A Student-Teacher RL Algorithm for Neural Program Synthesis with Visual Feedback

Jonathan Booher
Stanford University
Computer Science
jaustinb@stanford.edu

Matthew Sun
Stanford University
Computer Science
mattsun@stanford.edu

Abstract

Neural code synthesis has proven an extremely difficult task for computers to learn, even with hand-crafted objective functions. Solutions tend to be narrowly tailored, and often exhibit a tradeoff between computer performance and human readability. We aim to approach the problem of neural code synthesis through visual programming, a human-readable form of computer programming. Visual programming is a popular way to introduce computer science students to foundational programming concepts, and is taught in many computer science curricula such as Karel and Code.org.

We hope to produce meaningful advances in the field of program synthesis by exploiting several unique aspects of this task - specifically, teaching an agent how to program - that draw on cognitive psychology and reshaping the task as a strictly RL problem rather than a text/code generation problem. We demonstrate that a reinforcement learning algorithm is able to efficiently learn visual programming tasks via a "curriculum" of increasing difficulty, similar to how students might learn computer science. In particular, we note that as the agent progresses through the curriculum, it is able to achieve the current task relatively quickly through few-shot learning, indicating that it leverages the concepts learned in prior tasks.

1. Introduction

1.1. Motivation

Program synthesis is a well-known problem with numerous applications. The increased complexity of embedded software, the proliferation of numerous coding paradigms and software platforms, and growing demand for tools to aid in the development of software for programmers and non-programmers alike call for the development of AI-assisted software development tools. Successful code synthesis programs could one day automate the different steps of computer programming, including debugging, refactoring, and synthesizing code. As Neel Kant notes, there are many problems that can be framed as programming questions, such as proving or disproving a predicate. [3]

Neural program synthesis in a human-understandable, high-level language like Python is widely acknowledged to be a difficult problem without clear indications of promising directions to follow. Recent advances in neural program synthesis have been limited to uncommon programming languages like BF, which is arguably more machine-readable than human-readable, and rely on human-selected loss functions for particular programming objectives. [1]

Deep learning has been shown to excel at a variety of difficult tasks, often matching or exceeding performance of humans, particularly in tasks within the realm of computer vision.[4] In the last few years, convolutional neural networks (CNNs) have been applied to a broad range of vision-related problems, including automated image detection and segmentation [9], image-captioning [11], and even medical diagnosis from imaging modalities [8]. These advances in computer vision seem to have potential for ameliorating one of the fundamental difficulties of training program synthesis algorithms: because the chance of randomly generating a syntactically and semantically correct program is tiny, rewards are extremely sparse, explaining why other research has opted for hand-crafted loss functions that approximate some sense of "distance" to the correct solution. [1]

1.2. Visual Feedback and Framing the Task of Program Synthesis

We hypothesize that visual feedback is immensely helpful for training program synthesis, and posit that it can not only mitigate the need for hand-crafted heuristic functions, but also better represent the task of learning to program. Taking a cognitive psychology view of teaching programming, it is clear that human students learning how to program do not learn by essentially typing random characters and gaining rewards whenever they happen to create a program that compiles. Instead, they are guided through a cur-

riculum of increasing difficulty. Many introductory curricula such as Karel [6] and more recent interactive modules on Code.org are highly visual in nature, allowing students to gain immediate feedback and build a mental connection between on-screen actions of Karel to the commands they've typed, compiled, and run. These mental connections formed in the beginnings of one's computer science education make conceptualizing and imagining the execution of a program much easier, even when no expressly visual output is provided. (It should be noted that during debugging, many programmers appreciate visual tools like stepping through stack traces or manually inserting print statements, providing some visual feedback that helps provide insight into underlying bugs or processes.) The human brain is especially adapted to efficient learning from visual information. [7]

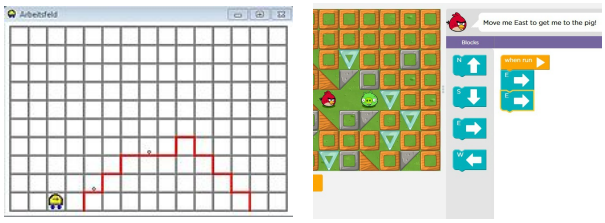


Figure 1. Left: a screenshot of the graphical Karel display of program execution. Right: An image of the Code.org Angry Birds educational visual programming game. Nearly everything about this educational module is visual: from the display of program execution on the left, to the input options on the right, with arrows indicating the direction of movement in the game environment.

Viewed in this light, most traditional program synthesis tasks are actually unfairly difficult for artificial agents. With debugging tools, stack traces, and computational graph visualizations, humans have access to visual feedback that provides extra insight into program execution dynamics. Moreover, we typically have the benefit of being taught a curriculum that is intentionally ordered in increasing difficulty, allowing us to draw on concepts learned earlier on during the educational process. In contrast, artificial agents are typically given no such access to any kind of feedback beyond a binary reward or heuristic function approximating the distance to the correct output of the program. While a human would likely succeed at most image classification tasks (i.e., be able to categorize novel images given similar images with labels), a human would most likely not be successful in traditional code synthesis tasks. That is, a person who is given no prior knowledge of program syntax, is asked to emit a set of characters and observe the reward. And yet, this problem may become tractable when the human is able to see intermediate steps of program execution or observe the effects on input throughout the duration of the program.

We seek to make the code synthesis problem more tractable and more analogous to the process of real-world programming by incorporating visual feedback into the training. We frame the problem not as a traditional natural language processing (NLP) task but rather as a reinforcement learning (RL) problem that involves a student (agent) constructing a program and observing its execution. In particular, we build a simulation based off of the Angry Birds visual programming module on Code.org (Link here: `https://studio.code.org/hoc/1`). In the Angry Birds module, the user is restricted to a particular set of "code blocks." We use this as inspiration for developing a restricted action space for the RL agent to choose from. In addition, once the user clicks "Run," the bird avatar in the game environment begins to execute the code blocks inputted by the user. Similarly, our agent must explicitly specify a "Run" action once it is done constructing the program and then observe the execution. The details of this process are explained in our methods section.

2. Related Work

There has been much work on the problems of neural program synthesis, improving reinforcement learning, and curricula. However, much of the research is brand new and many of the results leave room for improvement on various fronts.

2.1. Neural Program Synthesis

Previous attempts at neural program synthesis have largely focused around the generation of text tokens, characters, or, in some cases, larger subroutines. These approaches have the advantage of being able to directly leverage many of the aspects of the programming language. However, major drawbacks exist; namely dealing with variable names and the intricacies of the datasets in which to train them off of (i.e., code scraped from GitHub). Additionally, these approaches rely on human engineered measures of closeness like edit distance to samples in the dataset or measures of similarity between two generated code snippets (ie ranking outputs). These approaches simply cannot be used to solve the generation of novel algorithms at a large scale because there are too many hand-crafted components. There has been discussion of ways to incorporate elements like stack traces to help allow these methods to generalize. Stack traces, however, are text, and text is historically difficult to train with as the models are slow to converge and suffer from stability problems.

2.2. Imagination and Memory

When humans program, it is often useful to think ahead– to plan and imagine what is happening at each step of the program. This is one of the reasons that print statements are widely used for debugging purposes, as we can follow the execution of the program to see exactly where problems emerge. Until recently, the only notion of imagination in

the field has been backward-looking. This is to say, the focus was on attention, or choosing the most important previous tokens that influence the current token. Recent developments in reinforcement learning, however, have shown that mechanisms interpreted as read only memory or imagination can greatly improve performance and speed up learning. Many of these mechanisms are inspired by neurological phenomena like the actions of the basal ganglia, which is associated with imagination and planning. [10] Most recently, ideas of imagination have been proposed where a complex sub-network containing a variational autoencoder is trained alongside a policy network to predict what the next state will be before action is taken. This allows a form of rollouts where the agent can imagine different trajectories or sequences of actions to inform its decisions. This particular mechanism, developed by DeepMind researchers Wayne et al., has seen tremendous results but is computationally expensive during training. [12]

2.3. Curriculum

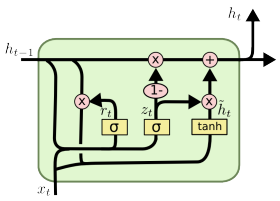
Curricula are common in areas like natural language processing where they have been used for years. For instance, when training language models, it is often beneficial to train on progressively longer subsequences so that the model can learn grammar and syntax. The downside of these curricula is that they require large amounts of time and compute resources to complete. This is largely because of the inefficiency of these algorithms in the sense that there are $n - k + 1$ substrings of length k from a string of length n and traditional curriculum training requires you train on all of them. For this reason, curricula have not widely been used in reinforcement learning because RL is already sample inefficient.

3. Methods

We propose solutions to the problems and gaps in the research above in order to make progress on the complex problem of neural program synthesis.

3.1. Neural Program Synthesis

To address some of the problems that have plagued neural program synthesis in the past, we propose a couple different solutions. The first is what we call a "visual stack trace". As previously stated, stack traces are seen as one possible avenue toward better program synthesis; however, these traces are limited by their inherent text structure and the limitations of the models that handle that information. Visual input, however, does not fall into that trap. Networks dealing with images are well established, relatively quick to train, and extremely expressive. This visual stack trace comes from the environment we run our agent in: a game where an agent outputs tokens representing blocks in an attempt to navigate another agent through a world with walls



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2. A diagram of the structure of an LSTM. We minimized the L2 distance between the encoding of the next timestep and the "imagined" encoding of the LSTM in order to facilitate the agent building an accurate model of the dynamics of its environment.

to an end state. The entire world is shown to the agent via an image and the agent learns to navigate to the end. This environment also allows us to frame the problem as an explicit RL problem. This framing removes the need to parse difficult datasets and normalize against variable names, architectures, and styles.

These blocks also serve to abstract away the intricacies of the programming language itself and put the focus on the algorithm and actually solving the environment. We remove the necessity of the agent to learn to match braces or properly define function prototypes. In this way we avoid some of the major problems that have slowed neural program synthesis in the past.

3.2. Imagination and Memory

Much of the recent research on improving memory in models to increase recall has produced great results. However, there are typically high computational costs to these mechanisms that not all people have access to. As a result, we look at a simpler method of achieving a similar result using a recurrent policy with an LSTM that is trained such that at every timestep the hidden state of the LSTM after forward pass should approximate an encoding of the next state. The general structure of the LSTM is shown in Figure 2 This is largely intuitive; the hidden state that the RNN passes on to the next timestep should contain all the information the LSTM needs in order to predict the next best action. Learning is made simple and interpretable if that state is an encoding of what the next state should be. We call this a simplified "imagination loss," and it is calculated using the state, action, reward nature of reinforcement learning. We can simply minimize the L2 distance between the encoding of the real next timestep and the "imagined" encoding of the LSTM.

3.3. Curricula

Curricula in RL have not been popular. However, we can see that the environment itself and how it produces its initial state can be seen as a curriculum of sorts. Just like a student learns to write code, the environment can present simple worlds to the agent to get it to learn the basics and

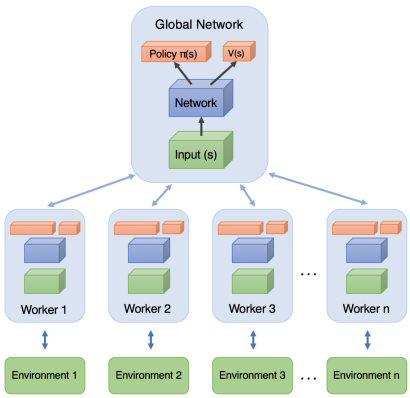


Figure 3. A diagram of the A3C network architecture. (Source)

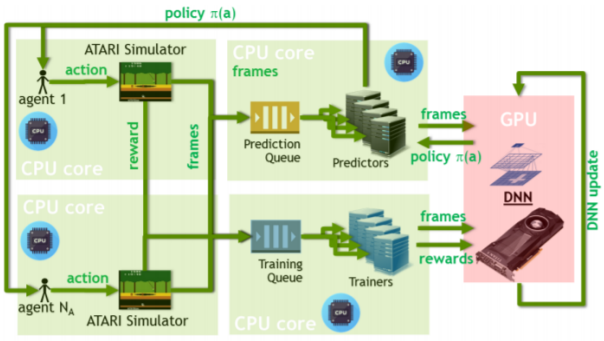


Figure 4. A diagram of the GA3C network architecture. Note that there is only ever one GPU instance of the model, rather than having multiple agent replicas of the model. In this context, trainers assemble experiences for the purpose of updating the network model itself, while predictors query the network for policies.

gradually give the agent harder and harder environments to work on. We propose creating a curriculum the following way. First, train the agent on a static map until it reaches a threshold for number of successes over a sliding window (we used 70%). This allows us to ensure that the model has a basic understanding of the world in which it is acting. We then begin to vary the solution path and the orientation of the agent in the world. We train on a given map for a certain number of episodes dependent on the number of successes that agent has on it (i.e., if the agent has a high number of successes on a map, we move on to the next one). This allows us to ensure that our model sees new environments and is not overfitting to one particular path or world configuration.

3.4. Model

The model we used was based on the GA3C paradigm [2] which is an extension of traditional advantage actor critic (A3C) [5] that runs on GPUs. This is important for our implementation as we will rely on convolutional and recurrent models that would be computationally intractable on CPUs. The asynchronous updates also serve to allow us more samples for a given set of weights for the network leading to better stability. The model itself has four main components: an encoder, decoder, policy head, and value head.

3.4.1 Encoder

This sub graph of the model is responsible for encoding images and producing a latent space for the recurrent part of the model. We draw on the encoder-decoder idea where we take the initial image (ie the initial state of the world) and encode it into a latent space for the decoder. The encoder that we used relies on VGG-16 without its last layer to produce a latent space of 4096 units. VGG was chosen as it is a relatively light network when compared to networks like DenseNet and ResNet but at the same time VGG is still extremely expressive.

3.4.2 Decoder

This sub graph of the model is an RNN composed of a single LSTM with 4096 units wrapped in an attention mechanism. The 4096 units is used to enable easy implementation of the imagination loss described above. The attention is there to enable the model to learn long term dependencies in the code and allow for the learning of the amount of syntax needed for the model to use control structures.

3.4.3 Policy Head

This sub graph of the model is responsible for predicting actions. It takes in the output of the RNN and outputs a softmax over the entire action space in order to allow the agent to pick the best action possible. In principle, this graph could be any network, our implementation relies on a single dense layer. Additionally, this head is used for computing the policy gradient portion of the loss.

3.4.4 Value Head

This sub graph of the model is responsible for evaluating actions. It takes in the output of the RNN and outputs a a single number to try to evaluate the transition in accordance to the rewards from the environment and the calculated advantages. In principle, this graph could be any network, our implementation relies on a single dense layer. Additionally, this head is used for computing the value portion of the loss.

3.4.5 Optimization

There are two cost functions associated with the policy function and the value function. For the policy function, the cost is the following equation:

$$f_\pi(\theta) = \log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta_t)) + \beta H(\pi(s_t;\theta)))$$

where θ_t are the parameters θ at time t, $R_t = sum_{i=0}^{k-1}\gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_t)$ is the estimated discounted reward from time t to time $t+k$ and k is upper-bounded by the maximum t update interval.$H(\pi(s_t;\theta))$ is an entropy term designed to favor exploration during the training process, while β subsequently controls the strength of this entropy term. For the value function, the cost function is as follows:

$$f_v(\theta) = (R_t - V(s_t\theta))^2$$

Training is performed by performing backpropagation on both of the cost functions and using the Adam optimizer to subsequently make weight updates.

3.5. Training Environment

We developed our simulation within PyGame Learning Environment (PLE), an open-source Python library for developing game environments with graphical interfaces that allows for a quick start to reinforcement learning. An image from the simulation can be seen in Figure 5

Figure 5. A screenshot of the PyGame Learning Environment. The objective of the game is for the red bird to reach the blue pig. To do so, the RL agent must input a set of code blocks that express instructions for where the red bird should move. The bird instantly dies if it touches a wall and only receives a reward when it reaches the sames pot as the green pig. See the similarity to the Angry Birds coding demo on Code.org in Figure 1.

We now specify the simulation mechanics. First, we specify a list of "code blocks" which serve as potential "actions" the RL agent can take. These blocks are `MoveForward`, `TurnLeft`, `TurnRight`, `WhileClearAhead`, `End` and `Run`. These are the foundational elements of the visual programming language of the game. The agent must generate a list of blocks ending in `Run` for the bird agent to execute. (One can analogize the `Run` block to an end token in an NLP sentence generation task.) For example, if the agent generates the list of blocks `[MoveForward, MoveForward, MoveForward, Run]`, the bird agent will move forward three times. Once the bird agent has executed the program list, the RL agent receives a reward proportional to +1 if the bird is at the same position as the pig. In any other case, including if the bird goes off the bounds of the grid or runs into a wall, the agent receives a reward proportional to -1.

Of special note are the `WhileClearAhead` and `End` blocks. If the `WhileClearAhead` block appears in the program list, it must be followed by an `End` block. Essentially, any blocks between the `WhileClearAhead` and `End` blocks will be repeatedly executed until the object in front of the bird agent is a wall. If there is a syntactic error (e.g., a `WhileClearAhead` block without a `End` block) or a detectable runtime error (e.g., a `WhileClearAhead` block and a `End` block immediately after with none in between, resulting in an infinite loop) in the program list, the bird agent automatically "dies" resulting in a reward proportional to -1.

4. Experiments and Results

We ran several different experiments in order to characterize our model and see how the design decisions that we made affected performance of the model. Experiments were run in the OpenAI Gym wrapper for the PyGame Learning Environment on 2 Titan Xp GPUs, and all models were implemented using TensorFlow.

4.1. RL Baseline

The first experiment that we ran had no imagination loss and no curriculum. We ran a vanilla implementation of GA3C and evaluated the performance of the RL agent on our nondeterministic environment that changes agent orientation and world state. These results can be seen in Figure 6.

The agent did not perform well on the task as the generalization proved too difficult for it. When the environment changed, the return went down and the agent was not able to easily recover from the changes to the world state. Due to its inability to filter out signal from the noise, it was not able to learn an accurate representation of the goal or the meaning of the different code blocks.

4.2. Imagination Without Curriculum

We then ran the same GA3C agent with imagination loss. The results can be seen in Figure 7.

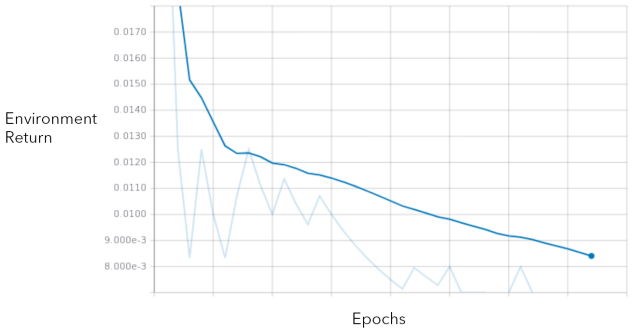


Figure 6. Environment return over time for the RL baseline on the generalized task (i.e., no use of curriculum and no imagination).

Figure 7. Environment Return over time for the imagination agent without curriculum.

It is clear that the agent was learning and was converging better than the baseline agent. Its learning was slow at the beginning but relatively stable throughout training. You can see the first of the environment changes at the far right of the graph. When the environment changes, the environment return decreases, although the drop is not too drastic. From this point on, environment returns appear to be relatively stable, indicating that imagination helps keep the rewards stable.

4.3. Curriculum and imagination

Our last experiment was the full proposed model with imagination loss and curriculum. The results are visualized in Fig 8.

This model is the best performing of the three, by far. The agent quickly learns from the static map and then progresses to new, harder tasks involving more movement and turning. Once it gets to the harder environments, it will encounter ones that cause its environmental return to momentarily decrease, however, the model was able to quickly recover from those mistakes and bring its return back up, showing good generalization ability. We also see that the expected environmental returns are consistently higher in magnitude than for the previous two agents.

5. Conclusion

We show that an RL agent is able to perform a visual programming task when trained according to a curriculum of increasingly generalized and diverse subtasks. In addition, the presence of "imagination" in the RL agent stabilizes training and allows the agent to "plan ahead" when selecting actions, similar to how a human programmer might think about future methods that will need to interact with a current piece of code. The presence of visual feedback was an essential part of the model itself, allowing the agent to observe the impact of its choices on the environment and/or agents within the aforementioned environment.

The larger point is that neural code synthesis need not be a narrowly defined problem, just as coding has varying levels of difficulty in real life. We have shown that an RL agent is able to accomplish a programming task typically performed by programming students early on in their educational careers. We hope that this work shows that RL is a promising framework with which to approach at least some subset of code synthesis problems. For example, if the problem involves choosing a certain subroutine out of many in order to accomplish a certain goal, an imagination-based RL agent trained on a curriculum that includes the nuances of that particular domain should be able to make a relatively good choice,.

5.1. Future Work

Throughout training, we had difficulty getting the agent to apply control structures to its actions. For example, for long corridors, it would make sense to have a loop. Instead the agent kept outputting forward blocks. This led to long sequences and non-human programs. We could add a length penalty to the loss function to weight the loses associated with different outputs according to their lengths. This would encourage the agent bias itself toward shorter solutions. This same idea was used by Abolafia et al. [1], and we imagine that it could also help act as a form of regularization.

Another possible area of improvement is in running backprop through the control structures. How do you deal with the gradient of a while block or and end block? Intuitively, the gradients of the loop body would be accumulated and scaled into the while block. But end blocks are harder because they have no purpose save for bookkeeping. However, implementation of these improvements would likely be based off of conditional execution (eager) which has only recently been implemented in Tensorflow, our framework of choice, and is still a work in progress.

Finally, the most ambitious area of future research would be to examine how we can implement a similar system of feedback with RL on non-visual programming problems. This could potentially involve integrating information from the stack trace or gathering other information about the exe-

Figure 8. Environment Return over time for the imagination agent trained on a curriculum.

cution profile of the generated program to provide the agent with more knowledge.

References

[1] D. A. Abolafia, M. Norouzi, and Q. V. Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

[2] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz. GA3C: gpu-based A3C for deep reinforcement learning. *CoRR*, abs/1611.06256, 2016.

[3] N. Kant. Recent advances in neural program synthesis. *arXiv preprint arXiv:1802.02353*, 2018.

[4] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[6] R. E. Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

[7] D. Purves, B. B. Monson, J. Sundararajan, and W. T. Wojtach. How biological vision succeeds in the physical world. *Proceedings of the National Academy of Sciences*, 111(13):4750–4755, 2014.

[8] P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya, et al. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225*, 2017.

[9] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[10] W. Taube, M. Mouthon, C. Leukel, H.-M. Hoogewoud, J.-M. Annoni, and M. Keller. Brain activity during observation and motor imagery of different balance tasks: an fmri study. *Cortex*, 64:102–114, 2015.

[11] C. Wang, H. Yang, C. Bartz, and C. Meinel. Image captioning with deep bidirectional lstms. In *Proceedings of the 2016 ACM on Multimedia Conference*, pages 988–997. ACM, 2016.

[12] G. Wayne, C. Hung, D. Amos, M. Mirza, A. Ahuja, A. Grabska-Barwinska, J. W. Rae, P. Mirowski, J. Z. Leibo, A. Santoro, M. Gemici, M. Reynolds, T. Harley, J. Abramson, S. Mohamed, D. J. Rezende, D. Saxton, A. Cain, C. Hillier, D. Silver, K. Kavukcuoglu, M. Botvinick, D. Hassabis, and T. P. Lillicrap. Unsupervised predictive memory in a goal-directed agent. *CoRR*, abs/1803.10760, 2018.