Surprise Pursuit Auxiliary Task for Deepmind Lab Maze

Luke Johnston
Stanford University
lukej@stanford.edu

Abstract

In this paper, I introduce a new unsupervised auxiliary task for reinforcement learning in complex environments such as Deepmind Lab [1]. This is inspired by recent work on unsupervised auxiliary tasks in [3]. The task I introduce is called "surprise pursuit", and attempts to quantify the "surprise" an agent encounters when navigating a partially known environment, and then trains a policy to maximize this surprise, with the goal of helping the agent learn valuable features for exploring an unknown environment such as a maze. This is done by first training the agent to be able to generate predictions of upcoming state observations, and then training the agent to produce a policy to maximize error in those predictions (the "surprise"). Initial results are promising but not definitive - training appears to proceed faster with these auxiliary tasks, but further analysis needs to be done to determine the full extent of the effect of these auxiliary tasks.

1. Introduction

Neural networks have shown great success in both computer vision and reinforcement learning in a given environment, for example image classification on the ImageNet dataset [6] and playing Atari games at and above the human level [5]. As reinforcement learning environments become more complex, novel techniques are required to effectively train neural networks to interpret the environment and produce function estimators for the policy (or value function). For reinforcement learning in complex 3D environments with 2D image environment observations, convolutional networks are best suited for feature extraction of the environment, and for time-dependent environments in which the state is not fully represented by the environment observation, some form of recurrent neural network controller must be used in the computation of the function estimator. There are many difficulties, however, in training such a neural network for reinforcement learning in a complex environment. One major difficulty is that gradient descent on neural networks does not perform well when subsequent datapoints are correlated, as is the case in many standard reinforcement learning algorithms. A number of solutions have been proposed, such as saving observations into a replay buffer that is later randomly sampled from for training [5]. The recent $A3C$ algorithm provides a solution that uses multiple agents training in parallel on different instantiations of the same environment to reduce this correlation of gradient updates. The $A3C$ algorithm achieves very good results on both the Atari environment and the Deepmind Lab environment, but these results are even further improved upon with the addition of auxiliary reinforcement learning tasks, as introduced in [3].

In this paper, my ultimate goal is to train a deep reinforcement learning agent to navigate the 3D labyrinth environment provided by Deepmind Lab [1]. I do this by modifying an existing model first introduced by [3], to add my own novel unsupervised auxiliary tasks to the training process in order to speed up training and achieve better score at convergence. The task is called "Surprise Pursuit" and is motivated by the idea that in order for an agent to learn to explore an environment, it should be able to: 1. build up a representation of the environment, keeping track of areas that it has already explored and knows, and 2. use that knowledge to seek out new areas that it is unfamiliar with. Toward this end, the agent is trained to predict pixels of subsequent states (and if it can do this succesfully, then it knows the area of the environment it is moving toward), and second, learn a policy that maximizes the "surprise" of exploring the environment, where "surprise" is defined as the error in the pixel prediction task. This is explained more thoroughly in the "Surprise Pursuit" section. This idea is somewhat inspired by recent research into intrinsic motivation of reinforcement learning agents [15] and curiosity-based exploration [16]

1.1. Deepmind Lab

Deepmind Lab [1] is a platform for machine learning in complex first-person 3D environments. Deepmind Lab provides various and complex three-dimensional environments for reinforcement learning. In every environment, the agent is a ball moving around a $3D$ environment looking for re-

wards, usually represented as fruits or other objects floating above the ground. For this project, the agent's observation at each state is a first-person image of the environment, represent as $(84 \times 84 \times 3)$ RGB values, although Deepmind lab also provides a few other formats. There are many actions available to the agent for some complex tasks (like jumping or turning on a flashlight), but for the environment used in this project only 6 actions are relevant: moving left, forward, right, and backward, and panning the view left and right. The Deepmind Lab environment I work with in this project is called `maze_random_goal_01`, and is depicted in a top-down view in Figure 1.

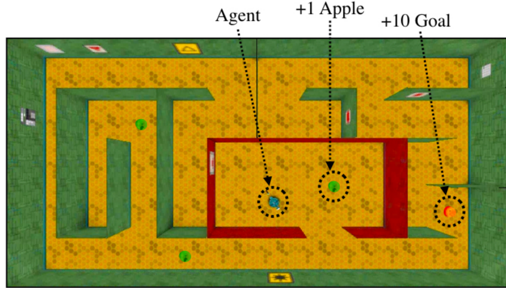

Figure 1. A top down view of the Deepmind Lab maze that I will be working with.

Example input states can be seen in the figures visualizing the network performance in the experiments section. In this environment, every time the agent moves to an apple, it gets a reward of $+1$. Every time it moves to the goal, it gets a reward of $+10$, and its location is randomized. So in order to achieve the maximal reward on this task, the agent must be able to repeatably make its way to the goal, collecting apples when it is worth the detour, wherever its initial position in the maze is. Since the maze is constant across episodes, the agent can learn the maze structure during training, so it does not have to explore the maze entirely anew every time, but it still has to "explore" when it is randomly placed in a location that is not unique by its observations (for example, facing a wall).

2. Background and Related Work

2.1. Reinforcement Learning

In the standard reinforcement learning formulation, an agent in an environment observes a series of states $s_0, s_1, \ldots$ and rewards $r_0, r_1, \ldots$, and at each time step t must use a policy π to select from all possible actions A a single action a_i, which affects both the future rewards and the future obvserved states. The goal of the agent is to learn a policy π to maximize the sum of discounted future rewards from each state s_t,

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

where $\gamma \in [0, 1]$ is a discount factor that motivates the agent to weight quick rewards more than distant rewards.

In policy-based reinforcement learning, the model directly learns a policy funtion $\pi(a|s, \theta) \in \mathbb{R}^{|A|}$, for the distribution of probabilities of each taking action a at state s. To train the model, gradient ascent is performed on the expectation of the discounted future rewards [7]:

$$\Delta\theta = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s, a)}{\partial \theta} Q^\pi(s, a)$$

Where $d^\pi(s)$ is the stationary distribution over states for policy π, and $Q^\pi(s, a)$ is the expected discounted future reward from taking action a at state s, or a function approximator thereof (typically a neural network, for complex environments such as the one in this project). There is a problem with training a neural network to optimize this objective directly with the (s, a, r) taken from interaction with the environment: sequential (s, a, r) observations are strongly correlated, and furthermore depend on the neural network function approximator for the policy, and this correlation causes instability and divergence in training. This can be solved in a number of ways, but the one most relevant to this paper is the $A3C$ algorithm.

2.2. A3C Algorithm

In the recent paper "Asynchronous Methods for Deep Reinforcement Learning", [2] Mnih et al. introduce a reinforcement learning paradigm to decorrelate the data and make deep reinforcement learning possible and efficient. The basic idea is that instead of excecuting one agent on one environment at a time, multiple agents are executed on many environments in parallel, and each agent accumulates gradient updates for its experience. These gradient updates are then periodically applied to a global network that is used to synchronize the paramaters of the asynchronous agents. Their algorithm that achieves the best performance is called "Asynchronous advantage actor-critic". They use two neural networks, one for the policy $\pi(s, a)$, and one for an estimate of the value function $V(s)$ (the value function maps states to expected discounted future rewards under the current policy). The main update to the parameters of $\pi(s, a)$ is

$$\Delta_\theta = \nabla_\theta \log_\pi(a_t|s_t, \theta) A(s_t, a_t, \theta')$$

where $A(s_t, a_t, \theta')$ is an estimate of the advantage function expanded forward over k time steps:

$$A(s_t, a_t, \theta') = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}, \theta') - V(s_t, \theta)$$

An advantage function is used instead of the pure value estimate for variance reduction [12], and significantly speeds up training time. The value network must be trained separately toward the objective $V(s_t) = E[R_t]$. Additionally, a entropy penalization term is added to the gradient update, to discourage quick convergence to local optima of singular policies, of the form $\beta \nabla_\theta H(\pi(s_t, \theta))$, where H is the entropy. To optimize these objectives, A3C uses a simple variant of the RMSProp update [8]:

$$g = \alpha g + (1 - \alpha)\Delta\theta^2$$

$$\theta \leftarrow \theta - \eta \frac{\Delta\theta}{\sqrt{g + \epsilon}}$$

where α is decaying average parameter, ϵ a small stability parameter, η the learning rate, and the statistics g are shared between threads. The A3C algorithm performs very well, surpassing state-of-the-art on the Atari [5] domain. That is, until the next paper made another improvement.

2.2.1 Implementation Details

The results of this paper depend on the implementation of A3C for the task, so I describe them here. The code for this was taken from an existing implementation for the UNREAL agent [11], which is described further below. My own original contributions are described in the "Surprise Pursuit" section.

1. First, each input is passed through a convolutional feature extractor. The convolutional feature extractor has 2 layers. The first applies 16 filters of size 8 to the input state observation, with a stride of 4 and "VALID" padding, to get a representation of size $(20, 20, 16)$. This is passed through the nonlinear ReLU activation, and then the second layer. The second layer applies 32 filters of size 4, with a stride of 2 and "VALID" padding, with ReLU activation again, to obtain a final feature representation of the input of shape $(9 \times 9 \times 32)$. Finally, this is flattened and passed through a fully connected + ReLU layer to obtain 256 features for the input state.

2. These features are then provided as input to a Long Short-Term Memory (LSTM) [4] controller with 256 cells. This allows the model to incorporate temporal information into the policy and value functions.

3. Finally, the output of the lstm controller is passed through two feed-forward networks: one for the policy and one for the value function. The first feed-forward netword has $|A| = 6$ outputs, and the softmax is taken to compute the policy action probabilities. The second feed-forward network has a single output, which is the value function approximation.

2.3. UNREAL Auxiliary Tasks

In the paper "Reinforcement Learning with Unsupervised Auxiliary Tasks", Jaderberg et al. add "unsupervised auxiliary tasks" to the A3C algorithm that significantly improve final performance and training spped on all tested environments, including Atari and Deepmind Lab (called Labyrinth in the paper). In many reinforcement learning environments, rewards are not observed frequently enough to apply meaningful learning during every iteration of training. But we would like the agent to be learning something during the time when rewards are not observed anyway. Hence, introduce auxiliary tasks, which are tasks added to the loss function that require the agent to learn information in an unsupervised fashion, which may not be directly used in the reinforcement learning policy, but which nevertheless helps the agent interpret the environment. These can be understood by example:

2.3.1 Pixel Control

One auxiliary task is the "Pixel Control" task. The motivation for this task is that an agent should understand how its actions will affect the environment. One way of representing this is to learn a policy that controls how pixels on the screen change. In the Deepmind Lab (*Labyrinth*) environment, state observations are $(84 \times 84 \times 3)$ RGB values of a 2D perspective of the 3D environment (see visualizations in Figures(4-6)). For this task, the agent is trained to control a central crop of (80×80) pixels. This crop is dividing into a (20×20) grid of windows viewing the state. For each of these 400 windows, a policy is trained to maximize total pixel change in that window from one state to the next. So we have 400 $(4 \times 4 \times 3)$ RGB windows, and the reward for action a_t in state s_t in each of these windows is the average absolute difference of the pixels in this window between the current frame and the next one. The network that produces this policy takes the LSTM outputs of the A3C algorithm as input, maps them to a $(32 \times 9 \times 9)$ feature map with a affine + ReLU layer, which is then passed through two transpose convolution layers with filter size 4, stride 2 to obtain a dueling network [9] parameterization of the Q_{pc} values for each policy of size $(|A| \times 20 \times 20)$. The target loss for these Q values is simply the l2 loss of their difference the estimated future rewards, expanded for k times steps:

$$L_{pc} = (Q_{pc}(s_t, a_t) - \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k \max_a Q_{pc}(s_{t+k}, a))^2$$

These policies are never used during exploration - they are only trained to help the model learn something useful for exploring, interacting with, and understanding the environment.

2.3.2 Reward Prediction

In this auxiliary task, the past three state observations are used to predict whether the reward for the next transition is positive, negative, or zero. The past three state observations are each passed through the convolutional feature extractor, and then the resulting features are concatenated and passed through a single fully-connected + ReLU layer to 128 activations, which are passed through a final fully-connected + softmax layer to classify the reward. Training this task requires the convolutional layers to learn to extract features useful for predicting rewards.

3. Surprise Pursuit

To the two original auxiliary tasks of the UNREAL agent, I add my own two auxiliary tasks: pixel prediction, and Surprise Pursuit. Pixel prediction has been explored in previous work [10, 3], and alone is not as effective as pixel control, but is necessary for the Surprise Pursuit task, which is the main contribution of this work.

3.1. Pixel Prediction

If an agent truly understands its environment, it should be able to make reasonable predictions about how its actions will affect the environment, and consequently its observations of the environment. Toward this end, I implement the auxiliary task of pixel prediction: at each state s_t, given that the action taken is a_t, the model must generate a prediction of the pixels of the following observation s_{t+1}. This is not a novel auxiliary task, but rather than use an existing implementation, I wrote my own implementation of this task. At state s_t, the LSTM outputs are passed through a fully connected + ReLU layer to map them to a $(32 \times 9 \times 9)$ representation of the controller state. This is passed through three transpose convolution layers:

1. A transpose convolution layer with 32 filters of size 4, ReLU activation, stride 2, and VALID padding maps the representation to size $(32 \times 20 \times 20)$.
2. A second transpose convolution layer with 32 filters of size 4, ReLU activation, stride 2, and VALID padding maps the representation to size $(32 \times 42 \times 42)$.
3. A third transpose convolution layer with 3 filters of size 4, sigmoid activation, stride 2, and SAME padding maps the representation to size $(3 \times 84 \times 84)$, the size of the image we are trying to predict. Sigmoid activation is chosen because the images are represented as RGB values, normalized in the range $(0, 1)$, so our predictions must also lie in this range.

The loss is then the l2 loss between the predicted image $P(s_t, a_t)$ and the state observation s_{t+1}:

$$L_{pp} = (P(s_t, a_t) - s_{t+1})^2$$

. This formulation is inspired by the autoencoder structures of [13], and generative adversial network structures of [14], which show that transpose convolution is an effective means of generating images (although this network is simpler than the citations).

3.2. Surprise Pursuit

For an agent to effectively learn to explore an unknown territory such as a maze of Deepmind Lab, an obvious approach would be to first learn how to encode areas of the territory that are known, and then to seek out areas that are unknown. In the broader context of reinforcement learning, it is important to explore as much as possible of the state space in order to obtain accurate estimates of all possible rewards. This is the motivation for the Surprise Pursuit auxiliary task: the notion of exploring unknown areas is approximated by a policy that seeks out "surprise", which I will quantify below. First, the pixel prediction auxiliary task provides a measure of how well the agent knows a particular (state, action) pair. If the pixel prediction loss is very low, then the agent is able to accurately predict what the future state will look like - and has a good model of the environment for that state and action. However, if the pixel prediction loss is high, then the agent's prediction of the future state is incorrect - it is "surprised" with the result. Hence, I define the surprise of a (state,action) pair as the negative of the pixel prediction loss, $S(s, a) = -L_{pp}(s, a)$. Then, the Surprise Pursuit task trains a policy to maximize surprise, in order to hopefully learn a policy that advocates for exploring unknown territory, which will help for locating the rewards in a maze after the agent's location is randomized (and it does not know where it is). Following a similar approach to the pixel control policy of the UNREAL agent, dueling networks are used to estimate Q values for each action, and the surprise pursuit network has the following architecture:

1. the LSTM outputs of the A3C architecture are passed through a fully connected + ReLU layer to obtain advantage-values $A_{sp}(s_t, a_t) \in \mathbb{R}^{|A|}$
2. the LSTM outputs of the A3C architecture are passed through a separate fully connected + ReLU layer to obtain a value function estimate $V_{sp}(s_t)$.
3. the final Q-values are computed $Q_{sp}(s_t, a_t) = V(s_t, a_t) + A(s_t, a) - \texttt{mean}(A(s_t, a_t))$ where the mean is taken over the action advantage values [9].
4. the network is optimized to approximate the surprise

reward with the Q values:

$$L_{sp} = (Q_{sp}(s_t, a_t) - \sum_{i=0}^{k-1} \gamma^i S(s_{t+k}, a_{t+k}) + \gamma^k max_a Q_{sp}(s_{t+k}, a))^2 \quad (1)$$

It is important to note that when optimizing with respect to this loss, the gradients through this loss to the Surprise are frozen, since we do not want learning of pixel predictionsn to be affected by learning of this policy.

4. Experiments and Results

Training was done on a Google Cloud instance with 32 CPU cores, until convergence (or as long as possible). Most of the default parameters of the implementation were kept, which correspond to [3]. The losses for each auxiliary task were adjusted until they were of relatively the same magnitude, so that training does not overly emphasize any one task. The final loss weightings were as follows: pixel control was weighted by $\lambda_{pc} = 0.05$, surprise pursuit weighted by $\lambda_{sp} = 5 \times 10^{-8}$, and pixel prediction weighted by $\lambda_{pp} = 10^{-3}$. Training to convergence takes more than 20 million iterations, and at a maximum of 200 iterations per second, this takes more than 24 hours. The unmodified UNREAL agent was trained with the same configuration to use as a baseline. The score curves for the baseline, and for my model are depicted in Figure 2, and loss curves for each auxiliary task for my model are depicted in Figure 3. The full run for my agent (the purple line) had a bug in the surprise pursuit loss computation that I did not discover until recently, so the orange line depicts a debugged run that is not yet fully complete. This run looks very promising. The smoothed curve has been consistently above the baseline for more than 7 million iterations. However, since there is a massive amount of variance in individual episode scores, and since training is stochastic in the first place, this could just be a lucky run. In order to determine whether my modification is definitively better than the baseline, I have to do a couple things: first, let the run go to completion to see if it converges to a better policy. If early training is faster at the sacrifice of final score, the modification is probably not worthwhile - although that would be a interesting tradeoff to explore. Second, training would have to repeated from scratch a number of times to ensure that the different scores and training speeds are not only due to the stochasticity of training runs. Third, I would need to run another baseline with the pixel prediction task, but without the Surprise Pursuit loss, to determine the relative effect of each addition, since pixel prediction alone could be the cause of any observed improvement. Unfortunately I did not have time for this.

A few visualizations of the agent's different tasks during an evaluation episode are depicted in figures 4-6, updated from the original UNREAL implementation to include the pixel predictions and a representation of the surprise pursuite policy q-values. Furthermore, a video of this visualation for an episode is included in the supplementary materials.

5. Conclusions

Using the visualization video (Figures 4-6), we can qualitatively lanalyze the performance of the pixel prediction and Surprise Pursuit auxiliary tasks. The most important thing to check is that the pixel prediction is working, since if pixel prediction isn't working, then the surprise pursuit will be impossible (or at least, mean something very different than I hoped). We can see from the figure that pixel prediction doing moderately well - at the very least, it has figured out how to generate a hazy representation of the maze location the agent is at. It is hard to tell if it is succesfully predicting one frame ahead, or if it is just reconstructing the current state for an approximately correct prediction of the future. To get a better idea if the model is indeed predicting a frame ahead, I looked at the very last frame after the agent reaches the goal (and right before its location is randomized). If the model is predicting ahead, it should have no idea what the next frame will be, so the prediction should be mostly meaningless (something like an average of all states). However, this is not entirely the case - although the prediction at this state is hazier than most predictions, it still looks more similar to the current state than it should. So this is both good and bad news - the model is learning to predict into the future a little bit (or else this prediction would be no hazier than normal), but it is still mostly just reproducing the current state. This makes sense, as reproducing the current state would be a solution that achieves a relatively low loss, without requiring the model to learn any temporal information at all, and training might get stuck here.

Secondly, I have printed out the Surprise Pursuit policy Q-values. As can be seen from the three figures (and better from the video in the supplementary materials), this policy is much more stochastic than the main reward-seeking policy. This suggests that the Surprise Pursuit policy either 1. does not have as much meaning as I had hoped, or 2. is not being learned effectively. I suspect that learning of this policy could be significantly improved by improvements in the pixel prediction, which while not completely terrible, is not very accurate. There are a couple other options for improvements in the "surprise pursuit" idea. Instead of predicting all the pixels of the screen, which is difficult to do correctly but easy to do incorrectly by copying the current state, we could predict changes in pixel values, and measure surprise on those predictions. Or we could predict changes

in activations of key internal states of the neural network. This would require less predictions than pixel prediction, and focus the prediction on the important features of the state (rather than the pixel representation), and may converge faster to a non-trivial result.

In conclusion, the Surprise Pursuit auxiliary task looks promising for improving training speed and possible increasing converged score on the Deepmind lab `maze_random_goal_01` environment, although further training and analysis is necessary to determine the full extent of its effect.

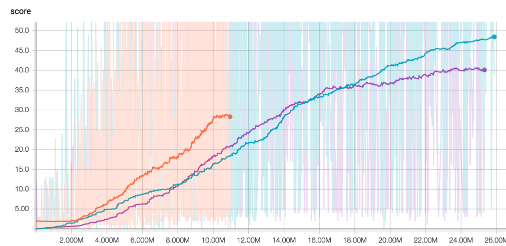

Figure 2. The scores of three different training runs, by iteration. Exponential smoothing is applied, with a paramater of 0.996. The blue run is the baseline, the purple is a complete run of my model which had a bug that I did not discover until very recently, and the orange curve is the most recent run with the bug fixed. A score of 50 represents getting to the goal 4 or 5 times over the course of a single episode, 300 seconds. Human score is easily above 100, but a random agent will rarely get more than 1.

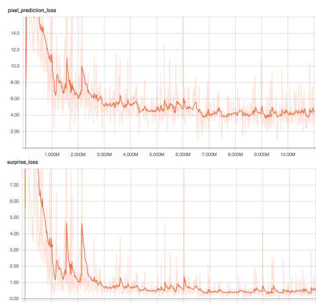

Figure 3. The loss curves for the pixel prediction (top) and Surprise Pursuit (bottom) tasks. Loss does decrease initially, but remains fairly constant after the first 5 million iterations.

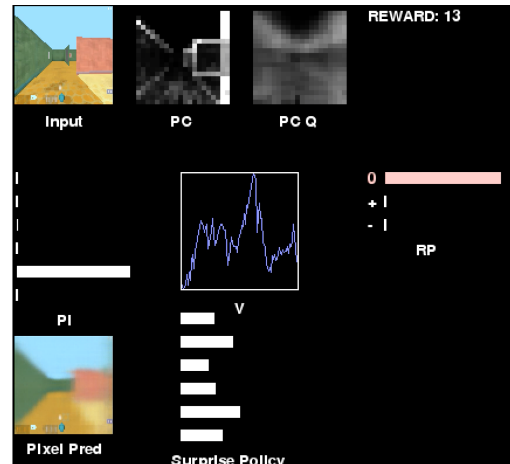

Figure 4. One screen capture of the visualization tool. This tool is taken from [11], but I modified it to display the pixel predictions and policy of my Surprise Pursuit auxiliary task. The game state is in the upper left corner. The pixel predictions for the next state is in the lower left corner, and the Surprise Pursuit policy Q-values are to the right of the surprise prediction. The rest of the figures depict the reinforcement learning main policy, the pixel control policy auxiliary task, reward prediction auxiliary task, and value estimation of the original UNREAL implementation

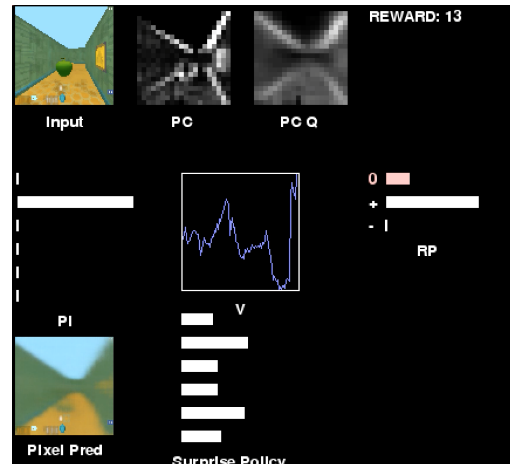

Figure 5. A second screen capture of the visualization tool, with an apple on the screen. Apparently the pixel prediction network does a very poor job of representing apples or goal states. See conclusions section for further analysis.

References

[1] Beattie, Charles, et al. "DeepMind Lab." arXiv preprint arXiv:1612.03801 (2016).

[2] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International Conference on Machine Learning. 2016.

[3] Jaderberg, Max, et al. "Reinforcement learning with unsupervised auxiliary tasks." arXiv preprint arXiv:1611.05397 (2016).

[4] Hochreiter, Sepp, and Jrgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[5] Mnih, Volodymyr, et al. "Playing atari with deep re-

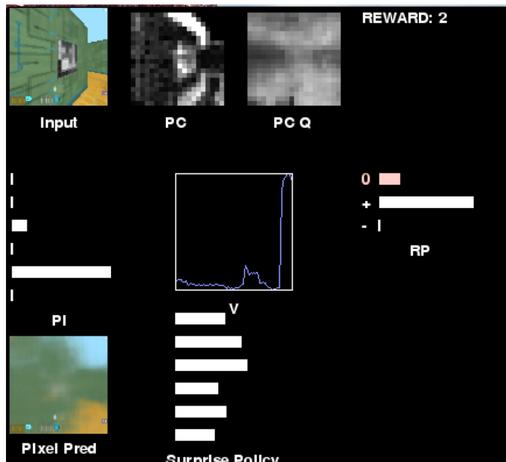

Figure 6. A third screen capture of the visualization tool, directly after the agent reaches the goal and right before its position is randomized. See conclusions section for analysis.

inforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[6] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[7] Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." Advances in neural information processing systems. 2000.

[8] Tieleman, Tijmen, and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural networks for machine learning 4.2 (2012).

[9] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXiv preprint arXiv:1511.06581 (2015).

[10] Kulkarni, Tejas D., et al. "Deep successor reinforcement learning." arXiv preprint arXiv:1606.02396 (2016).

[11] https://github.com/miyosuda/unreal

[12] Peters, Jan, and Stefan Schaal. "Natural actor-critic." Neurocomputing 71.7 (2008): 1180-1190.

[13] Masci, Jonathan, et al. "Stacked convolutional auto-encoders for hierarchical feature extraction." Artificial Neural Networks and Machine LearningICANN 2011 (2011): 52-59.

[14] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

[15] Chentanez, Nuttapong, Andrew G. Barto, and Satinder P. Singh. "Intrinsically motivated reinforcement learning." Advances in neural information processing systems. 2005.

[16] Schmidhuber, J. (1991b). A possibility for implementing curiosity and boredom in model-building neural controllers, in Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats, eds J. A. Meyer and S. W. Wilson (Cambridge, MA: MIT Press/Bradford Books), 222227.