The Sensory Neuron as a Transformer: Permutation-Invariant Neural Networks for Reinforcement Learning

Yujin Tang[†]
Google Brain
yujintang@google.com

David Ha[†]
Google Brain
hadavid@google.com

Abstract

In complex systems, we often observe complex global behavior emerge from a collection of agents interacting with each other in their environment, with each individual agent acting only on locally available information, without knowing the full picture. Such systems have inspired development of artificial intelligence algorithms in areas such as swarm optimization and cellular automata. Motivated by the emergence of collective behavior from complex cellular systems, we build systems that feed each sensory input from the environment into distinct, but identical neural networks, each with no fixed relationship with one another. We show that these sensory networks can be trained to integrate information received locally, and through communication via an attention mechanism, can collectively produce a globally coherent policy. Moreover, the system can still perform its task even if the ordering of its inputs is randomly permuted several times during an episode. These permutation invariant systems also display useful robustness and generalization properties that are broadly applicable. Interactive demo and videos of our results: https://attentionneuron.github.io/

1 Introduction

Sensory substitution refers to the brain's ability to use one sensory modality (e.g., touch) to supply environmental information normally gathered by another sense (e.g., vision). Numerous studies have demonstrated that humans can adapt to changes in sensory inputs, even when they are fed into the *wrong* channels [4, 5, 24, 62]. But difficult adaptations–such as learning to "see" by interpreting visual information emitted from a grid of electrodes placed on one's tongue [5], or learning to ride a "backwards" bicycle [62]–require months of training to achieve mastery. Can we do better, and create artificial systems that can rapidly adapt to sensory substitutions, without the need to be retrained?

Brain can adapt to some changes to sensory inputs. Can AI do better?

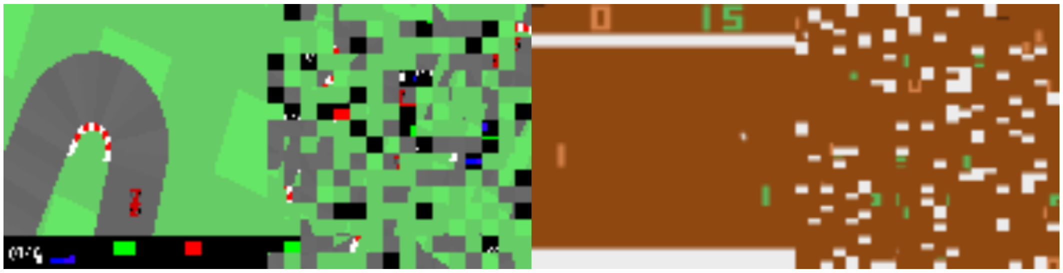


Figure 1: *Comparison of visual input intended for the game player, and what our system receives.* We partition the visual input from CarRacing (Left) and Atari Pong (right) into a 2D grid of small patches, and randomly permute their ordering. Each sensory neuron in the system receives a stream of visual input at a particular permuted patch location, and through coordination, must complete the task at hand, even if the visual ordering is randomly permuted again several times during an episode.

[†] Equal Contribution

Modern deep learning systems are generally unable to adapt to a sudden reordering of sensory inputs, unless the model is retrained, or if the user manually corrects the ordering of the inputs for the model. However, techniques from continual meta-learning, such as adaptive weights [2, 34, 63], Hebbian-learning [50, 51, 55], and model-based [1, 19, 35, 36] approaches can help the model adapt to such changes, and remain a promising active area of research.

In this work, we investigate agents that are explicitly designed to deal with sudden random reordering of their sensory inputs while performing a task. Motivated by recent developments in self-organizing neural networks [26, 53, 59] related to cellular automata [13, 16, 17, 56, 76], in our experiments, we feed each sensory input (which could be an individual state from a continuous control environment, or a patch of pixels from visual environments) into an individual neural network module that integrates information from only this particular sensory input channel over time. While receiving information locally, each of these individual sensory neural network modules also continually broadcasts an output message. Inspired by the Set Transformer [46, 72] architecture, an attention mechanism combines these messages to form a global latent code which is then converted into the agent's action space. The attention mechanism can be viewed as a form of adaptive weights of a neural network, and in this context, allows for an arbitrary number of sensory inputs that can be processed in any random order.

In our experiments, we find that each individual sensory neural network module, despite receiving only localized information, can still collectively produce a globally coherent policy, and that such a system can be trained to perform tasks in several popular reinforcement learning (RL) environments. Furthermore, our system is able to utilize a varying number of sensory input channels in any randomly permuted order, even when the order is shuffled again several times during an episode.

Permutation invariant systems have several advantages over traditional fixed-input systems. We find that encouraging a system to learn a coherent representation of a permutation invariant observation space leads to policies that are more robust and generalizes better to unseen situations. We show that, without additional training, our system continues to function even when we inject additional input channels containing noise or redundant information. In visual environments, we show that our system can be trained to perform a task even if it is given only a small fraction of randomly chosen patches from the screen, and at test time, if given more patches, the system can take advantage of the additional information to perform better. We also demonstrate that our system is able to generalize to visual environments with different background images, despite training on a single fixed background. Lastly, to make training more practical, we propose a behavioral cloning scheme to convert policies trained with existing methods into a permutation invariant policy with desirable properties.

2 Related Work

Self-organization is a process where some form of global order emerges from local interactions between parts of an initially disordered system. It is also a property observed in cellular automata (CA) [16, 17, 56], which are mathematical systems consisting of a grid of cells that perform computation by having each cell communicate with its immediate neighbors and performing a local computation to update its internal state. Such local interactions are useful in modeling complex systems [76] and have been applied to model non-linear dynamics in various fields [13]. Cellular Neural Networks [15] were first introduced in the 1980s to use neural networks in place of the algorithmic cells in CA systems. They were applied to perform image processing operations with parallel computation. Eventually, the concept of self-organizing neural networks found its way into deep learning in the form of Graph Neural Networks (GNN) [60, 77].

Using modern deep learning tools, recent work demonstrate that *neural CA*, or self-organized neural networks performing only local computation, can generate (and re-generate) coherent images [53] and voxel scenes [68, 82], and even perform image classification [59]. Self-organizing neural network agents have been proposed in the RL domain [10, 11, 57, 58], with recent work demonstrating that shared local policies at the actuator level [41], through communicating with their immediate neighbors, can learn a global coherent policy for continuous control locomotion tasks. While existing CA-based approaches present a modular, self-organized solution, they are *not* inherently permutation invariant. In our work, we build on neural CA, and enable each cell to communicate beyond its immediate neighbors via an attention mechanism that enables permutation invariance.

Meta-learning recurrent neural networks (RNN) [22, 38, 40, 74] have been proposed to approach the problem of learning the learning rules for a neural network using the reward or error signal, enabling meta-learners to learn to solve problems presented outside of their original training domains.

The goals are to enable agents to continually learn from their environments in a single lifetime episode, and to achieve much better data efficiency than conventional learning methods such as stochastic gradient descent (SGD). A meta-learned policy that can adapt the weights of a neural network to its inputs during inference time have been proposed in fast weights [63, 65], associative weights [2], hypernetworks [34], and Hebbian-learning [50, 51] approaches. Recently works [44, 61] combine ideas of self-organization with meta-learning RNNs, and have demonstrated that modular meta-learning RNN systems not only can learn to perform SGD-like learning rules, but can also discover more general learning rules that transfer to classification tasks on unseen datasets.

In contrast, the system presented here do not use an error or reward signal to meta-learn or fine-tune its policy. But rather, by using the shared modular building blocks from the meta-learning literature, we focus on learning or converting an existing policy to one that is permutation invariant, and we examine the characteristics such policies exhibit in a zero-shot setting, *without* additional training.

Attention can be viewed as an adaptive weight mechanism that alters the weight connections of a neural network layer based on what the inputs are. Linear *dot-product* attention have first been proposed for meta-learning [64], and versions of linear attention with $softmax$ non-linearity appeared later [32, 49], now made popular with Transformer [72]. The adaptive nature of attention provided the Transformer with a high degree of expressiveness, enabling it to learn inductive biases from large datasets and have been incorporated into state-of-the-art methods in natural language processing [8, 20], image recognition [21] and generation [25], audio and video domains [30, 42, 69].

Attention mechanisms has found many uses for RL [12, 54, 66, 70, 81]. Our work here specifically uses attention to enable communication between arbitrary number of modules in an RL agent. While previous work [31, 43, 52, 73, 79, 83] utilized attention as a communication mechanism between independent neural network modules of a GNN, our work focuses on studying the permutation invariant properties of attention-based communication applied to RL agents. Related work [48] used permutation invariant critics to improve performance of multi-agent RL. Building on Deep Sets [80], Set Transformers [46] investigated the use of attention explicitly for permutation invariant problems that deal with set-structured data, which have provided the theoretical foundation for our work.

3 Method

3.1 Background

Our goal is to devise an agent that is permutation invariant (PI) in the action space to the permutations in the input space. While it is possible to acquire a quasi-PI agent by training with randomly shuffled observations and hope the agent's policy network has enough capacity to memorize all the patterns, we aim for a design that achieves true PI even if the agent is trained with fix-ordered observations. Mathematically, we are looking for a non-trivial function $f(x) : \mathcal{R}^n \mapsto \mathcal{R}^m$ such that $f(x[s]) = f(x)$ for any $x \in \mathcal{R}^n$, and s is any permutation of the indices $\{1, \cdots, n\}$. A different but closely related concept is permutation equivariance (PE) which can be described by a function $h(x) : \mathcal{R}^n \mapsto \mathcal{R}^n$ such that $h(x[s]) = h(x)[s]$. Unlike PI, the dimensions of the input and the output must equal in PE.

Self-attentions can be PE. In its simplest form, self-attention is described as $y = \sigma(QK^\top)V$ where $Q, K \in \mathcal{R}^{n \times d_q}, V \in \mathcal{R}^{n \times d_v}$ are the Query, Key and Value matrices and $\sigma(\cdot)$ is a non-linear function. In most scenarios, Q, K, V are functions of the input $x \in \mathcal{R}^n$ (e.g. linear transformations), permuting x therefore is equivalent to permuting the rows in Q, K, V and based on its definition it is straightforward to verify the PE property. Set Transformer [46] cleverly replaced Q with a set of learnable seed vectors, so it is no longer a function of input x, thus enabling the output to become PI. A simple, intuitive explanation of the PI property of self-attention is available in Appendix A.1.

3.2 Sensory Neurons with Attention

To create PI agents, we propose to add an extra layer in front of the agent's policy network π, which accepts the current observation o_t and the previous action a_{t-1} as its inputs. We call this new layer AttentionNeuron, and Figure 2 gives an overview of our method. Inside AttentionNeuron, we model the observation o_t as an arbitrarily ordered, variable-length list of sensory inputs, each of which is passed into its own *sensory neuron*, a neural network module. Each sensory neuron only has partial access to the agent's observation, at time t, the ith neuron can see only the ith component of the observation $o_t[i]$. Combined with the previous action a_{t-1}, each sensory neuron computes messages $f_k(o_t[i], a_{t-1})$ and $f_v(o_t[i])$ that are broadcast to the rest of the system. We then use attention to aggregate these messages into a *global latent code*, m_t, that is PI with respect to the inputs.

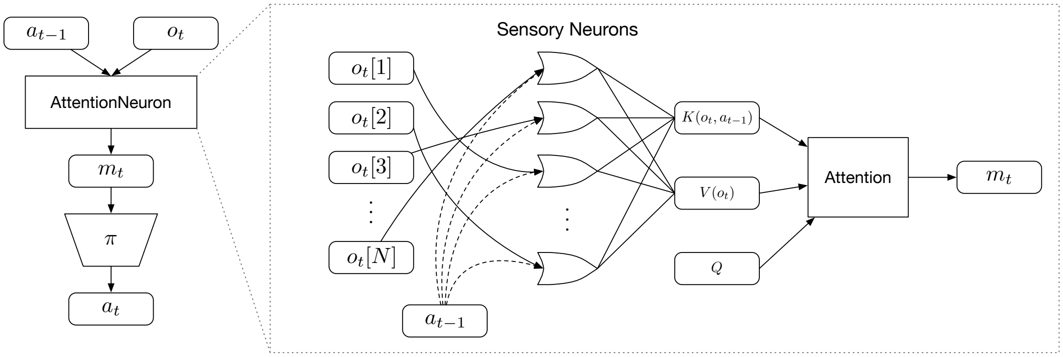


Figure 2: *Overview of Method.* AttentionNeuron is a standalone layer, in which each sensory neuron only has access to a part of the unordered observations o_t. Together with the agent's previous action a_{t-1}, each neuron generates messages independently using the shared functions $f_k(o_t[i], a_{t-1})$ and $f_v(o_t[i])$. The attention mechanism summarizes the messages into a global latent code m_t.

The operations inside AttentionNeuron can be described by the following two equations. For clarity, Table 1 summarizes the notations as well as the corresponding setups we used for the experiments.

dot-product attention with fixed queries

$$K(o_t, a_{t-1}) = \begin{bmatrix} f_k(o_t[1], a_{t-1}) \\ \cdots \\ f_k(o_t[N], a_{t-1}) \end{bmatrix} \in \mathcal{R}^{N \times d_{f_k}}, V(o_t) = \begin{bmatrix} f_v(o_t[1]) \\ \cdots \\ f_v(o_t[N]) \end{bmatrix} \in \mathcal{R}^{N \times d_{f_v}} \quad (1)$$

$$m_t = \sigma\Big(\frac{[QW_q][K(o_t, a_{t-1})W_k]^\top}{\sqrt{d_q}}\Big)[V(o_t)W_v] \quad (2)$$

Equation 1 shows how each of the N sensory neuron independently generates its messages f_k and f_v, which are functions shared across all sensory neurons. Equation 2 shows the attention mechanism aggregate these messages. Note that although we could have absorbed the projection matrices W_q, W_k, W_v into Q, K, V, we keep them in the equation to show explicitly the formulation. Equation 2 is almost identical to the simple definition of self-attention mentioned earlier. Following [46], we make our Q matrix a bank of fixed embeddings, rather than depend on the observation o_t.

Note that permuting the observations only affects the row orders of K and V, and that applying the same permutation to the rows of both K and V still results in the same m_t which is PI. As long as we set constant the number of rows in Q, the change in the input size affects only the number of rows in K and V and does not affect the output m_t. In other words, our agent can accept inputs of arbitrary length and output a fixed sized m_t. Later, we apply this flexibility of input dimensions to RL agents.

Table 1: In this notation list, we provide the dimensions used in our model for different RL environments, to give the reader a sense of the relative magnitudes involved in each part of the system.

Description	Notation	CartPole	Ant	CarRacing	Atari Pong
Full observation space	o_t	$\mathcal{R}^5$	$\mathcal{R}^{28}$	$\mathcal{R}^{96\times96\times4}$	$\mathcal{R}^{84\times84\times4}$
Individual sensory input space	$o_t[i]$	$\mathcal{R}^1$	$\mathcal{R}^1$	$\mathcal{R}^{6\times6\times4=144}$	$\mathcal{R}^{6\times6\times4=144}$
Number of sensory neurons	N	5	28	$(96/6)^2 = 256$	$(84/6)^2 = 196$
Dimension of action space	$\|A\|$	1	8	3	6 (one-hot)
Number of embeddings in Q	M	16	32	1024	400
Projection matrix for Q	W_q	$\mathcal{R}^{8\times32}$	$\mathcal{R}^{8\times32}$	$\mathcal{R}^{8\times16}$	$\mathcal{R}^{8\times32}$
Projection matrix for K	W_k	$\mathcal{R}^{8\times32}$	$\mathcal{R}^{8\times32}$	$\mathcal{R}^{111\times16}$	$\mathcal{R}^{114\times32}$
Projection matrix for V	W_v	I	I	$\mathcal{R}^{144\times16}$	$\mathcal{R}^{144\times32}$
Post-attention activation function	$\sigma(\cdot)$	$tanh$	$tanh$	$softmax$	$softmax$
Global latent code	m_t	$\mathcal{R}^{16}$	$\mathcal{R}^{32}$	$\mathcal{R}^{1024\times16}$	$\mathcal{R}^{400\times32}$

Post attention activation function is not always softmax

3.3 Design Choices

It is worthwhile to have a discussion on the design choices made. Since the ordering of the input is arbitrary, each sensory neuron is required to interpret and identify their received signal. To achieve this, we want $f_k(o_t[i], a_{t-1})$ to have temporal memories. In practice, we find both RNNs and feed-forward neural networks (FNN) with stacked observations work well, with FNNs being more practical for environments with high dimensional observations.

Each sensory neuron gets the history (stacked frames)

In addition to the temporal memory, including previous actions is important for the input identification too. Although the former allows the neurons to infer the input signals based on the characteristics of the temporal stream, this may not be sufficient. For example, when controlling a legged robot, most of the sensor readings are joint angles and velocities from the legs, which are not only numerically identically bounded but also change in similar patterns. The inclusion of previous actions gives each sensory neuron a chance to infer the casual relationship between the input channel and the applied actions, which helps with the input identification.

Finally, in Equation 2 we could have combined $QW_q \in \mathcal{R}^{M \times d_q}$ as a single learnable parameters matrix, but we separate them for two reasons. First, by factoring into two matrices, we can reduce the number of learnable parameters. Second, we find that instead of making Q learnable, using the positional encoding proposed in Transformer [72] encourages the attention mechanism to generate distinct codes. Here we use the row indices in Q as the positions for encoding.

4 Experiments

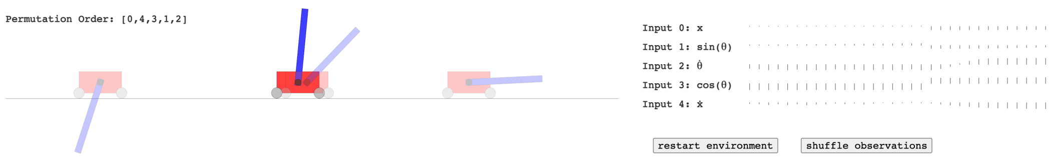
We conduct experiments on several different RL environments to study various properties of permutation invariant RL agents. Due to the nature of the underlying tasks, we will describe the different architectures of the policy networks used and discuss different training methods. However, the AttentionNeuron layers in all agents are similar, so we first describe the common setups. Hyper-parameters and other details for all experiments are summarized in Appendix A.4.

For non-vision continuous control tasks, the agent receives an observation vector $o_t \in \mathcal{R}^{|O|}$ at time t. We assign $N = |O|$ sensory neurons for the tasks, each of which sees one element from the vector, hence $o_t[i] \in \mathcal{R}^1, i = 1, \cdots, |O|$. We use an LSTM [39] as our $f_k(o_t[i], a_{t-1})$ to generate Keys, the input size of which is $1 + |A|$ (2 for Cart-Pole and 9 for PyBullet Ant). A simple pass-through function $f(x) = x$ serves as our $f_v(o_t[i])$, and $\sigma(\cdot)$ is $tanh$. For simplicity, we find $W_v = I$ works well for the tasks, so the learnable components are the LSTM, W_q and W_k.

For vision based tasks, we gray-scale and stack $k = 4$ consecutive RGB frames from the environment, and thus our agent observes $o_t \in \mathcal{R}^{H \times W \times k}$. o_t is split into non-overlapping patches of size $P = 6$ using a sliding window, so each sensory neuron observes $o_t[i] \in \mathcal{R}^{6 \times 6 \times k}$. Here, $f_v(o_t[i])$ flattens the data and returns it, hence $V(o_t)$ returns a tensor of shape $N \times d_{f_v} = N \times (6 \times 6 \times 4) = N \times 144$. Due to the high dimensionality for vision tasks, we do not use RNNs for f_k, but instead use a simpler method to process each sensory input. $f_k(o_t[i], a_{t-1})$ takes the difference between consecutive frames ($o_t[i]$), then flattens the result, appends a_{t-1}, and returns the concatenated vector. $K(o_t, a_{t-1})$ thus gives a tensor of shape $N \times d_{f_k} = N \times [(6 \times 6 \times 3) + |A|] = N \times (108 + |A|)$ (111 for CarRacing and 114 for Atari Pong). We use $softmax$ as the non-linear activation function $\sigma(\cdot)$, and we apply layer normalization [3] to both the input patches and the output latent code.

4.1 Cart-pole swing up

We examine Cart-pole swing up [28, 29, 33, 84] to first illustrate our method, and also use it to provide a clear analysis of the attention mechanism. We use `CartPoleSwingUpHarder` [27], a more difficult version of the task where the initial positions and velocities are highly randomized, leading to a higher variance of task scenarios. In the environment, the agent observes $[x, \dot{x}, cos(\theta), sin(\theta), \dot{\theta}]$, outputs a scalar action, and is rewarded at each step for getting x close to 0 and $cos(\theta)$ close to 1.



They have a web demo

Figure 3: *Interactive demo of CartPoleSwingUpHarder.* In our web demo, the user can shuffle the order of the 5 inputs at any time, and observe how the agent adapts to the new ordering of the inputs.

We use a two-layer neural network as our agent. The first layer is an AttentionNeuron layer with $N = 5$ sensory neurons and outputs $m_t \in \mathcal{R}^{16}$. A linear layer takes m_t as input and outputs a scalar action. For comparison, we also trained an agent with a two-layer FNN policy with 16 hidden units. We use direct policy search to train agents with CMA-ES [37], an evolution strategies (ES) method.

Table 2: *Cart-pole Tests.* For each experiment, we report the average score and the standard deviation from 1000 test episodes. Our agent is trained only in the environment with 5 sensory inputs.

	5 obs	5 obs (shuffled)	10 obs	5 obs + 5 noise
FNN (trained with 5 obs)	593 ± 433	38 ± 120	N/A	N/A
FNN (trained with 10 obs)	N/A	N/A	593 ± 433	137 ± 242
Ours (trained with 5 obs)	472 ± 426	471 ± 426	471 ± 425	461 ± 410

Our agent is able to perform the task and balance the cart-pole from an initially random state. Its average score is slightly lower than the baseline (See column 1 of Table 2) because each sensory neuron requires some time steps in each episode to interpret the sensory input signal it receives. However, as a trade-off for the performance sacrifice, our agent is able to maintain its performance when the input sensor array is randomly shuffled, which is not the case for an FNN policy (column 2). Moreover, although our agent is only trained in an environment with five inputs, it can accept an arbitrary number of inputs in any order without re-training[2]. We test our agent by duplicating the 5 inputs to give the agent 10 observations (column 3). When we replace the 5 extra signals with white noises with $\sigma = 0.1$ (column 4), we do not see a significant drop in performance.

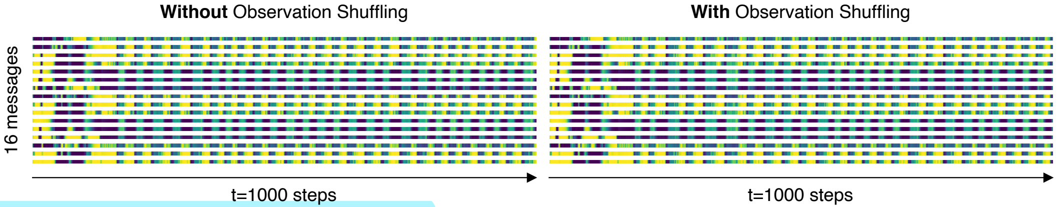


Figure 4: *Permutation invariant outputs.* The output (16-dimensional global latent code) from the AttentionNeuron layer does not change when we input the sensor array as-is (left) or when we randomly shuffle the array (right). Yellow represents higher values, and blue for lower values.

The AttentionNeuron layer should possess 2 properties to achieve these: its output is permutation invariant to its input, and its output carries task-relevant information. Figure 4 is a visual confirmation of the permutation invariant property, whereby we plot the output messages from the layer and their changes over time from two tests. Using same environment seed, we keep the observation as-is in the first test but we shuffle the order in the second. As the figure shows, the output messages are identical in the two roll-outs. We also perform a simple linear regression analysis on the outputs (based on the shuffled inputs) to recover the 5 inputs in their original order. Table 3 shows the R^2 values[3] from this analysis, suggesting that some important indicators (e.g. $\dot{x}$ and $\dot{\theta}$) are well represented in the output.

Table 3: *Linear regression analysis on the output.* For each of the $N = 5$ sensory inputs we have one linear regression model with $m_t \in \mathcal{R}^{16}$ as the explanatory variables.

	x	$\dot{x}$	$cos(\theta)$	$sin(\theta)$	$\dot{\theta}$
R^2	0.354	0.620	0.626	0.233	0.550

Table 4: *PyBullet Ant results.*

	Score
FNN (teacher)	2700 ± 28
FNN (shuffled)	232 ± 112
Ours (ES, shuffled)	2576 ± 75
Ours (BC, shuffled)	2034 ± 948
Ours (BC, shuffled, larger)	2579 ± 457

4.2 PyBullet Ant

While direct policy search methods such as evolution strategies (ES) can train permutation invariant RL agents, often times we already have access to pre-trained agents or recorded human data performing the task at hand. Behavior cloning (BC) can allow us to convert an existing policy to a version that is permutation invariant with desirable properties associated with it.

In Table 4, we train a standard two-layer FNN policy to perform `AntBulletEnv-v0`, a 3D locomotion task in PyBullet [18], and use it as a teacher for BC. For comparison, we also train a two-layer agent with AttentionNeuron for its first layer. Both networks are trained with ES. Similar to CartPole, we expect to see a small performance drop due to some time steps required for the agent to interpret an arbitrarily ordered observation space. We then collect data from the FNN teacher policy to train permutation invariant agents using BC. More details of the BC setup can be found in Appendix A.4.2.

[2] Because our agent was not trained with normalization layers, we scaled the output from the AttentionNeuron layer by 0.5 to account for the extra inputs in the last 2 experiments.

[3] R^2 measures the goodness-of-fit of a model. An R^2 of 1 implies that the regression perfectly fits the data.

The performance of the BC agent is lower than the one trained from scratch with ES, despite having the identical architecture. This suggests that the inductive bias that comes with permutation invariance may not match the original teacher network, so the small model used here may not be expressive enough to clone any teacher policy, resulting in a larger variance in performance. A benefit of gradient-based BC, compared to RL, is that we can easily train larger networks to fit the behavioral data. We show that increasing the size of the subsequent layers for BC does increase the performance.

As we will demonstrate next, BC is a useful technique for training permutation invariant agents in environments with high dimensional visual observations that may require larger networks.

4.3 Atari Pong

Here, we are interested in solving screen-shuffled versions of vision-based RL environments, where each observation frame is divided up into a grid of patches, and like a puzzle, the agent must process the patches in a shuffled order to determine a course of action to take. A shuffled version of Atari Pong [7] (See Figure 1, right pair) can be especially hard for humans to play when inductive biases from human priors [23] that expect a certain type of spatial structure is missing from the observations.

But rather than throwing away the spatial structure entirely from our solution, we find that convolution neural network (CNN) policies work better than fully connected multi-layer perceptron (MLP) policies when trained with behavior cloning for Atari Pong. In this experiment, we reshape the output m_t of the AttentionNeuron layer from $\mathcal{R}^{400\times32}$ to $\mathcal{R}^{20\times20\times32}$, a 2D grid of latent codes, and pass this 2D grid into a CNN policy. This way, the role of the AttentionNeuron layer is to take a list of unordered observation patches, and learn to construct a 2D grid representation of the inputs to be used by a downstream policy that expects some form of spatial structure in the codes. Our permutation invariant policy trained with BC is able to consistently reach a perfect score of 21, even with shuffled screens. The details of the CNN policy and BC training can be found in Appendix A.4.3.

AttentionNeuron can construct a 2D grid from the unordered inputs

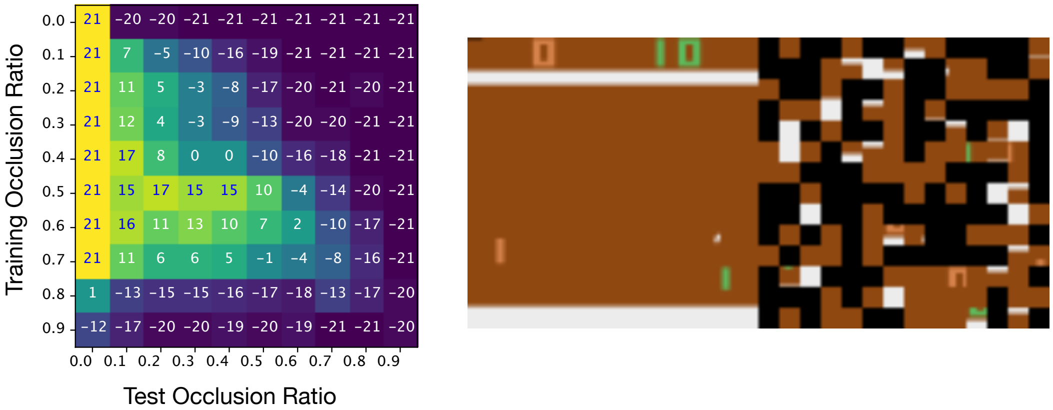


Figure 5: *Mean test scores in Atari Pong, and example of a randomly-shuffled occluded observation.* In the heat map, each value is the average score from 100 test episodes. For comparison, we show the original screen (left), and the agent's observation (right). Discarded patches shown here in black.

Unlike typical CNN policies, our agent can accept a subset of the screen, since the agent's input is a variable-length list of patches. It would thus be interesting to deliberately randomly discard a certain percentage of the patches and see how the agent reacts. The net effect of this experiment for humans is similar to being asked to play a partially occluded and shuffled version of Atari Pong (see Figure 5, right). During training via BC, we randomly remove a percentage of observation patches. In tests, we fix the randomly selected positions of patches to discard during an entire episode.

We present the results in a heat map in Figure 5 (left), where the y-axis shows the patches removed during training and the x-axis gives the patch occlusion ratio in tests. The diagram shows clear patterns for interpretation. Looking horizontally along each row, the performance drops because the agent sees less of the screen which increases the difficulty. Interestingly, an agent trained at a high occlusion rate of 80% rarely wins against the Atari opponent, but once it is presented with the full set of patches during tests, it is able to achieve a fair result by making use of the additional information.

To gain insights into understanding the policy, we projected the AttentionNeuron layer's output in a test roll-out to 2D space using t-SNE [71]. In Figure 6, we highlight several groups and show their corresponding inputs. The AttentinNeuron layer clearly learned to cluster inputs that share similar features. For example, the 3 sampled inputs in the blue group show the situation when the agent's paddle moved toward the bottom of the screen and stayed there. Similarly, the orange group show the cases when the ball was not in sight, this happened right before/after a game started/ended. We believe these discriminative outputs enabled the downstream policy to accomplish the agent's task.

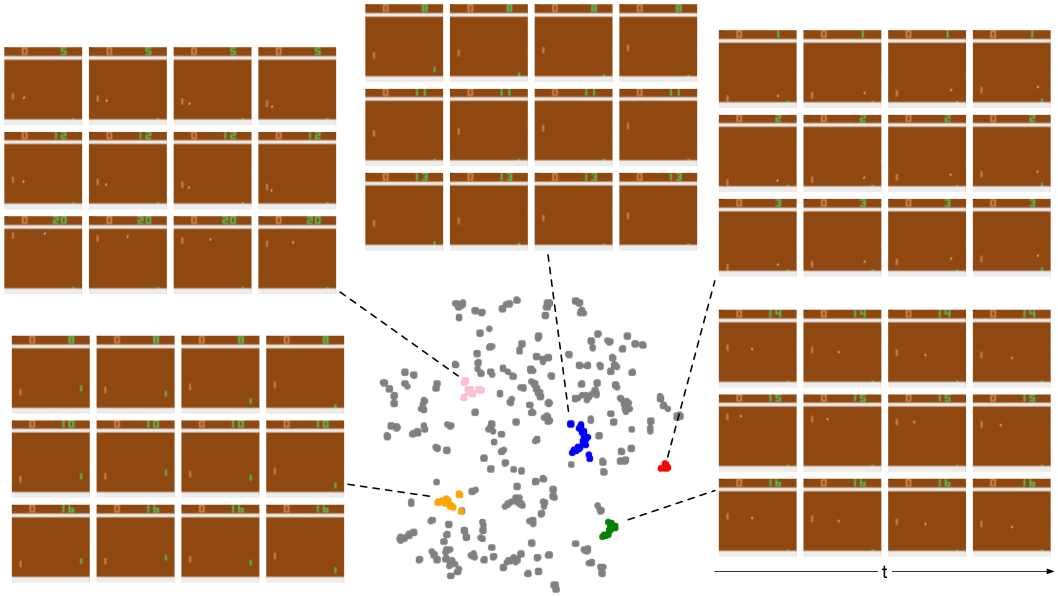


Figure 6: *2D embedding of the AttentionNeuron layer's output in a test episode.* We highlight several representative groups in the plot, and show the sampled inputs from them. For each group, we show 3 corresponding inputs (rows) and unstack each to show the time dimension (columns).

4.4 CarRacing

We find that encouraging an agent to learn a coherent representation of a deliberately shuffled visual scene leads to agents with useful generalization properties. Such agents are still able to perform their task even if the visual background of the environment changes, despite being trained only on a single static background. Out-of-domain generalization is an active area, and here, we combine our method with AttentionAgent [70], a method that uses selective, hard-attention via a patch voting mechanism. AttentionAgents in [70] generalize well to several unseen visual environments where task irrelevant elements are modified, but fails to generalize to drastic background changes in a zero-shot setting.

In this experiment, we combine the permutation invariant AttentionNeuron layer with the policy network used in AttentionAgent. As their hard-attention-based policy is non-differentiable, we train the entire system using ES. We reshape the AttentionNeuron layer's outputs to adapt for the policy network. Specifically, we reshape the output message to $m_t \in \mathcal{R}^{32\times32\times16}$ such that it can be viewed as a 32-by-32 grid of 16 channels. The end result is a policy with two layers of attention: the first layer outputs a latent code book to represent a shuffled scene, and the second layer performs hard attention to select the top $K = 10$ codes from a 2D global latent code book. A detailed description of the selective hard attention policy from [70] and other training details can be found in Appendix A.4.4.

We first train the agent in the CarRacing [45] environment, and report the average score from 100 test roll-outs in Table 5. As the first column shows, our agent's performance in the training environment is slightly lower but comparable to the baseline method, as expected. But because our agent accepts randomly shuffled inputs, it is still able to navigate even when the patches are shuffled. Figure 1 (left pair) gives an illustration, where the right screen is what our agent observes and the left is for human visualization. A human will find driving with the shuffled observation to be very difficult because we are not constantly exposed to such tasks, just like in the "reverse bicycle" example mentioned earlier.

Without additional training or fine-tuning, we test whether the agent can also navigate in four modified environments where the green grass background is replaced with various images (See Figure 7). As Table 5 (from column 2) shows, our agent generalizes well to most of the test environments with only mild performance drops while the baseline method fails to generalize. We suspect this is because

Table 5: CarRacing Test Results.

	Training Env	KOF	Mt. Fuji	Ukiyoe	DS
AttentionAgent [70]	901 ± 54	-81 ± 4	-57 ± 38	-107 ± 50	-56 ± 23
NetRand [47]	480 ± 144	20 ± 84	356 ± 159	533 ± 111	-27 ± 34
NetRand + AttentionAgent	885 ± 64	-51 ± 14	709 ± 94	656 ± 131	122 ± 134
Ours	801 ± 147	646 ± 189	503 ± 152	661 ± 140	171 ± 146

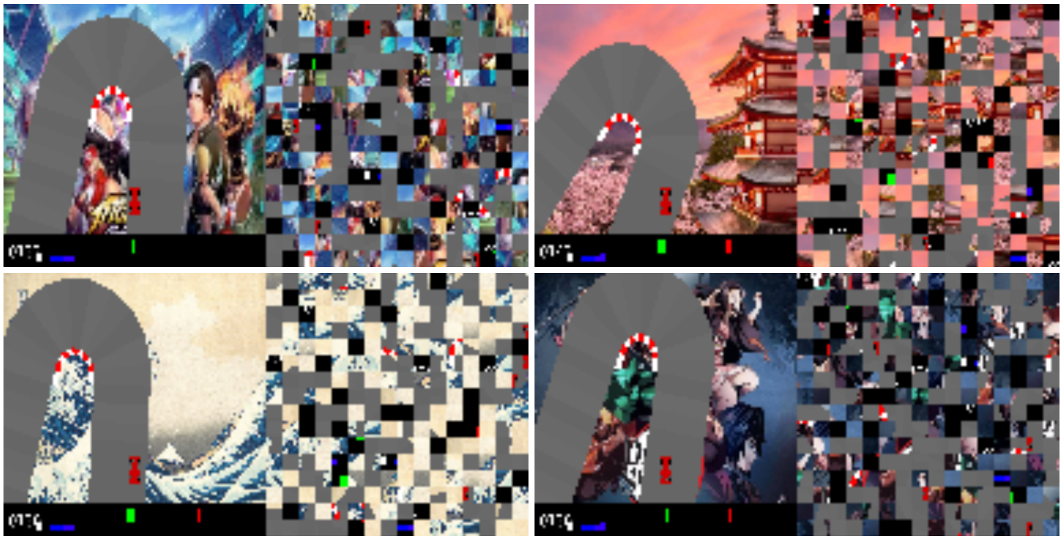


Figure 7: *Screenshots of test environments.* In each pair of images, the left is for human visualization and the right is what our agent sees. From the top left and in the clockwise order, the environments are “KOF”, “Mt. Fuji”, “DS” and “Ukiyoe”.

the AttentionNeuron layer has transformed the original RGB space to a useful hidden representation (represented by m_t) that has eliminated task irrelevant information after observing and reasoning about the sequences of (o_t, a_{t-1}) during training, enabling the downstream hard attention policy to work with an optimized abstract representation tailored for the policy, instead of raw RGB patches.

We also compare our method to NetRand [47], a simple but effective technique developed to perform similar generalization tasks. In the second row of Table 5 are the results of training NetRand on the base CarRacing task. The CarRacing task proved to be too difficult for NetRand, but despite a low performance score of 480 in the training environment, the agent generalizes well to the “Mt. Fuji” and “Ukiyoe” modifications. In order to achieve a meaningful comparison, we combined NetRand with AttentionAgent so that it can get close to a mean score of 900 on the base task. To do that, we used NetRand as an input layer to the AttentionAgent policy network, and trained the combination end-to-end using ES, which is consistent with our proposed method for this task. The combination achieved a respectable mean score of 885, and as we can see in the third row of the above table, this approach also generalizes to a few of the unseen modifications of the CarRacing environment.

Our score on the base CarRacing task is lower than NetRand, but this is expected since our agent requires some amount of time steps to identify each of the inputs (which could be shuffled), while the NetRand and AttentionAgent agent will simply fail on the shuffled versions of CarRacing. Despite this, our method still compares favorably on the generalization performance.

We visualize the attentions from the AttentionNeuron layer in Figure 8. In CarRacing, the agent has learned to focus its attention (indicated by the highlighted patches) on the road boundaries which are intuitive to human beings and are critical to the task. Notice that the attended positions are consistent before and after the shuffling. More details about this visualization can be found in Appendix A.4.4.

5 Discussion and Future Work

In this work, we investigate the properties of RL agents that can treat their observations as an arbitrarily ordered, variable-length list of sensory inputs. By processing each input stream independently, and consolidating the processed information using attention, our agents can still perform their tasks even if the ordering of the observations is randomly permuted several times during an episode, without explicitly trained for frequent re-shuffling (See Table 6).

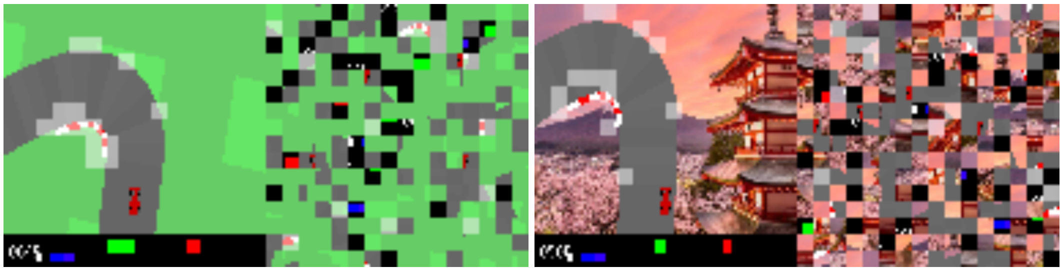


Figure 8: *Attention visualization.* We highlight the observed patches that receive the most attention. Left: Training environment. Right: Test environment with unseen background.

Table 6: *Reshuffle observations during a roll-out.* In each test episode, we reshuffle the observations every t steps. For CartPole, we test for 1000 episodes because of its larger task variance. For the other tasks, we report mean and standard deviation from 100 tests. All environments except for Atari Pong have a hard limit of 1000 time steps per episode. In Atari Pong, while the maximum length of an episode does not exist, we observed that an episode usually lasts for around 2500 time steps.

	CartPole	PyBullet Ant	Atari Pong	CarRacing
$t = 25$	107 ± 146	2053 ± 225	-20 ± 1	732 ± 161
$t = 50$	163 ± 198	2319 ± 188	-20 ± 2	772 ± 163
$t = 100$	242 ± 254	2406 ± 178	-10 ± 12	768 ± 167
$t = 200$	318 ± 310	2493 ± 105	-2 ± 17	774 ± 182
$t = 500$	407 ± 380	2548 ± 87	18 ± 9	805 ± 158
No reshuffle	472 ± 426	2576 ± 75	21 ± 0	801 ± 147

Applications By presenting the agent with shuffled, and even incomplete observations, we encourage it to interpret the meaning of each local sensory input and how they relate to the global context. This could be useful in many real world applications. For example, such policies could avoid errors due to cross-wiring or complex, dynamic input-output mappings when being deployed in real robots. A similar setup to the CartPole experiment with extra noisy channels could enable a system that receives thousands of noisy input channels to identify the small subset of channels with relevant information.

Limitations For visual environments, patch size selection will affect both performance and computing complexity. We find that patches of 6x6 pixels work well for our tasks, as did 4x4 pixels to some extent, but single pixel observations fail to work. Small patch sizes also results in a large attention matrix which may be too costly to compute, unless approximations are used [14, 75, 78].

Another limitation is that the permutation invariant property apply only to the inputs, and not to the outputs. While the ordering of the observations can be shuffled, the ordering of the actions cannot. For permutation invariant outputs to work, each action will require feedback from the environment, including reward information, in order to learn the relationship between itself and the environment.

Future Work An interesting future direction is to also make the action layer have the same properties, and model each "motor neuron" as a module connected using attention. With such methods, it may be possible to train an agent with an arbitrary number of legs, or control robots with different morphology using a single policy that is also provided with a reward signal as feedback. It is exciting to see future works that include signals such as environmental rewards to train permutation invariant meta-learning agents that can adapt to not only changes in the observed environment, but also to changes to itself.

Societal Impact Like most algorithms proposed in computer science and machine learning, our method can be applied in ways that will have potentially positive or negative impacts to society. While our small-scale, self-contained experiments study only the properties of agents that are PI to their observations, and we believe our results do not directly cause harm to society, the robustness and flexible properties of the method may be of use for data-collection systems that receive data from a large variable number of sensors. For instance, one could apply permutation invariant sensory systems to process data from millions of sensors for anomaly detection, which may result in both positive or negative impacts, if used in applications such as large-scale sensor analysis for weather forecasting, or deployed in large-scale surveillance systems that could undermine our basic freedoms.

Our work also provides a way to view the Transformer [72] through the lens of self-organizing neural networks. Transformers are known to have potentially negative societal impacts highlighted in studies about possible data-leakage and privacy vulnerabilities [9], malicious misuse and issues concerning bias and fairness [6], and energy requirements for training them [67].

Acknowledgements

The authors would like to thank Rishabh Agarwal, Jie Tan, Yingtao Tian, Douglas Eck, Aleksandra Faust for their valuable discussion and feedback. The experiments in this work were conducted using virtual machines provided by Google Cloud Platform.

A Appendix

A.1 Intuitive explanation of Self-Attention's permutation invariant property

Here, we provide a simple, non-rigorous example demonstrating permutation invariant property of the self-attention mechanism, to give some intuition to readers who may not be familiar with self-attention. For a detailed treatment, please refer to Deep Sets [80] and Set Transformer [46].

As mentioned in Section 3.1, in its simplest form, self-attention is described as:

$$y = \sigma(QK^{\top})V \tag{3}$$

where $Q \in \mathcal{R}^{N_q \times d_q}, K \in \mathcal{R}^{N \times d_q}, V \in \mathcal{R}^{N \times d_v}$ are the Query, Key and Value matrices and $\sigma(\cdot)$ is a non-linear function. In this work, Q is a fixed matrix, and K, V are functions of the input $X \in \mathcal{R}^{N \times d_{in}}$ where N is the number of observation components (equivalent to the number of sensory neurons) and d_{in} is the dimension of each component. In most settings, $K = XW_k, V = XW_v$ are linear transformations, thus permuting X therefore is equivalent to permuting the rows in K, V.

We would like to show that the output y is the same regardless of the ordering of the rows of K, V. For simplicity, suppose $N = 3, N_q = 2, d_q = d_v = 1$, so that $Q \in \mathcal{R}^{2\times 1}, K \in \mathcal{R}^{3 \times 1}, V \in \mathcal{R}^{3 \times 1}$:

$$\begin{aligned} y &= \sigma\left(\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \begin{pmatrix} k_1 & k_2 & k_3 \end{pmatrix}\right) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \\ &= \sigma\left(\begin{pmatrix} q_1k_1 & q_1k_2 & q_1k_3 \\ q_2k_1 & q_2k_2 & q_2k_3 \end{pmatrix}\right) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \\ &= \begin{pmatrix} \sigma(q_1k_1)v_1 + \sigma(q_1k_2)v_2 + \sigma(q_1k_3)v_3 \\ \sigma(q_2k_1)v_1 + \sigma(q_2k_2)v_2 + \sigma(q_2k_3)v_3 \end{pmatrix} \end{aligned} \tag{4}$$

The output $y \in \mathcal{R}^{2\times 1}$ remains the same when the rows of K, V are permuted from $[1, 2, 3]$ to $[3, 1, 2]$:

$$\begin{aligned} y &= \sigma\left(\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \begin{pmatrix} k_3 & k_1 & k_2 \end{pmatrix}\right) \begin{pmatrix} v_3 \\ v_1 \\ v_2 \end{pmatrix} \\ &= \sigma\left(\begin{pmatrix} q_1k_3 & q_1k_1 & q_1k_2 \\ q_2k_3 & q_2k_1 & q_2k_2 \end{pmatrix}\right) \begin{pmatrix} v_3 \\ v_1 \\ v_2 \end{pmatrix} \\ &= \begin{pmatrix} \sigma(q_1k_3)v_3 + \sigma(q_1k_1)v_1 + \sigma(q_1k_2)v_2 \\ \sigma(q_2k_3)v_3 + \sigma(q_2k_1)v_1 + \sigma(q_2k_2)v_2 \end{pmatrix} \end{aligned} \tag{5}$$

We have highlighted the same terms with the same color in Equations 4 and 5 to show the results are indeed identical. In general, we have $y_{ij} = \sum_{b=1}^{N} \sigma\big(\sum_{a=1}^{d_q} Q_{ia}K_{ba}\big)V_{bj}$. Permuting the input is equivalent to permuting the indices b (i.e. rows of K and V), which only affects the order of the outer summation and does not affect y_{ij} because summation is a permutation invariant operation. Notice that in the above example and the proof here we have assumed that $\sigma(\cdot)$ is an element-wise operation—a valid assumption since most activation functions satisfy this condition.[4]

As discussed in Section 3.2, this formulation lets us convert an observation signal from the RL environment into a permutation invariant representation y. We can use this representation in place of the actual observation as the input that goes into the downstream policy network of an RL agent.

[4] Applying *softmax* to each row only brings scalar multipliers to each row and the proof still holds.

A.2 Hyper-parameters

Table 1 contains the hyper-parameters used for each experiment. We did not employ exhaustive hyper-parameter tuning, but have simply selected parameters that can appropriately size our models to work with training methods such as evolution strategies, where the number of parameters cannot be too large. As mentioned in the discussion section about the limitations, we tested a small range of patch sizes (1 pixel, 4 pixels, 6 pixels), and we find that a patch size of 6x6 works well across tasks.

A.3 Description of compute infrastructure used to conduct experiments

For all ES results, we train on Google Kubernetes Engines (GKE) with 256 CPUs (N1 series) for each job. The approximate time, including both training and periodic tests, for the jobs are: 3 days (CartPole), 5 days (PyBullet Ant ES) and 10 days (CarRacing). For BC results, we train with Google Computing Engines (GCE) on an instance that has one V100 GPU. The approximate time, including both training and periodic tests, for the jobs are: 5 days (PyBullet Ant BC), 1 day (Atari Pong).

A.4 Detailed setups for the experiments

A.4.1 Training budget

The costs of ES training are summarized in the following table. A maximum of 20K generations is specified in the training, but stopped early if the performance converged. Each generation has $256 \times 16 = 4096$ episode rollouts, where 256 is the population size and 16 is the rollout repetitions. The Pong permutation-invariant (PI) agents were trained using behavior cloning (BC) on a pre-trained PPO policy (which is not PI-capable), with 10M training steps.

Environment	CartPoleSwingUpHarder	PyBullet Ant	Atari Pong	CarRacing
Number of Generations	14,000	12,000	-	4,000

Note that we used the hyper-parameters (e.g., population size, rollout repetitions) that proved to work on a wide range of tasks in past experiences, and did not tune them for each experiment. In other words, these settings were not chosen with sample-efficiency in mind, but rather for learning a working PI-capable policy using distributed computation within a reasonable wall-clock time budget.

We consider two possible approaches when we take sample-efficiency into consideration. In the experiments, we have demonstrated that it is possible to simply use state-of-the-art RL algorithms to learn a non-PI policy, and then use BC to produce a PI version of the policy. The first approach is thus to rely on the conventional RL algorithms to increase sample efficiency, which is a hot and on-going topic in the area. On the other hand, we do think that an interesting future direction is to formulate environments where BC will fail in a PI setting, and that interactions with the environment (in a PI setting) is required to learn a PI policy. For instance, we have demonstrated in PyBullet Ant that the BC method requires the cloned agent to have a much larger number of parameters compared to one trained with RL. This is where an investigation in sample-efficiency improvements in the RL algorithm explicitly in the PI setting may be beneficial.

A.4.2 PyBullet Ant

In the PyBullet Ant experiment, we demonstrated that a pre-trained policy can be converted into a permutation invariant one with behavior cloning (BC). We give detailed task description and experimental setups here. In `AntBulletEnv-v0`, the agent controls an ant robot that has 8 joints ($|A| = 8$), and gets to see an observation vector that has base and joint states as well as foot-ground contact information at each time step (|O|=28). The mission is to make the ant move along a pre-defined straight line as fast as possible. The teacher policy is a 2-layer FNN policy that has 32 hidden units trained with ES. We collected data from 1000 test roll-outs, each of which lasted for 500 steps. During training, we add zero-mean Gaussian noise ($\sigma = 0.03$) to the previous actions. For the student policy, We set up two networks. The first policy is a 2-layered network that has the AttentionNeuron with output size $m_t \in \mathcal{R}^{32}$ as its first layer, followed by a fully-connected (FC) layer. The second, larger policy is similar in architecture, but we added one more FC layer and expanded all hidden size to 128 to increase its expressiveness. We train the students with a batch size of 64, an Adam optimizer of $lr = 0.001$ and we clip the gradient at maximum norm of 0.5.

A.4.3 Atari Pong

In the Atari game Pong, we append a deep CNN to the AttentionNeuron layer in our agent (student policy). To be concrete, we reshape the AttentionNeuron's output message $m_t \in \mathcal{R}^{400 \times 32}$ to $m_t \in \mathcal{R}^{20 \times 20 \times 32}$ and pass it to the trailing CNN: [Conv(in=32, out=64, kernel=4, stride=2), Conv(in=64, out=64, kernel=3, stride=1), FC(in=3136, out=512), FC(in=512, out=6)]. We use $ReLU$ as the activation functions in the CNN. We collect the stacked observations and the corresponding logits output from a pre-trained PPO agent (teacher policy) from 1000 roll-outs, and we minimize the MSE loss between the student policy's output and the teacher policy's logits. The learning rate and norm clip are the same as the previous experiment, but we use a batch size of 256.

For the occluded Pong experiment, we randomly remove a certain percentage of the patches across a training batch of stacked observation patches. In tests, we sample a patch mask to determine the positions to occlude at the beginning of the episode, and apply this mask throughout the episode.

A.4.4 CarRacing

In AttentionAgent [70], the authors observed that the agent generalizes well if it is forced to make decisions based on only a fraction of the available observations. Concretely, [70] proposed to segment the input image into patches and let the patches vote for each other via a modified self-attention mechanism. The agent would then take into consideration only the top $K = 10$ patches that have the most votes and based on the coordinates of which an LSTM controller makes decisions. Because the voting process involves sorting and pruning that are not differentiable, the agent is trained with ES. In their experiments, the authors demonstrated that the agent could navigate well not only in the training environment, but also zero-shot transfer to several modified environments.

We need only to reshape the AttentionNeuron layer's outputs to adapt for AttentionAgent's policy network. Specifically, we reshape the output message $m_t \in \mathcal{R}^{1024 \times 16}$ to $m_t \in \mathcal{R}^{32 \times 32 \times 16}$ such that it can be viewed as a 32-by-32 "image" of 16 channels. Then if we make AttentionAgent's patch segmentation size 1, the original patch voting becomes voting among the m_t's and thus the output fits perfectly into the policy network. Except for this patch size, we kept all hyper-parameters in AttentionAgent unchanged, we also used the same CMA-ES training hyper-parameters.

Although the simple settings above allows our augmented agent to learn to drive and generalize to unseen background changes, we found the car jittered left and right through the courses. We suspect this is because of the frame differential operation in our $f_k(o_t, a_{t-1})$. Specifically, even when the car is on a straight lane, constantly steering left and right allows $f_k(o_t, a_{t-1})$ to capture more meaningful signals related to the changes of the road. To avoid such jittering behavior, we make m_t a rolling average of itself: $m_t = (1 - \alpha)m_t + \alpha m_{t-1}, 0 \leq \alpha \leq 1$. In our implementation $\alpha = g([h_{t-1}, a_{t-1}])$, where h_{t-1} is the hidden state from AttentionAgent's LSTM controller and a_{t-1} is the previous action. $g(\cdot)$ is a 2-layer FNN with 16 hidden units and a $sigmoid$ output layer.

We analyzed the attention matrix in the AttentionNeuron layer and visualized the attended positions. To be concrete, in CarRacing, the Query matrix has 1024 rows. Because we have $16 \times 16 = 256$ patches, the Key matrix has 256 rows, we therefore have an attention matrix of size 1024×256. To plot attended patches, we select from each row in the attention matrix the patch that has the largest value after softmax, this gives us a vector of length 1024. This vector represents the patches each of the 1024 output channels has considered to be the most important. 1024 is larger than the total patch count, however there are duplications (i.e. multiple output channels have mostly focused on the same patches). The unique number turns out to be $10 \sim 20$ at each time step. We emphasize these patches on the observation images to create an animation.

References

[1] B. Amos, I. D. J. Rodriguez, J. Sacks, B. Boots, and J. Z. Kolter. Differentiable mpc for end-to-end planning and control. *arXiv preprint arXiv:1810.13400*, 2018.

[2] J. Ba, G. Hinton, V. Mnih, J. Z. Leibo, and C. Ionescu. Using fast weights to attend to the recent past. *arXiv preprint arXiv:1610.06258*, 2016.

[3] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[4] P. Bach-y Rita, C. C. Collins, F. A. Saunders, B. White, and L. Scadden. Vision substitution by tactile image projection. *Nature*, 221(5184):963–964, 1969.

[5] P. Bach-y Rita and S. W. Kercel. Sensory substitution and the human–machine interface. *Trends in cognitive sciences*, 7(12):541–546, 2003.

[6] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623, 2021.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[9] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, et al. Extracting training data from large language models. *arXiv preprint arXiv:2012.07805*, 2020.

[10] M. Chang, S. Kaushik, S. M. Weinberg, T. Griffiths, and S. Levine. Decentralized reinforcement learning: Global decision-making via local economic transactions. In *International Conference on Machine Learning*, pages 1437–1447. PMLR, 2020.

[11] N. Cheney, R. MacCurdy, J. Clune, and H. Lipson. Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding. *ACM SIGEVOlution*, 7(1):11–23, 2014.

[12] J. Choi, B.-J. Lee, and B.-T. Zhang. Multi-focus attention network for efficient deep reinforcement learning. *arXiv preprint arXiv:1712.04603*, 2017.

[13] B. Chopard and M. Droz. *Cellular automata*, volume 1. Springer, 1998.

[14] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

[15] L. O. Chua and L. Yang. Cellular neural networks: Theory. *IEEE Transactions on circuits and systems*, 35(10):1257–1272, 1988.

[16] E. F. Codd. *Cellular automata*. Academic press, 1968.

[17] J. Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[18] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning, 2016.

[19] M. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472. Citeseer, 2011.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[21] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[22] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[23] R. Dubey, P. Agrawal, D. Pathak, T. L. Griffiths, and A. A. Efros. Investigating human priors for playing video games. *arXiv preprint arXiv:1802.10217*, 2018.

[24] D. Eagleman. *Livewired: The inside story of the ever-changing brain*. Canongate Books, 2020.

[25] P. Esser, R. Rombach, and B. Ommer. Taming transformers for high-resolution image synthesis. *arXiv preprint arXiv:2012.09841*, 2020.

[26] V. Fortuin, M. Hüser, F. Locatello, H. Strathmann, and G. Rätsch. Som-vae: Interpretable discrete representation learning on time series. *arXiv preprint arXiv:1806.02199*, 2018.

[27] D. Freeman, D. Ha, and L. Metz. Learning to predict without looking ahead: World models without forward prediction. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. https://learningtopredict.github.io.

[28] A. Gaier and D. Ha. Weight agnostic neural networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. https://weightagnostic.github.io.

[29] Y. Gal, R. McAllister, and C. E. Rasmussen. Improving PILCO with Bayesian neural network dynamics models. In *Data-Efficient Machine Learning workshop, ICML*, Apr. 2016.

[30] R. Girdhar, J. Carreira, C. Doersch, and A. Zisserman. Video action transformer network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 244–253, 2019.

[31] A. Goyal, A. Lamb, J. Hoffmann, S. Sodhani, S. Levine, Y. Bengio, and B. Schölkopf. Recurrent independent mechanisms. In *International Conference on Learning Representations*, 2021.

[32] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[33] D. Ha. Evolving stable strategies. *http://blog.otoro.net/*, 2017.

[34] D. Ha, A. Dai, and Q. V. Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[35] D. Ha and J. Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems 31*, pages 2451–2463. Curran Associates, Inc., 2018. https://worldmodels.github.io.

[36] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018.

[37] N. Hansen. The cma evolution strategy: a comparing review. *Towards a new evolutionary computation*, pages 75–102, 2006.

[38] M. Haruno, D. M. Wolpert, and M. Kawato. Mosaic model for sensorimotor learning and control. *Neural computation*, 13(10):2201–2220, 2001.

[39] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[40] S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[41] W. Huang, I. Mordatch, and D. Pathak. One policy to control them all: Shared modular policies for agent-agnostic control. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 4455–4464. PMLR, 2020.

[42] A. Jaegle, F. Gimeno, A. Brock, A. Zisserman, O. Vinyals, and J. Carreira. Perceiver: General perception with iterative attention. *arXiv preprint arXiv:2103.03206*, 2021.

[43] C. Joshi. Transformers are graph neural networks. *The Gradient*, 2020.

[44] L. Kirsch and J. Schmidhuber. Meta learning backpropagation and improving it. *arXiv preprint arXiv:2012.14905*, 2020.

[45] O. Klimov. Carracing-v0, 2016.

[46] J. Lee, Y. Lee, J. Kim, A. Kosiorek, S. Choi, and Y. W. Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3744–3753. PMLR, 09–15 Jun 2019.

[47] K. Lee, K. Lee, J. Shin, and H. Lee. Network randomization: A simple technique for generalization in deep reinforcement learning. *arXiv preprint arXiv:1910.05396*, 2019.

[48] I.-J. Liu, R. A. Yeh, and A. G. Schwing. Pic: permutation invariant critic for multi-agent deep reinforcement learning. In *Conference on Robot Learning*, pages 590–602. PMLR, 2020.

[49] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[50] T. Miconi, A. Rawal, J. Clune, and K. O. Stanley. Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity. *arXiv preprint arXiv:2002.10585*, 2020.

[51] T. Miconi, K. Stanley, and J. Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3559–3568. PMLR, 2018.

[52] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5115–5124, 2017.

[53] A. Mordvintsev, E. Randazzo, E. Niklasson, and M. Levin. Growing neural cellular automata. *Distill*, 2020. https://distill.pub/2020/growing-ca.

[54] A. Mott, D. Zoran, M. Chrzanowski, D. Wierstra, and D. J. Rezende. Towards interpretable reinforcement learning using attention augmented agents. *arXiv preprint arXiv:1906.02500*, 2019.

[55] E. Najarro and S. Risi. Meta-learning through hebbian plasticity in random networks. *arXiv preprint arXiv:2007.02686*, 2020.

[56] J. Neumann, A. W. Burks, et al. *Theory of self-reproducing automata*, volume 1102024. University of Illinois press Urbana, 1966.

[57] S. Ohsawa, K. Akuzawa, T. Matsushima, G. Bezerra, Y. Iwasawa, H. Kajino, S. Takenaka, and Y. Matsuo. Neuron as an agent, 2018.

[58] J. Ott. Giving up control: Neurons as reinforcement learning agents. *arXiv preprint arXiv:2003.11642*, 2020.

[59] E. Randazzo, A. Mordvintsev, E. Niklasson, M. Levin, and S. Greydanus. Self-classifying mnist digits. *Distill*, 2020. https://distill.pub/2020/selforg/mnist.

[60] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[61] M. Sandler, M. Vladymyrov, A. Zhmoginov, N. Miller, A. Jackson, T. Madams, et al. Meta-learning bidirectional update rules. *arXiv preprint arXiv:2104.04657*, 2021.

[62] D. Sandlin. The backwards brain bicycle: Un-doing understanding, 2019.

[63] J. Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

[64] J. Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *International Conference on Artificial Neural Networks*, pages 460–463. Springer, 1993.

[65] J. Schmidhuber. A 'self-referential'weight matrix. In *International Conference on Artificial Neural Networks*, pages 446–450. Springer, 1993.

[66] I. Sorokin, A. Seleznev, M. Pavlov, A. Fedorov, and A. Ignateva. Deep attention recurrent q-network. *arXiv preprint arXiv:1512.01693*, 2015.

[67] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[68] S. Sudhakaran, D. Grbic, S. Li, A. Katona, E. Najarro, C. Glanois, and S. Risi. Growing 3d artefacts and functional machines with neural cellular automata. *arXiv preprint arXiv:2103.08737*, 2021.

[69] C. Sun, F. Baradel, K. Murphy, and C. Schmid. Learning video representations using contrastive bidirectional transformer. *arXiv preprint arXiv:1906.05743*, 2019.

[70] Y. Tang, D. Nguyen, and D. Ha. Neuroevolution of self-interpretable agents. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2020. https://attentionagent.github.io.

[71] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[72] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[73] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[74] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[75] S. Wang, B. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[76] S. Wolfram. Cellular automata as models of complexity. *Nature*, 311(5985):419–424, 1984.

[77] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

[78] Y. Xiong, Z. Zeng, R. Chakraborty, M. Tan, G. Fung, Y. Li, and V. Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention. *arXiv preprint arXiv:2102.03902*, 2021.

[79] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim. Graph transformer networks. *arXiv preprint arXiv:1911.06455*, 2019.

[80] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola. Deep sets. *arXiv preprint arXiv:1703.06114*, 2017.

[81] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. Botvinick, O. Vinyals, and P. Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019.

[82] D. Zhang, C. Choi, J. Kim, and Y. M. Kim. Learning to generate 3d shapes with generative cellular automata. *arXiv preprint arXiv:2103.04130*, 2021.

[83] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.

[84] X. Zuo. Pytorch implementation of improving pilco with bayesian neural network dynamics models, 2018. https://github.com/zuoxingdong/DeepPILCO.