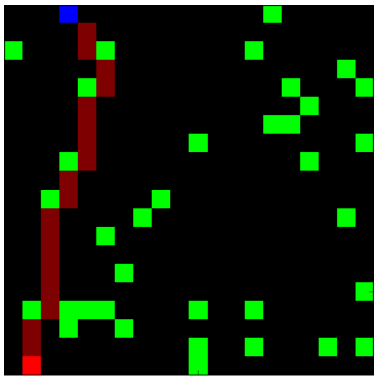

Figure 8: Illustration of a Plan

Next, we see the results after adding temporally extended actions.

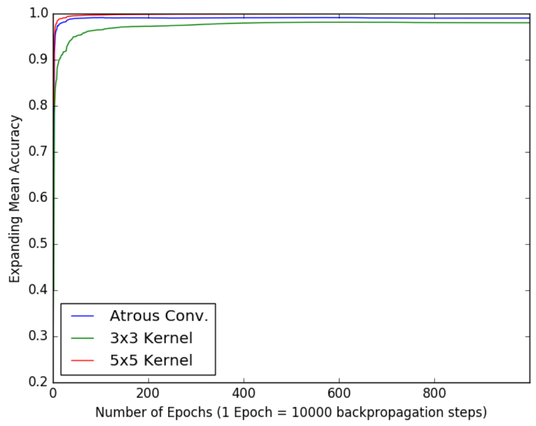

Figure 9: Expanding Mean Accuracy on Training Set

In Figure 9 we see that we get reasonable performance with the 3×3 kernels, 5×5 kernels and Atrous 3×3 kernels (the last of which was used with rate 2 giving 7×7 receptive fields). As intuition would lead us to believe, the 5×5 kernels perform better than

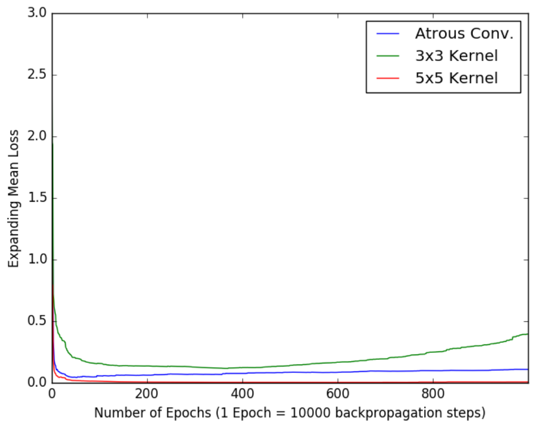

Figure 10: Expanding Mean Loss Over Testing Iterations

the 3×3 kernels. However, while the Atrous convolutions have a larger receptive field, it's containing of holes could be the reason for poorer performance.

From the loss curves Figure 10, we see that, even though the 3×3 kernel performs well on the training set, the loss on the test begins to increase after a certain point, signifying overfitting. Thus, we see the effects of using a discounted transition probability kernel that is not large enough to cover the "span of the options" which is 2 (see the section titled "Metacontrollers and Temporally Extended Actions").

Finally, we consider at the Four Rooms Domain. We get very interesting and optimal plans as shown in **??** and the results are very similar and positive to the previous experiments. Visualizations of the metacontroller's reward and value functions follow for a particular state (Note: This one is different from **??** .)

While the reward function Figure 12 seems to be generally low everywhere except the player location. The reward appears to also be high around the gaps and also maybe importantly one square to the left of the player position. This could signify that the metacontroller is assigning high reward towards moving left (which indeed is in the direction closest to the goal.) Moreover and possibly more significantly, the value function strongly appears to indicate that moving diagonally upward and to the left is the way to go, and indeed, the option it chooses at this point is the Room 3 to Room 3 / Room 4 gap.

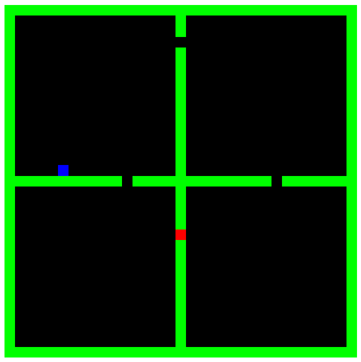

Figure 11: Four Room Domain State Observation

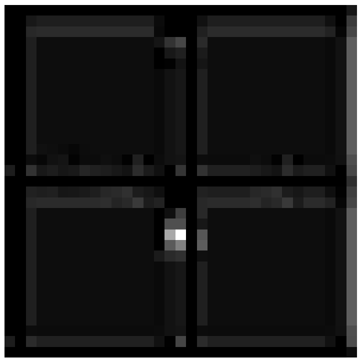

Figure 12: Four Room Domain Reward Function

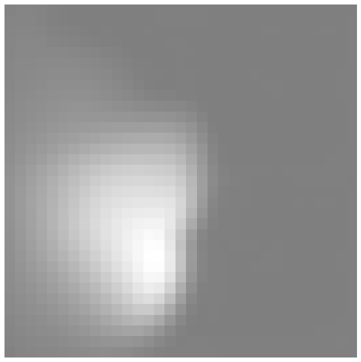

Figure 13: Four Room Domain Value Function

4 Conclusion

Embedding planning into deep learning is an interesting research direction that can be quite fruitful. As we move towards general AI, having long term planning is quite crucial. Differentiable Neural Planners and Value Iteration Networks are useful in their interpretability and interesting as a first step in their inherent MDP modeling which fits well in the Semi-MDP [19] setting as options can be viewed as optimal policies in certain variations of the "primary" MDP that an agent is evaluated in. However, there is still work to be done. For example, while the focus here has been to use VINs as a sort of Neural Program Interpreter where the actions are programs, learning options and using memory to store and erase useful policy embeddings that encode the options is a very important research area. Additionally, while end-to-end learning and use of options can help and there is the disadvantage of needing to specify separate MDPs to train options, there is the consideration that training multiple controllers separately can greatly decrease the overall training time through parallelization. Figuring out how to train large end-to-end hierarchical neural network based agents or how to decompose an MDP and finding some "induced" MDPs that are useful for training options are also very intriguing research directions.

References

[1] Tensorflow atrous convolution github webpage. `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/g3doc/api_docs/python/functions_and_classes/shard7/tf.nn.atrous_conv2d.md`.

[2] Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. Unifying count-based exploration and intrinsic motivation. *arXiv preprint arXiv:1606.01868* (2016).

[3] Bellman, R. A markovian decision process. *Indiana Univ. Math. J. 6* (1957), 679–684.

[4] Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. Benchmarking deep reinforcement learning for continuous control. *arXiv preprint arXiv:1604.06778* (2016).

[5] Dulac-Arnold, G., Evans, R., Sunehag, P., and Coppin, B. Reinforcement learning in large discrete action spaces. *CoRR abs/1512.07679* (2015).

[6] Gu, S., Holly, E., Lillicrap, T., and Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. *arXiv preprint arXiv:1610.00633* (2016).

[7] Jaderberg, M., Simonyan, K., Zisserman, A., et al. Spatial transformer networks. In *Advances in Neural Information Processing Systems* (2015), pp. 2017–2025.

[8] Lample, G., and Chaplot, D. S. Playing fps games with deep reinforcement learning. *arXiv preprint arXiv:1609.05521* (2016).

[9] LeCun, Y., and Bengio, Y. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks 3361*, 10 (1995), 1995.

[10] Li, Z., Gong, B., and Yang, T. Improved dropout for shallow and deep learning. *arXiv preprint arXiv:1602.02220* (2016).

[11] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[13] Oh, J., Chockalingam, V., Singh, S., and Lee, H. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128* (2016).

[14] Rodriguez-Clare, A. The division of labor and economic development. *Journal of Development Economics 49*, 1 (1996), 3–32.

[15] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. Trust region policy optimization. *CoRR, abs/1502.05477* (2015).

[16] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[17] Stadie, B. C., Levine, S., and Abbeel, P. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814* (2015).

[18] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[19] Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence 112*, 1 (1999), 181–211.

[20] Tamar, A., Levine, S., and Abbeel, P. Value iteration networks. *CoRR abs/1602.02867* (2016).

[21] Yu, F., and Koltun, V. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122* (2015).

[22] Zhu, Y., Mottaghi, R., Kolve, E., Lim, J. J., Gupta, A., Fei-Fei, L., and Farhadi, A. Target-driven visual navigation in indoor scenes using deep reinforcement learning. *arXiv preprint arXiv:1609.05143* (2016).