Table 6: Learning representation of entities, dynamics of those entities and interaction between entities: Different ways in which the previous work has learned representation of different entities as a set of slots, dynamics of those entities (i.e., whether different entities follow the same dynamics, different dynamics or in a dynamic way i.e context dependent manner) and how these entities interact with each other. We note that the proposed model is agnostic as to how one learn the representation of different entities, as well as how these entities behave (i.e., dynamics of those entities).

Relevant Work	Entity Encoder	Entity Dynamics	Entity Interactions
RIMs (Goyal et al., 2019)	Interactive Enc.	Different	Dynamic
MONET (Burgess et al., 2019)	Sequential Enc.	NA	NA
IODINE (Greff et al., 2019)	Iterative Reconstructive Enc.	NA	NA
C-SWM (Kipf et al., 2019)	Bottom-Up Enc.	NA	GNN
OP3 (Veerapaneni et al., 2020)	Iterative Reconstructive Enc.	Same	GNN
SA (Locatello et al., 2020a)	Iterative Interactive Enc.	NA	NA
SCOFF (Goyal et al., 2020)	Interactive Enc.	Dynamic	Dynamic

Appendix

Algorithm 1 Sequential Neural Production System model

Input: Input sequence $\{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^t, \ldots, \boldsymbol{x}^T\}$, set of embeddings describing the rules $\vec{\boldsymbol{R}}_i$, and set of MLPs (MLP_i) corresponding to each rule $\boldsymbol{R}_{1\ldots N}$. Hyper-parameters specific to NPS are the number of stages K, the number of slots M, and the number of rules N. W^k, W^q, $\widetilde{W}^k$, and $\widetilde{W}^q$ are learnable weights.

for *each input element* $\boldsymbol{x}^t$ *with* $t \leftarrow 1$ *to* T **do**

Step 1: *Update or infer the entity state in each slot* j, $\boldsymbol{V}_j^{t,0}$, *from the previous state,* $\boldsymbol{V}_j^{t-1,K}$ *and the current input* $\boldsymbol{x}_t$.

for *each stage* $h \leftarrow 0$ *to* $K-1$ **do**

Step 2: *Select {rule, primary slot} pair*

- $\boldsymbol{k}_i = \vec{\boldsymbol{R}}_i \boldsymbol{W}^k \quad \forall i \in \{1, \ldots, N\}$
- $\boldsymbol{q}_j = \boldsymbol{V}_j^{t,h} \boldsymbol{W}^q \quad \forall j \in \{1, \ldots, M\}$
- $r, p = \text{argmax}_{i,j} \left(\boldsymbol{q}_j \boldsymbol{k}_i + \gamma\right)$
  where $\gamma \sim \text{Gumbel}(0, 1)$

Step 3: *Select contextual slot*

- $\boldsymbol{q}_{r,p} = V_p^{t,h} \widetilde{\boldsymbol{W}}^q$
- $\boldsymbol{k}_j = \boldsymbol{V}_j^{t,h} \widetilde{\boldsymbol{W}}^k \quad \forall j \in \{1, \ldots, M\}$
- $c = \text{argmax}_j \left(\boldsymbol{q}_{r,p} \boldsymbol{k}_j + \gamma\right)$
  where $\gamma \sim \text{Gumbel}(0, 1)$

Step 4: *Apply selected rule to primary slot conditioned on contextual slot*

- $\widetilde{\boldsymbol{R}} = \text{MLP}_r(\text{Concatenate}([\boldsymbol{V}_p^{t,h}, \boldsymbol{V}_c^{t,h}]))$
- $\boldsymbol{V}_p^{t,h+1} = \boldsymbol{V}_p^{t,h} + \widetilde{\boldsymbol{R}}$

end

end

A Related Work

McMillan et al. (1991) have studied a neural net model, called RuleNet, that learns simple string-to-string mapping rules. RuleNet consists of two components: a feature extractor and a set of simple condition-action rules – implemented in a neural net – that operate on the extracted features. Based on a training set of input-output examples, RuleNet performs better than a standard neural net architecture in which the processing is completely unconstrained.

Key-Value Attention. Key-value attention (Bahdanau et al., 2014) defines the backbone of updates to the slots in the proposed model. This form of attention is widely used in Transformer models (Vaswani et al., 2017). Key-value attention selects an input value based on the match of a query

Algorithm 2 Parallel Neural Production System model

Input: Input sequence $\{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^t, \ldots, \boldsymbol{x}^T\}$, set of embeddings describing the applicable rules $\vec{\boldsymbol{R}}_i$, set of MLPs (MLP_i) corresponding to each rule $\boldsymbol{R}_{1\ldots N}$,and an embedding vector corresponding to the Null Rule $\vec{R}_{Null}$. Hyper-parameters specific to NPS are the number of stages K, the number of slots M, and the number of rules N. W^q, $\hat{W}^k$, W^{R_a}, W^k, $\widetilde{W}^k$, and $\widetilde{W}^q$ are learnable weights.

for *each input element* $\boldsymbol{x}^t$ *with* $t \leftarrow 1$ *to* T **do**

Step 1: *Update or infer the entity state in each slot* j, $\boldsymbol{V}_j^{t,0}$, *from the previous state,* $\boldsymbol{V}_j^{t-1,K}$ *and the current input* $\boldsymbol{x}_t$.

Step 2: *Select the set of primary slots* $\boldsymbol{P}$

- $\boldsymbol{R}_a = Concatenate([\vec{R}_i \quad \forall \boldsymbol{i} \in \{1, \ldots, N\}])\boldsymbol{W}^{R_a}$
- $\boldsymbol{k}_j = \boldsymbol{V}_j^t \hat{\boldsymbol{W}}^k \quad \forall \boldsymbol{j} \in \{1, \ldots, M\}$
- $\boldsymbol{P} = \{\boldsymbol{j} \quad \text{if} \quad \boldsymbol{R}_a \boldsymbol{k}_j + \gamma > \boldsymbol{R}_{Null} \boldsymbol{k}_j + \gamma, \quad \text{where} \quad j \in \{1, \ldots M\} \quad \text{and} \quad \gamma \sim \text{Gumbel}(0, 1)\}$

Step 3: *Select a rule for each primary slot in* $\boldsymbol{P}$

- $\boldsymbol{k}_i = \vec{\boldsymbol{R}}_i \boldsymbol{W}^k \quad \forall i \in \{1, \ldots, N\}$
- $\boldsymbol{q}_p = \boldsymbol{V}_p^t \boldsymbol{W}^q \quad \forall p \in \boldsymbol{P}$
- $\boldsymbol{r}_p = \text{argmax}_i(\boldsymbol{q}_p \boldsymbol{k}_i + \gamma) \quad \forall \boldsymbol{i} \in \{1, \ldots, N\} \quad \forall \boldsymbol{p} \in \boldsymbol{P}, \quad \text{where} \quad \gamma \sim \text{Gumbel}(0, 1)$

Step 4: *Select a contextual slot for each primary slot*

- $\boldsymbol{q}_p = \boldsymbol{V}_p^t \widetilde{\boldsymbol{W}}^q \quad \forall \boldsymbol{p} \in \boldsymbol{P}$
- $\boldsymbol{k}_j = \boldsymbol{V}_j^t \widetilde{\boldsymbol{W}}^k \quad \forall j \in \{1, \ldots, M\}$
- $c_p = \text{argmax}_j(\boldsymbol{q}_p \boldsymbol{k}_j + \gamma) \quad \forall \boldsymbol{j} \in \{1, \ldots, M\} \quad \forall \boldsymbol{p} \in \boldsymbol{P}, \quad \gamma \sim \text{Gumbel}(0, 1)$

Step 5: *Apply selected rule to each primary slot conditioned on the contextual slot*

- $\widetilde{\boldsymbol{R}}_p = \text{MLP}_{\boldsymbol{r}_p}(\text{Concatenate}([\boldsymbol{V}_p^t, \boldsymbol{V}_{c_p}^t])) \quad \forall \boldsymbol{p} \in \boldsymbol{P}$
- $\boldsymbol{V}_p^{t+1} = \boldsymbol{V}_p^t + \widetilde{\boldsymbol{R}}_p \quad \forall \boldsymbol{p} \in \boldsymbol{P}$

end

vector to a key vector associated with each value. To allow easier learnability, selection is soft and computes a convex combination of all the values. Rather than only computing the attention once, the multi-head dot product attention mechanism (MHDPA) runs through the scaled dot-product attention multiple times in *parallel*. There is an important difference with NPS: in MHDPA, one can treat different heads as different rule applications. Each head (or rule) considers *all* the other entities as relevant arguments as compared to the sparse selection of arguments in NPS.

Sparse and Dense Interactions. GNNs model pairwise interactions between all the slots hence they can be seen as capturing *dense* interactions (Scarselli et al., 2008; Bronstein et al., 2017; Watters et al., 2017; Van Steenkiste et al., 2018; Kipf et al., 2018; Battaglia et al., 2018; Tacchetti et al., 2018). Instead, verbalizable interactions in the real world are sparse (Bengio, 2017): the immediate effect of an action is only on a small subset of entities. In NPS, a selected rule only updates the state of a subset of the slots hence the interactions in NPSare sparse.

Modularity and Neural Networks. A network can be composed of several modules, each meant to perform a distinct function, and hence can be seen as a combination of experts (Jacobs et al., 1991; Bottou & Gallinari, 1991; Ronco et al., 1997; Reed & De Freitas, 2015; Andreas et al., 2016; Rosenbaum et al., 2017; Fernando et al., 2017; Shazeer et al., 2017; Kirsch et al., 2018; Rosenbaum et al., 2019; Lamb et al., 2020) routing information through a gated activation of modules. The framework can be stated as having a meta-controller c which from a particular state s, selects a particular expert or rule $a = c(s)$ as to how to transform the state s. These works generally assume that only a single expert (i.e., winner take all) is active at a particular time step. Such approaches factorize knowledge as a set of experts (i.e a particular expert is chosen by the controller). Whereas in the proposed work, there's a factorization of knowledge both in terms of entities as well as rules (i.e., experts) which act on these entities.

Graph Neural Networks. GNNs model pairwise interactions between all the entities hence they can be termed as capturing *dense* interactions (Scarselli et al., 2008; Bronstein et al., 2017; Watters et al., 2017; Van Steenkiste et al., 2018; Kipf et al., 2018; Battaglia et al., 2018; Tacchetti et al., 2018).

Type	Size	Activation
Linear	128	ReLU
Linear	Slot Dim. i.e size of $\boldsymbol{V}$	

Table 7: Architecture of the rule-specific MLP (MLP_r) in algorithm 1.

Interactions in the real world are sparse. Any action affects only a subset of entities as compared to all entities. For instance, consider a set of bouncing balls, in this case a collision between 2 balls a and b only affects a and b while other balls follow their default dynamics. Therefore, in this case it may be useful to model only the interaction between the 2 balls that collided (sparse) rather than modelling the interactions between all the balls (dense). This is the primary motivation behind NPS.

In the NPS, one can view the resultant computational graph as a result of sequential application of rules as a GNN, where the states of the entities represent the different nodes, and different rules dynamically instantiate an edge between a set of entities, again chosen in a dynamic way. Its important to emphasize that the topology of the graph induced in the NPS is dynamic, while in most GNNs the topology is fixed. Through thorough set of experiments, we show that learning sparse and dynamic interactions using NPS indeed works better than learning dense interactions using GNNs. We show that NPS outperforms state-of-the-art GNN-based architectures such as C-SWM Kipf et al. (2019) and OP3 Veerapaneni et al. (2019) while learning world-models.

Neural Programme Induction. Neural networks have been studied as a way to address the problems of learning procedural behavior and program induction (Graves et al., 2014; Reed & De Freitas, 2015; Neelakantan et al., 2015; Cai et al., 2017; Xu et al., 2018; Trask et al., 2018; Bunel et al., 2018; Li et al., 2020). The neural network parameterizes a policy distribution $p(a|s)$, which induces such a controller, which issues an instruction $a = f(s)$ which has some pre-determined semantics over how it transforms s. Such approaches also factorize knowledge as a set of experts. Whereas in the proposed work, there's a factorization of knowledge both in terms of entities as well as rules (i.e., experts) which act on these entities. Evans et al. (2019) impose a bias in the form of rules which is used to to define the state transition function, but we believe both the rules and the representations of the entities can be learned from the data with sufficiently strong inductive bias.

RIMs, SCOFF and NPS. Goyal et al. (2019, 2020) are a key inspiration for our work. RIMs consist of ensemble of modules sparingly interacting with each other via a bottleneck of attention. Each RIM module is specialized for a particular computation and hence different modules operate according to different dynamics. RIM modules are thus not interchangeable. Goyal et al. (2020) builds upon the framework of RIMs to make the slots interchangeable, and allowing different slots to follow similar dynamics. In SCOFF, the interaction different entities is via direct entity to entity interactions via attention, whereas in the NPS the interactions between the entities are mediated by sparse rules i.e., rules which have consequences for only a subset of the entities.

B NPS Specific Parameters

We use query and key size of 32 for the attention mechanism used in the selection process in steps 2 and 3 in algorithm 1. We use a gumbel temperature of 1.0 whenever using gumbel softmax. Unless otherwise specified, the architecture of the rule specific MLP is shown in table 7.

C MNIST Transformation Task

The two entities are first encoded in two slots $M = 2$: the first entity (the image) is encoded with a convolutional encoder to a d-dimensional vector; the second entity (the one-hot operation vector) is mapped to a d-dimensional vector using a learned weight matrix. Note that this step is different than Step 1 of Algorithm 1 since we don't have multiple timesteps here and we use the transformation embedding as one of the slots which is not done for any of the other experiments. We feed these two slots to the NPS module. Step 2 in algorithm 1 will match the transformation embedding to the corresponding rule. Step 3 in algorithm 1 can be used to select the correct slot for rule application (i.e. the image ($\boldsymbol{X}$) representation). Step 4 can then apply the MLP of the selected rule to the selected

	Type	Channel	Activation	Stride
Encoder	Conv2D $[4 \times 4]$	16	ELU	2
	Conv2D $[4 \times 4]$	32	ELU	2
	Conv2D $[4 \times 4]$	64	ELU	2
	Linear	100	ELU	-
Decoder	Linear	4096	ReLU	-
	Interpolate (scale factor = 2)	-	-	-
	Conv2D $[4 \times 4]$	32	ReLU	1
	Interpolate scale factor = 2	-	-	-
	Conv2D $[4 \times 4]$	16	ReLU	1
	Interpolate scale factor = 2	-	-	-
	Conv2D $[3 \times 3]$	1	ReLU	1

Table 8: The architecture of the convolutional encoder and decoder for the MNIST Transformation task.

.

slot from step 3 and output the result, which is passed through a common decoder to generate the transformed image. We train the model using binary cross entropy loss.

As highlighted before, NPS has 3 components: (1) Rule selection, (2) Entity selection, and (3) Dynamic edge instantiation. The main motivation behind using mnist transformation task is to study the rule selection aspect of NPS. Therefore, we have only 1 entity and study whether NPS can learn 4 different rules to represent the 4 operations and learn to used them correctly. We observe that NPS is indeed able to do so.

In this task, we use images of size 64×64. There are 4 possible transformations that can be applied on an image: [Translate Up, Translate Down, Rotate Left, and Rotate Right]. During training, we present an image and the corresponding operation vector and train the model to output the transformed image. We use binary cross entropy loss as our training objective. As mentioned before, we observe that NPS learns to assign a separate rule to each transformation. We can use the learned model to perform and compose novel transformations on MNIST digits. For example, if we want to perform the *Rotate Right* operation on a particular digit, we input the one-hot vector specifying the *Rotate Right* operation alongwith the digit to the learned model. The model than outputs the resultant digit after performing the operation. A demonstration of this process is presented in Figure 7. Figure 7 shows the actual outputs from the model.

Setup. We first encode the image using the convolutional encoder presented in table 8. We present the one-hot transformation vector and the encoded image representation as slots to NPS (algorithm 1). NPS applies the rule MLP corresponding to the given transformation to the encoded representation of the image which is then decoded to give the transformed image using the decoder in table 8. We use a batch size of 50 for training. We run the model for 100 epochs. This experiment takes 2 hours on a single v100.

D Coordinate Arithmetic Task

This task is mainly designed to test whether NPS can learn all the operations in the environment correctly when the operation to be performed is not provided as input and whether it can select the correct entity to apply the operation to. The task is structured as follows: We first sample a pair random coordinates $X = [(x_i, y_i), (x_j, y_j)]$. The expected output $Y = [(\hat{x_i}, \hat{y_i}), (\hat{x_j}, \hat{y_j})]$ is obtained by performing a randomly selected operation on a randomly selected coordinate (hereafter referred to as "primary coordinate") from the input. Therefore, one of the coordinates from the expected output Y has the same value as the corresponding coordinate in the input X while the other coordinate (primary coordinate) will have a different value. Since each defined operation takes 2 arguments, we also need to select another coordinate (hereafter referred to as "contextual coordinate"), to perform the operation on the primary coordinate. This contextual coordinate is selected randomly from the input X. For example, if the index of the primary coordinate is j, the index of the contextual coordinate is i, and the selected transformation is *Y Subtraction*, then the expected output $Y = [(x_i, y_i), (x_j, y_j - y_i)])$.

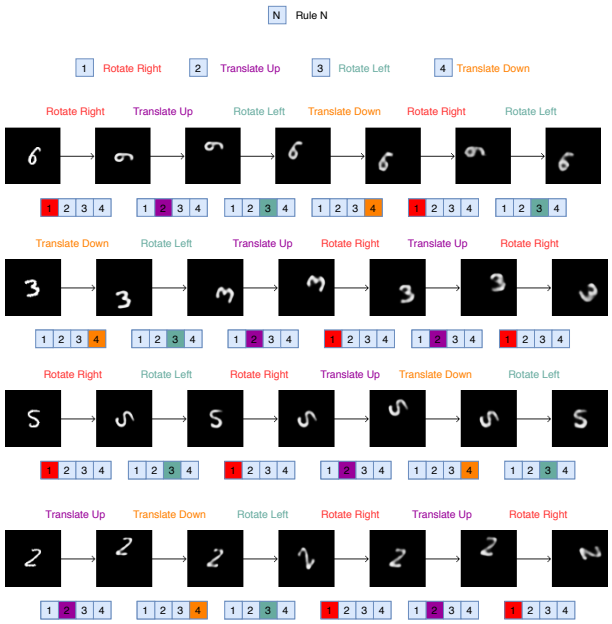


Figure 7: **MNIST Transformation Task**. Demonstration of NPS on the MNIST transformations task. The proposed model can be used to compose any novel combination of transformations on any digit by hand-picking the rule vector that was assigned to each corresponding operation during training.

We use an NPS model with 4 rules. Each coordinate is a 2-dimensional vector. The slots that are input to algorithm 1 consist of a set of 4 coordinates (2 input coordinates and their corresponding output coordinates): $\boldsymbol{X} = [(x_i, y_i), (x_j, y_j)]$ and $Y = [(\hat{x_i}, \hat{y_i}), (\hat{x_j}, \hat{y_j})]$. We concatenate the input coordinates with the corresponding output coordinates to form 4-dimensional vectors: $\boldsymbol{X} = [(x_i, y_i, \hat{x_i}, \hat{y_i}), (x_j, y_j, \hat{x_j}, \hat{y_j})]$. These make up the 2 slots that are input algorithm 1. Note that the slots are hand designed and not obtained using any convolution encoder. This is the main difference between this implementation and algorithm 1. The remaining steps of algorithm 1 are followed as is. These 2 slots are used in step 2 and step 3 of algorithm 1 to select the {primary slot, rule} pair and the contextual slot. While applying the rule MLP (i.e. step 4 of algorithm 1), we only use the input coordinates and discard the output coordinates: $\boldsymbol{X} = [(x_i, y_i), (x_j, y_j)]$. We use a rule embedding dimension of 12. We use 16 as the intermediate dimension of the Rule MLP. We also use dropout with $p = 0.35$ on the selection scores in subpart 3 of step 2 in algorithm 1.

For the baseline, we use similar rule MLPs as in NPS and replace the primary slot, contextual slot, rule selection procedure by a routing MLP similar to Fedus et al. (2021). The routing MLP consists of a 4-layered MLP with intermediate dimension of 32 interleaved with Relu activations. The input to this MLP is an 8-dimension vector consisting of both the slots mentioned in the previous paragraph. The output of this MLP is passed to 3 different linear layers: one for selecting the primary slot, one for selecting the contextual slots, and one for selecting the rule. We then apply the rule MLP of the selected rules to the corresponding slots. Here again, while applying the rule MLP we discard the outputs and only use the inputs.

We evaluate the model on 2 criterion: (1) Whether it can correctly recover all available operations from the data and learn to use a separate rule to represent each operation. (2) The mean-square error between the actual output and expected output.

Setup. We generate a training dataset of 10000 examples and a test dataset of 2000 examples. We train the model for 300 epochs using a batch size of 64. We use adam optimizer for training with a learning rate of 0.0001. Training takes 15 minutes of a single GPU.

Figure 8: Here we analyse the effect of Number of Rules on both PNPS and SNPS in the shapes stack environment. We observe that there is an optimal number of rules in both cases which is 2 for PNPS and 4 for SNPS.

E Parallel vs Sequential Rule Application

E.1 Shapes Stack

Effect of Number of Rules. We also study the effect of number of rules on both PNPS and SNPS in the test and transfer settings. Figure 8 shows the results of our analysis. We can see that there is an optimal number of rules $N = k$ for both PNPS and SNPS and the performance drops for $N < k$ and $N > k$. We can see that $k = 2$ for PNPS and $k = 4$ for SNPS. The drop in performance for $N < k$ can be attributed to lack of capacity (i.e. number of rules being less than the required number of rules for the environment). Consequently, the drop in performance for $N > k$ can be attributed to the availability of more than the required number of rules. The extra rules may serve as noise during the training process.

Training Details. We train the model for 1000000 iterations using a batch size of 20. The training takes 24 hours for PNPS and 48 hours for SNPS. We use a single v100 gpu for each run. We set the rule embedding dimension to 64.

E.2 Bouncing Balls

Setup. We consider a bouncing-balls environment in which multiple balls move with billiard-ball dynamics. We validate our model on a colored version of this dataset. This task is setup as a next step prediction task where the model is supposed to predict the motion of a set of balls that follow billiard ball dynamics. The output of each of our models consists of a seperate binary mask for each object in a frame along with an rgb image corresponding to each mask which contains the rgb values for the pixels specified by that mask. During training each of the balls can have one of the four possible colors, and during testing we increase the number of balls from 4 to 6-8.

We use the following baselines for this task:

Model Name	Test	Transfer
OP3	$0.32_{\pm 0.04}$	$0.14_{\pm 0.1}$
SNPS	$0.51_{\pm 0.07}$	$0.34_{\pm 0.08}$

Table 9: This table shows the comparison of the proposed SNPS models against the OP3 baseline in terms of ARI scores (higher is better) on the bouncing balls task. SNPS replaces the GNN used in OP3 by rules.

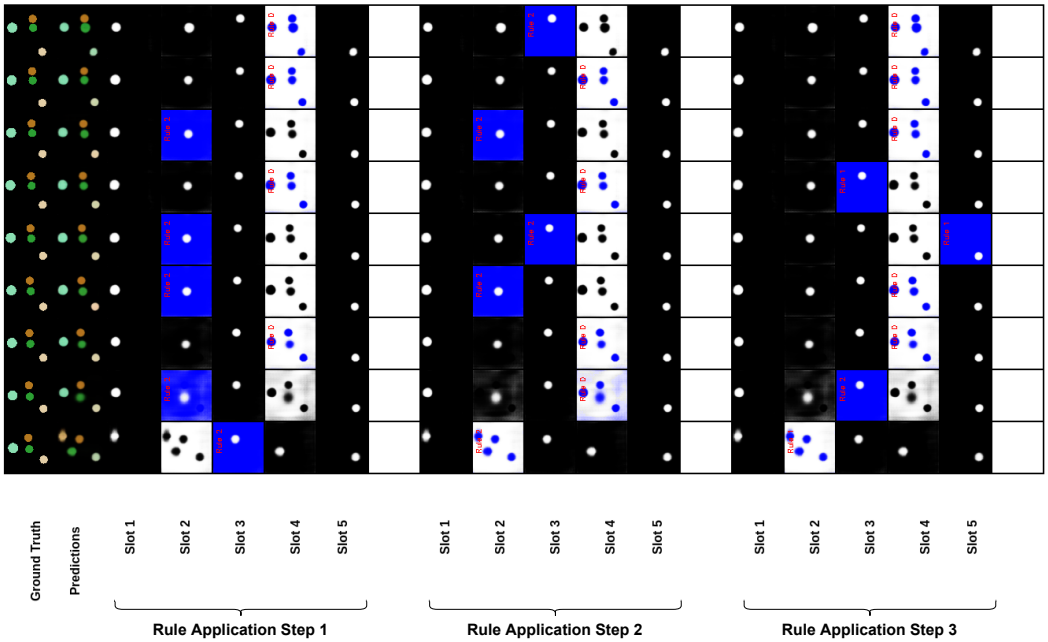


Figure 9: In the figure, we use an NPS model with 3 rules and 3 rule application steps. We analyze the entity and rule selection by NPS per time step. A rule application on a slot is shown by highlighting that slot with blue color. The index of the applied rule is also mentioned in the slot. We can see that whenever a rule is applied on the slot representing the background, rule 0 is used. On the other hand, whenever it is applied on the slots representing one of the balls, rule 1 or rule 2 are used. We can also see that rules are mainly only being applied to the two balls in the middle that are touching or close to touching while no rules are being applied to the ball on the top since it stays constant throughout this episode. The ball at the bottom is also mostly constant and receives only 1 rule application when its close to colliding with the wall.

- **SCOFF Goyal et al. (2020)**: This factorizes knowledge in terms of object files that represent entities and schemas that represent dynamical knowledge. The object files compete to represent the input using a top-down input attention mechanism. Then, each object file updates its state using a particular schema which it selects using an attention mechanism.
- **SCOFF++**: Here we use SCOFF with 1 schema. We replace the input attention mechanism in SCOFF with an iterative attention mechanism as proposed in slot attention Locatello et al. (2020b). We note that slot attention proposes to use iterative attention by building on the idea of top-down attention as proposed in (Goyal et al., 2019). We note that slot attention was only evaluated on the static images. Here, the query is a function of the hidden state of the different object files in SCOFF from the previous timestep, which allows temporal consistency in slots across the video sequence.

For our instantiation of NPS, we replace pairwise communication attention in SCOFF++ with PNPS or SNPS. From the discussion in Section 4.2 we find that SNPS outperforms the SCOFF-based model and PNPS.

We further test the proposed SNPS model against another strong object-centric baseline called OP3 (Veerapaneni et al. (2019)). We follow the exact same setup as Weis et al. (2020). For the proposed model we replace the GNN in OP3 with SNPS. We present the results of this comparison in Table 9. We can see that SNPS comfortably outperforms OP3.

An intuitive visualization of the rule and entity selection for the bouncing balls environment is presented in Figure 9.

Training Details. We run each model for 20 epochs with batch size 8 which amounts to 24 hours on a single v100 gpu. For the OP3 experiment, we use 5 rules and 3 rule application steps. We use a rule embedding dimension of 32 for our experiments.

E.3 Discussion of parallel vs sequential NPS

We have introduced two forms of NPS - Parallel and Sequential. Parallel NPS offers lower application sparsity as compared Sequential NPS while both have the same contextual sparsity. The same contextual sparsity indicates that the interactions or rules learned by both SNPS and PNPS are of same capacity since the rules in both take 2 arguments (primary slot and contextual slot). SNPS has the favourable property of being able to compose multiple rules by virtue of multiple rule application steps. We find that PNPS works better in environments where the nature of interactions are inherently dense and frequent which is expected due to the lower application sparsity of PNPS while SNPS works better in environments where interactions are much more rare. One caveat of this approach is that we need to know the structure of the environment beforehand to make an informed decision on whether to use SNPS or PNPS. Ideally, we would want an algorithm that can learn to use PNPS or SNPS depending on the environment. We leave this exploration for future work.

F Benefits of Sparse Interactions Offered by NPS

F.1 Sprites-MOT: Learning Rules for Physical Reasoning

Setup. We use the OP3 model Veerapaneni et al. (2019) as our baseline for this task. We follow the exact same setup as Weis et al. (2020). To test the proposed model, we replace the GNN-based transition model in OP3 with the proposed NPS. The output of the model consists of a seperate binary mask for each object in a frame along with an rgb image corresponding to each mask which contains the rgb values for the pixels specified by the mask.

Evaluation protocol. We use the same evaluation protocol as followed by Weis et al. (2020) which is based on the MOT (Multi-object tracking) challenge Milan et al. (2016). To compute these metrics, we have to match the objects in the predicted masks with the objects in the ground truth mask. We consider a match if the intersection over union (IoU) between the predicted object mask and ground truth object mask is greater than 0.5. The results on these metrics can be found in Table 10. We consider the following metrics:

- **Matches (Higher is better)**: This indicates the fraction of predicted object masks that are mapped to the ground truth object masks (i.e. IoU > 0.5).
- **Misses (Lower is better)**: This indicates the fraction of ground truth object masks that are not mapped to any predicted object masks.
- **False Positives (Lower is better)**: This indicates the fraction of predicted object masks that are not mapped to any ground truth masks.
- **Id Switches (Lower is better)**: This metric is designed to penalize *switches*. When a predicted mask starts modelling a different object than the one it was previously modelling, it is termed a an *id switch*. This metric indicates the fraction of objects that undergo and id switch.
- **Mostly Tracked (Higher is better)**: This is the ratio of ground truth objects that have not undergone and id switch and have been tracked for at least 80% of their lifespan.
- **Mostly Detected (Higher is Better)**: This the ratio of ground truth objects that have been correctly tracked for at least 80% of their lifespan without penalizing id switches.
- **MOT Accuracy (MOTA) (Higher is better)**: This measures the fraction of all failure cases i.e. false positives, misses, and id switches as compared to the number of objects present in all frames. Concretely, MOTA is indicated by the following formula:

$$MOTA = 1 - \frac{\sum_{t=1}^{T} M_t + FP_t + IDS_t}{\sum_{t=1}^{T} O_t} \tag{1}$$

  where, M_t, FP_t, and IDS_t indicates the misses, false positives, id switches at timestep t and O_t indicates the number of objects at timestep t.
- **MOT Precision (MOTP) (Higher is better)**: This metric measures the accuracy between the predicted object mask and the ground truth object mask relative to the total number of

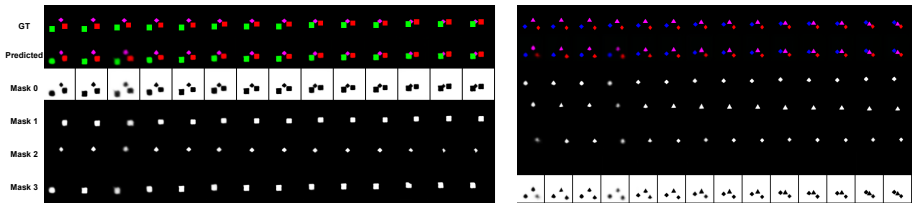


Figure 10: This figure shows the predictions of the OP3 model using the proposed NPS as a transition model. We can see that the proposed model succeeds in segregating each entity into separate slots and predicting the motion of each individual entity.

matches. Here, accuracy is measured in IoU between the predicted masks and ground truth mask. Concretely, MOTP is indicated using the following formula:

$$MOTP = \frac{\sum_{t=1}^{T} \sum_{i=1}^{I} d_t^i}{\sum_{t=1}^{T} c_t} \tag{2}$$

where, d_t^i measures the accuracy for the i^{th} matched object between the predicted and the ground truth mask measured in IoU. c_t indicates the number of matches in timestep t.

Model	MOTA ↑	MOTP ↑	Mostly Detected ↑	Mostly Tracked ↑	Match ↑	Miss ↓	ID Switches ↓	False Positives ↓
OP3	89.1 ± 5.1	78.4 ± 2.4	92.4 ± 4.0	91.8 ± 3.8	95.9 ± 2.2	3.7 ± 2.2	0.4 ± 0.0	6.8 ± 2.9
NPS	90.72 ± 5.15	79.91 ± 0.1	94.66 ± 0.29	93.18 ± 0.84	96.93 ± 0.16	2.48 ± 0.07	0.58 ± 0.02	6.2 ± 3.5

Table 10: **Sprites-MOT**. Comparison between the proposed NPS and the baseline OP3 for various MOT (multi-object tracking) metrics on the sprites-MOT dataset (↑: higher is better, ↓: lower is better). Average over 3 random seeds.

Model output. We show the predictions of the proposed model in figure 10. We use 10 rules and 3 rule application steps for our experiments. We use a rule embedding dimension of 64 for our experiments. Each rule is parameterized by a neural network as described in Tab. 7.

F.2 Physics Environment

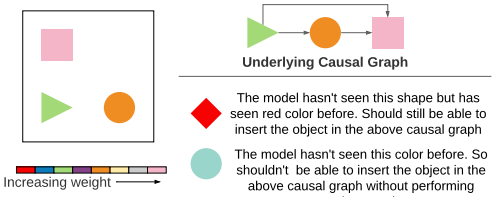


Figure 11: Demonstration of the physics environment.

A demonstration of this environment can be found in Figure 11. Each color in the environment is associated with a unique weight. The model does not have access to this information. For the model to accurately predict the outcome of an action, it needs to infer the weights from demonstrations. Inferring the correct weights will allow the model to construct the correct causal graph for each example as shown in Figure 11 which will allow it predict the correct outcome of each action in the environment. We can see that as long as the model has the correct mapping from the colors to the weights, it will be able to deal with an object of any shape irrespective of whether it has seen the shape before or not as long as it has observed the color before. Therefore, to perform well in this environment the model must infer the correct mapping from colors to weights . Learning any form of spurious correlation between the shape and the weight will penalize its performance.

The agent performs stochastic interventions (actions) in the environment to infer the weights of the blocks. Each intervention makes a block move in any of the 4 available directions (left, right, up, and down). When an intervened block A with weight W_A comes into contact with another block B with weight W_B, the block B may get pushed if $W_B < W_A$ else B will remain still. Hence, any interaction in this environment involves only 2 blocks (i.e., 2 entities) and the other blocks are not affected. Therefore, modelling the interactions between all blocks for every intervention, as is generally done using GNNs, may be wasteful. NPS is particularly well suited for this task as any

rule application takes into account only a subset of entities, hence considering interactions between those blocks only. Note that the interactions in this environment are not symmetrical and NPS can handle such relations. For example, consider a set of 3 blocks: $\{A_0, A_1, A_2\}$. If an intervention leads to A_1 pushing A_0, then NPS would apply the rule to the slot representing entity A_0. Since the movement of A_0 would depend on whether A_0 is heavier or lighter than A_1, NPS would also select the slot representing entity A_1 as a contextual slot and take it into account while applying the rule to A_0. Therefore, NPS can represent sparse and directed rules, which as we show, is more useful in this environment then learning dense and undirected relationships (or commutative operations).

Data collection. For the data, the agent performs random interventions or actions in the environment and collects the corresponding episodes. We collect 1000 episodes of length 100 for training and 10000 episodes of length 10 for evaluation.

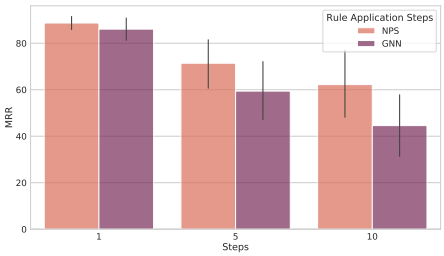
Metrics. For this task, we evaluate the predictions of the models in the latent space. We use the following metrics described in (Kipf et al., 2019) for evaluation: **Hits at Rank 1 (H@1)**: This score is 1 for a particular example if the predicted state representation is nearest to the encoded true observation and 0 otherwise. Thus, it measures whether the rank of the predicted representation is equal to 1 or not, where ranking is done over all reference state representations by distance to the true state representation. We report the average of this score over the test set. Note that, higher H@1 indicates better model performance. **Mean Reciprocal Rank (MRR)**: This is defined as the average inverse rank, i.e, MRR $= \frac{1}{N}\sum_{n=1}^{N}\frac{1}{\text{rank}_n}$ where rank_n is the rank of the n^{th} sample of the test set where ranking is done over all reference state representations. Here also, higher MRR indicates better performance.

Setup. Here, we follow the experimental setup in Ke et al. (2021). We use images of size 50×50. We first encode the current frame $\boldsymbol{x}^t$ using a convolutional encoder and pass this encoded representation using top-down attention as proposed in (Goyal et al., 2019, 2020) to extract the separate entities in the frame as slots. We use a 4-layered convolutional encoder which preserves the spatial dimensions of the image and encodes each pixel into 64 channels. We pass this encoded representation to the object encoder which extracts the entities in the frame as M 64-sized slots ($\boldsymbol{V}^t_{1\ldots M}$). We then concatenate each slot with the action ($\boldsymbol{a}^t$) taken in the current frame and pass this representation to NPS which selects a rule to apply to one of the slots using algorithm 1. The following equations describe our model in detail. We use $M = 5$ since there are 5 objects in each frame. We use a rule embedding dimension of 64 for our experiments.

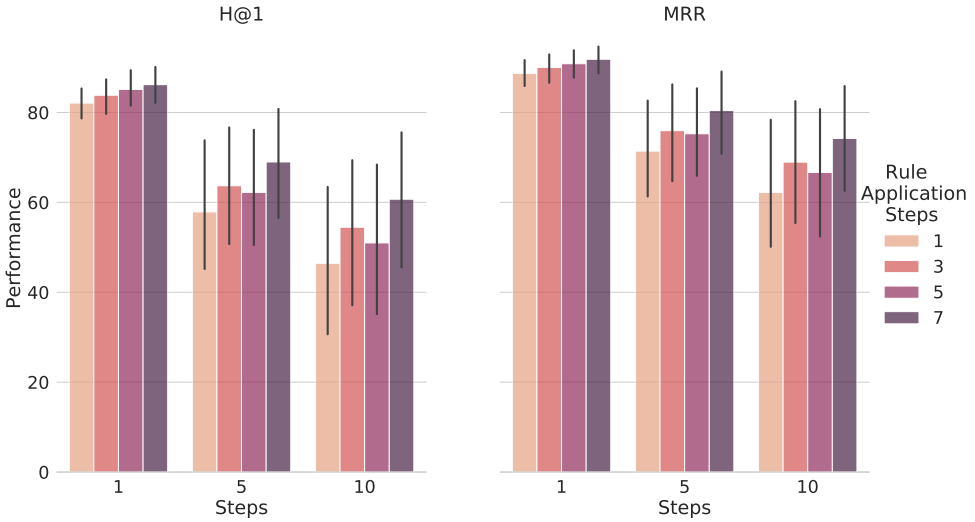
- $\hat{\boldsymbol{x}^t} = \text{Encoder}(\boldsymbol{x}^t)$
- $\boldsymbol{V}^t_{1\ldots M} = \text{Slot Attention}(\hat{\boldsymbol{x}}^t)$
- $\boldsymbol{V}^t_i = \text{Concatenate}(\boldsymbol{V}^t_i, \boldsymbol{a}^t)\ \forall \text{i} \in \{1, \ldots, M\}$
- $\boldsymbol{V}^{t+1}_{1\ldots M} = \text{NPS}(\boldsymbol{V}^t_{1\ldots M})$

Here, NPS acts as a transition model. For the baseline, we use GNN as the transition model similar to Kipf et al. (2019).

Training Details. The objective of the model is to make accurate predictions in the latent space. Mathematically, given the current frame $\boldsymbol{x}^t$ and a set of actions $\boldsymbol{a}^t, \boldsymbol{a}^{t+1}, \boldsymbol{a}^{t+2}, \ldots, \boldsymbol{a}^k$, the model performs these actions in the latent space as described in the above equations and predicts the latent state after applying these actions, i.e., $\boldsymbol{V}^{t+k}_{1\ldots M}$. Both the metrics, H@1 and MRR measure the closeness between the predicted latent state and $\boldsymbol{V}^{t+k}_{1\ldots M}$ and the ground truth latent state $\bar{\boldsymbol{V}}^{t+k}_{1\ldots M}$ which is obtained by passing frame x^{t+k} through the convolutional encoder and slot attention module. During training we use the contrastive loss which optimizes the distance between the predicted and the ground truth representations. We train the model for 100 epochs (1 hour on a single v100 gpu). Mathematically, the training objective can be formulated as follows:



(a) **Performance on MRR**



(b) **Effect of Rule Application Steps on H@1 and MRR**

Figure 12: **Physics Environment**. (a) Here we compare the performance of NPS and GNN on the MRR metric for various forward-prediction steps.(We use 1 rule and 1 rule application step) (b) Here we analyse the effect of the rule application steps on the H@1 and MRR metric for NPS. We can see that, in general, the performance increases as we increase the number of rule application steps.

$$\textit{Contrastive Training} \ : \ \underset{\text{Encoder, Transition}}{\arg\min} \quad H + \max(0, \gamma - \tilde{H})$$

$$H = \text{MSE}(\hat{\boldsymbol{V}}^{t+1}_{1\ldots M}, \boldsymbol{V}^{t+1}_{1\ldots M})$$

$$\tilde{H} = \text{MSE}(\tilde{\boldsymbol{V}}^{t+1}_{1\ldots M}, \boldsymbol{V}^{t+1}_{1\ldots M})$$

$\tilde{\boldsymbol{V}}^{t+1}$: Negative latent state obtained from random shuffling of batch

$\hat{\boldsymbol{V}}^{t+1}$: Ground truth latent state for frame x^{t+1}

Results on MRR. We show the results on the MRR metric for the physics environment in Figure 12(a). From the figure, we can see that NPS outperforms GNN on the MRR metric while predicting future steps.

Effect of Rule Application Steps. We show the effect of varying rule application steps on the physics environment in Figure 12(b). We can see that the performance increases with increasing rule application steps. Increasing the number of rule application steps seems to help since at each rule application step only sparse changes occur. We would also like to note that increasing the rule application steps comes with the cost of increased compute time due to their sequential nature.

	1 Step		5 Step		10 Step	
Model,	H@1	MRR	H@1	MRR	H@1	MRR
NPS	44.09 +/- 8.30	60.577 +/- 7.919	22.25 +/- 7.567	38.731 +/- 9.005	15.615 +/- 5.748	30.154 +/- 7.538
GNN	30.442 +/- 11.178	48.788 +/- 10.428	11.596 +/- 6.109	25.481 +/- 8.331	7.25 +/- 4.057	18.942 +/- 6.156

Table 11: Here we compare the performance of NPS and GNN on the pong environment in atari. We can see that NPS convincingly outperforms GNN. Results across 50 seeds.

	1 Step		5 Step		10 Step	
Model,	H@1	MRR	H@1	MRR	H@1	MRR
NPS	61.269 +/- 11.576	74.558 +/- 9.425	32.827 +/- 9.179	52.981 +/- 9.073	21.385 +/- 7.118	39.173 +/- 8.505
GNN	68.75 +/- 8.44	79.942 +/- 5.662	21.673 +/- 6.81	39.635 +/- 7.395	14.712 +/- 4.932	31.0 +/- 6.26

Table 12: Here we compare the performance of NPS and GNN on the space invaders environment in atari. We can see that NPS outperforms GNN in the multiple step setting (5 and 10 step). Results across 50 seeds.

F.3 Atari

This task is also setup as a next step prediction task in the latent space. We follow the same setup as the physics environment for this task. We use the same H@1 and MRR metric for evaluation. We test the proposed approach on 5 games: Pong, Space Invaders, Freeway, Breakout, and QBert.

Data collection. Similar to the physics environment, here also the agent performs random interventions or actions in the environment and collects the corresponding episodes. We collect 1000 episodes for training and 100 episodes for evaluation.

Performance on MRR. We present the results on the MRR metric in figure 13. We can see that NPS outperforms on GNN for steps. We use a rule embedding dimension of 32 for our experiments.

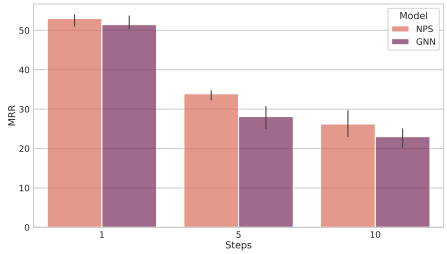


Figure 13: **Atari**. Here we compare the performance of NPS and GNN on the MRR metric for various forward-prediction steps. The results shown in the plot are averaged across 5 games.