

CSC2626 Final Project Report

Aniket Sanjay Gujarathi, Miguel Rogel, Michael (Spok) Szpakowicz

Department of Computer Science
University of Toronto
Canada

aniket.gujarathi 1007629979
miguel.rogel 1007054401
m.szpakowicz 1002285323
@mail.utoronto.ca

Abstract: The scope of this project is to reproduce the results of Neural Programmer-Interpreters (NPI) [6] on the Addition and Sorting tasks, as well as implement NPI for a new, Pick-and-Place task. NPI is a recurrent and compositional neural network that learns to represent and execute programming tasks by training on their execution traces. Our model generalizes to numbers over 8 digits long in the Addition task after training on just four digit numbers. For Sorting, we show generalization to unseen arrays with 100% accuracy, but observe declining performance with increased array length. Similarly, we find that NPI does not generalize well to previously unseen specifications in the Pick-and-Place task. Our code implementation is made open source on github¹.

Keywords: Meta-Learning, Robotics

1 Introduction

In traditional machine learning, training an algorithm to an effective level requires thousands of labeled data points. While this is acceptable when solving a single problem, it is an infeasible strategy if we wish to learn and solve a wide variety of problems in a reasonable amount of time. Therefore, efficient generalization remains a key objective of machine learning.

To make learning easier in sequential decision processes, we can group certain actions into sub-tasks. This way, we can train the algorithm on the order of these sub-tasks to learn the high-level flow of a program, which is much simpler than learning a long series of low-level processes. Such a system would be very beneficial in robotics, where tasks often consist of separate recognition and manipulation sub-tasks.

Our research objective is to verify the performance of compositional architectures that deconstruct high level task demonstrations hierarchically, and learn sequences of low level API calls. To this end, we investigate Neural Programming’s generalization potential from a relatively low amount of task demonstrations. We implement Neural Programmer-Interpreters [6] from scratch, evaluate their performance on the Addition and Sorting tasks, and compare them with an existing Keras implementation on Addition. Furthermore, we implement NPI on a new robot simulation task: Pick-and-Place. We leverage existing environment implementations for Addition² and Pick-and-Place³, and we modify the Addition environment for Sorting.

¹https://github.com/migooll/pytorch_npi

²https://github.com/mokemokechicken/keras_npi

³<https://github.com/StanfordVL/NTP-vat-release>

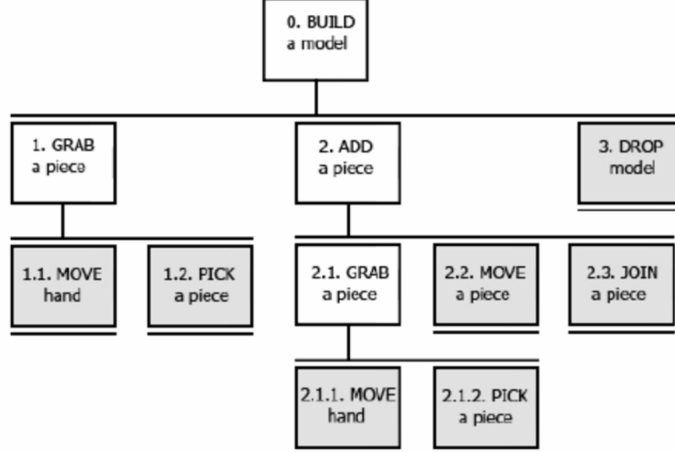


Figure 1: Hierarchical task decomposition groups decision processes into subtasks, shortening the sequence lengths necessary to finish each.

2 Related work

There are several systems that decompose an overarching task into simpler sub-tasks. DeepMind’s Neural Programmer-Interpreters paper (NPI) [6] stands out as one of the first papers to formalize this problem and provide a solution. NPI created a recurrent and compositional neural network that learns to represent and execute programs. While NPI is great for tasks such as Addition, Sorting, and canonicalizing 3D models, it is not capable of generalizing to novel programs without training.

One-shot or few-shot imitation learning confronts the aforementioned issue by training an agent that can generalize to a new task from just a single (or few) demonstration(s) at test time. Neural Task Programming [7] is an example of a such a few-shot learning approach. It builds on top of the NPI framework to provide better generalization on robotic manipulation tasks. Instead of having a LSTM core, NTPs employ a standard feed forward network as a core in combination with a temporal convolutional network for task segmentation. An observation, a task specification encoding, and the current program are used as inputs to determine the next program to be called.

NTPs demonstrate successful generalization results on semantic, topological, and length variations of task demonstrations. However, they use supervised examples of expert trajectories for training. Neural Task Graphs (NTG)[2] are a self-supervised approach to few-shot imitation learning through composition. Graphs composed of action nodes and state edges are built to capture task trajectories. New task states are then compared to the previously built graph, and actions for the new task are chosen from the state in the graph that best resembles the unseen state.

Other works attempt to generalize to completely novel tasks without the need for new expert demonstrations by training on large amounts of data. Jang et al. [3] demonstrate 44% success rate on unseen tasks by training on 25,877 robot demonstrations and 18,726 videos on 100 different tasks.

3 Method

3.1 Model

Internally, an NPI has a recurrent neural network (RNN) core called a long-short term memory network (LSTM). The inputs to this core are built as follows. First, the program arguments a_t and environment observations o_t are concatenated and encoded by a two-layer fully connected (FC) network. Concurrently, the program ID is retrieved from memory M (implemented as an embedding

layer) to get a program embedding.

$$\begin{aligned} e_t &= \text{concat}(o_t, a_t) \\ s_t &= f_{enc}(e_t) \\ p_t &= M_{prog}^i \end{aligned} \quad (1)$$

The resulting state from the environment encodings and the program embeddings are then fused through another FC layer and fed to the core recurrent module.

$$\begin{aligned} Q_t &= f_{fuse}(s_t, p) \\ h_t &= f_{LSTM}(Q_t, h_{t-1}) \end{aligned} \quad (2)$$

The hidden states at each step of the sequence are fed into three decoding FC heads to obtain the stop probability of the program r_t , the next program ID p_{t+1} , and the arguments for the next program a_{t+1} [6]

$$\begin{aligned} r_t &= f_{end}(h_t) \\ p_{t+1} &= f_{prog}(h_t) \\ a_{t+1} &= f_{arg}(h_t) \end{aligned} \quad (3)$$

Details of the architecture 8.1 as well as the model’s backward graph for computing the end probability of a program as visualized in PyTorchViz⁴ can be found in the Appendix 8.2.

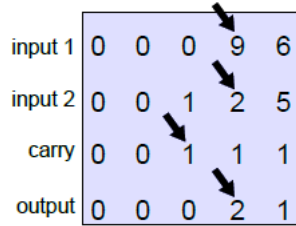
3.2 Loss function

Reed and Freitas [6] maximize the log likelihood of the output trace $\xi_t^{out} : \{p_{t+1}, a_{t+1}, r_t\}$ given the input trace $\xi_t^{in} : \{e_t, p_t, a_t\}$. We implement this as minimizing the Cross Entropy between the output and target, which should be equivalent.

$$\mathcal{L}(\xi_t^{out}, \xi_t^{in}) = \frac{1}{N} \sum_{n=1}^N \log \frac{\exp a_{t+1}}{\sum_{c=1}^C \exp a_{t+1,c}} + \log \frac{\exp p_{t+1}}{\sum_{k=1}^K \exp p_{t+1,k}} + \hat{r}_t \log r_t \quad (4)$$

3.3 Environment Observations

Addition: In the Addition environment, the observations are one-hot encoded values seen by each pointer at the current step. Every pointer corresponds to a row in an addition scratch pad, as described originally in the NPI paper[6] (see Fig. 6 in the Appendix).



input 1	0	0	0	9	6
input 2	0	0	1	2	5
carry	0	0	1	1	1
output	0	0	0	2	1

Figure 2: Scratch pad for addition as described by Reed and Freitas [6].

Sorting: Similar to the addition environment, the observations are one-hot encoded values for each pointer. There are three pointers: two traversing the array to compare contiguous values, and a third which acts as a counter. Once the third pointer reaches the end of the array, the sorting task finishes. We implement left-to-right Bubblesort, leveraging the addition environment structure.

Pick-and-Place: We train on the Pick-and-Place task by directly reading the object’s position states relative to the camera.

⁴<https://github.com/szagoruyko/pytorchviz>

```

[[0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 1 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0]]

```

Figure 3: Addition environment observation for step shown in Fig. 6. The observations are flattened and fed into the environment encoder along with the program arguments.

4 Results

We first verified the results in the NPI paper [6] by evaluating our implementation of NPI on the Addition and Sorting programs with different amounts of inputs and training examples. We then provided results of our NPI’s extension on Pick-and-Place with different initial positions and task specifications.

4.1 Addition

4.1.1 Existing Keras NPI Implementation

Before implementing our own version of NPI, we decided to start by looking at existing implementations. There are several TensorFlow and Keras implementations on github, so we decided to first look at a Keras NPI implementation by Ken Morishita [5]. Like the others, this implementation only included the Addition task specified in NPI [6].

The training process of the Keras NPI uses a DAgger approach: it first trains a model on a mini-batch of input numbers up to 10000, checks the model’s performance, retrains on the incorrect cases, and repeats the process. The algorithm checks the performance first on 10 tests on numbers up to 100, then then 100 tests on numbers up to 1000, then 1000 test on numbers up to 10000. The algorithm moves up to the next test set after succeeding on all samples of the current one, it goes back to training after failing a test set, and it terminating training upon succeeding on all three sets. It was interesting to note that while it took a long time to train on input values up to 100 ($\sim 1h$), after doing so, the network was able to generalize to values up to 1000 and 10000 fairly quickly ($\sim 30m$).

After running the training data generation scripts and training it ($\sim 1.5h$ on RTX 3070), the network was evaluated on a test set of 401 Addition problems with a perfect score. The network’s generalization abilities were tested further with larger numbers it never trained on, and it achieved a 100% success rate on numbers up to 1,000,000. When testing on numbers up to 10,000,000, it made a single mistake, predicting the answer of $9,240,899 + 290,680 = 19,531,579$ instead of 9,531,579. This environment was only meant to work with up to 8 digits, and the model functioned correctly on all numbers that add up to an 8 digit number. Throughout running the training code multiple times, it did however fail in two more cases, predicting that $10 + 9 = 119$ and $18 + 1 = 119$. It is interesting to note that in all three failure cases, the model failed in the same way - it carried a 1 to the beginning of the program when it did not need to.

4.1.2 Our PyTorch NPI Implementation

In Addition to running a previously existing NPI implementation, we implemented our own PyTorch version of the NPI model from scratch. We adapted our model to train and test with the Addition environment implemented in Keras NPI [5], and we leveraged PyTorch Lightning [1] for training.

We generated 10,301 datapoints using Keras NPI training data generation scripts. Each datapoint consists of the addition of two random numbers, where the maximum for a single number is 10,000. As each pair of numbers is generated, their value increases; thus, the Addition task increases in difficulty as the model traverses through the data (if not shuffled). Our training procedure is influenced by the original NPI setup [6]: we train for 10 epochs using the Adam optimizer [4] with an initial

learning rate of 0.0001, batch size 1, and learning rate decay of 0.95 every 10,000 steps (~ 1 h on GTX 1080).

We generated 401 testing datapoints for each maximum number of digits of the numbers to be added as specified in table 1. Our NPI implementation was able to perform addition of two numbers of up to 8 digits with 100% accuracy. Furthermore, we evaluated our NPI’s training efficiency by comparing test results against a sequence-to-sequence LSTM. Both models were trained for 5000 steps on 1301 data-points of up to 4 digits. NPI was able to obtain a much better accuracy in the same number of training steps. However, frequency re-sampling has adverse effects on performance, as seen by the NPI_{fr} model results. We thus drop frequency re-sampling on further training procedures.

Max Number of Digits	PyTorch NPI Accuracy
4	100%
5	100%
6	100%
7	100%
8	100%

Table 1: Accuracy vs max number length in Addition. The PyTorch NPI model was only trained with numbers up to 4 digits

Model	Accuracy
NPI	65.64%
NPI_{fr}	60.72%
LSTM	25.67%

Table 2: Models accuracies after training for 5000 steps on the same Addition data.

```
{
  "in1": 130, "in2": 699, "expect": 829, "result": 829, "correct": true
}
{
  "in1": 509, "in2": 641, "expect": 1150, "result": 1150, "correct": true
}
{
  "in1": 930, "in2": 522, "expect": 1452, "result": 1452, "correct": true
}
{
  "in1": 265, "in2": 112, "expect": 377, "result": 377, "correct": true
}
{
  "in1": 921, "in2": 305, "expect": 1226, "result": 1226, "correct": true
}
{
  "in1": 467, "in2": 221, "expect": 688, "result": 688, "correct": true
}
{
  "in1": 104, "in2": 902, "expect": 1006, "result": 1006, "correct": true
}
{
  "in1": 9814902, "in2": 5565751, "expect": 15380653, "result": 15380653, "correct": true
}
{
  "in1": 8787596, "in2": 443674, "expect": 9231270, "result": 9231270, "correct": true
}
{
  "in1": 8090210, "in2": 2501641, "expect": 10591851, "result": 10591851, "correct": true
}
{
  "in1": 9016485, "in2": 9392765, "expect": 18409250, "result": 18409250, "correct": true
}
{
  "in1": 2217185, "in2": 5089112, "expect": 7306297, "result": 7306297, "correct": true
}
{
  "in1": 5855093, "in2": 6852045, "expect": 12707138, "result": 12707138, "correct": true
}
{
  "in1": 8760183, "in2": 1195934, "expect": 9956117, "result": 9956117, "correct": true
}
{
  "in1": 6554632, "in2": 4052359, "expect": 10606991, "result": 10606991, "correct": true
}
{
  "in1": 350125, "in2": 7132782, "expect": 7482907, "result": 7482907, "correct": true
}
{
  "in1": 6793451, "in2": 1385783, "expect": 8179234, "result": 8179234, "correct": true
}
{
  "in1": 7712358, "in2": 6756538, "expect": 14468896, "result": 14468896, "correct": true
}
{
  "in1": 9166735, "in2": 1603737, "expect": 10770472, "result": 10770472, "correct": true
}
{
  "in1": 2732236, "in2": 8037064, "expect": 10769300, "result": 10769300, "correct": true
}
```

Figure 4: Some examples of PyTorch NPI Addition results

4.2 Sorting

A high-level overview of the program trace implemented for Sorting is shown in Table 4. We first tested NPI’s generalization to unseen sorting examples while keeping task length. We trained on 1,000 arrays of lengths ranging from 2 to 20 for 10 epochs using the same architecture and optimizer

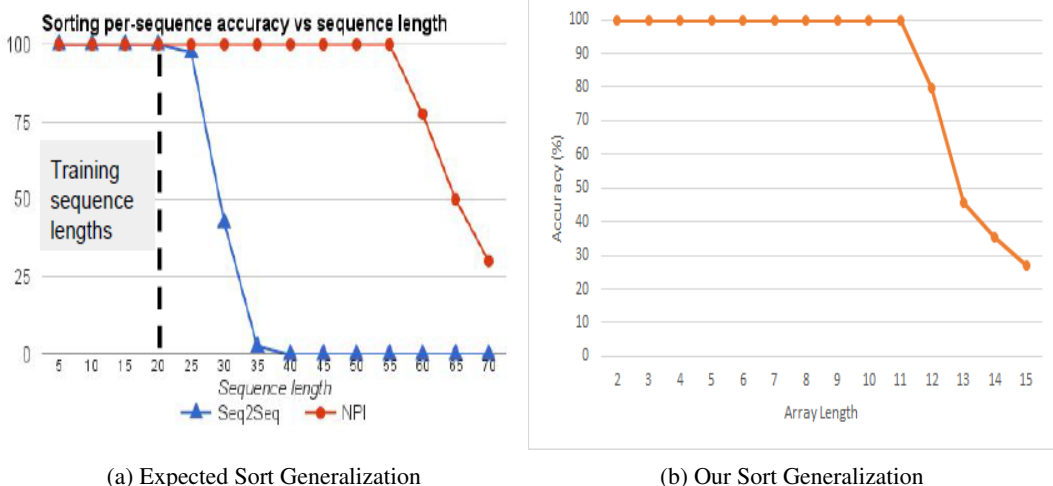


Figure 5: Out of Domain Sort Generalization Comparison

settings as mentioned in Addition. We did not share the same model between tasks. NPI was able to achieve 100% accuracy on arrays with lengths 2 to 20 not seen during training.

Nonetheless, when testing in out of domain lengths, our NPI implementation was not able to generalize. We trained on 1000 arrays of lengths ranging from 2 to 10 for 30 epochs and tested performance on arrays from length 10 to 15. Performance decreased rapidly as length increased, as seen in 5. Our generalization is similar to the Seq2Seq performance curve shown, and indeed we noticed that our initial model was being trained similarly to a Seq2Seq fashion. Once we corrected the training procedure, we were able to obtain much better results in the addition task with less training, but we were unable to match our previous best performance on sorting.

Program	Function definition	Sub-programs
BUBBLESORT	Perform bubble sort in ascending order	BUBBLE, RESET
BUBBLE	Perform one sweep of pointers left to right	ACT, BSTEP
RESET	Move both pointers all the way to the left	LSHIFT
BSTEP	Conditionally swap and advance pointers	COMPSWAP, RSHIFT
COMPSWAP	Conditionally swap two elements	ACT
LSHIFT	Shift a specific pointer one step left	ACT
RSHIFT	Shift a specific pointer one step right	ACT
ACT	Swap two values at pointer locations or move a pointer	-

Table 3: Sorting Program Trace

4.3 Pick-and-Place

After understanding Stanford’s VAT environment we interfaced it with NPI and carried out testing similarly to the previous tasks. When specifying the Pick-and-Place task with initial object states, NPI was able to generalize to unseen initial conditions. NPI was trained on 55 Pick-and-Place tasks, and when tested on the same tasks with different initial object states, it performed with a 100% accuracy. As expected, NPI was unable to generalize to unseen tasks, as after training on 40 Pick-and-Place tasks, it failed on all 15 unseen test tasks. Through this, we were able to verify the claim in NTP that NPI is unable to generalize to unseen task specifications.

5 Limitations

NPI is rely heavily on expert supervision. This limits NPI’s applications to tasks where an expert already exists, which may defeat the purpose of having Neural Programmer Interpreters when a program can already solve the problem. Furthermore, NPI has problems with generalizing to out-of-domain task specifications, as we saw in the Pick-and-Place task. Moreover, we weren’t able to make NPI generalize to out-of-domain lengths. It is unclear to us whether there is still a fault in our implementation, or if the generalizing results by [6] on Sorting need careful fine-tuning and / or extra training to reproduce. Finally, NPI only works on environments in which a task is already “embedded”. Without modifications, NPI is not able to perform tasks where a different end result or trajectory is necessary, such as sorting in ascending as opposed to descending order when being trained only on descending order.

6 Conclusion

We implemented NPI from scratch on three tasks: Addition, Sort, and Pick-and-Place. Our Addition results showed that NPI has the ability to leverage hierarchical composition to generalize to two-number addition with larger numbers not seen during training. We also were able to obtain 100% sorting performance on unseen arrays of one digit numbers, but unfortunately our NPI implementation did not extend to longer arrays. We do provide the sorting environment in our code for further experimentation, which to the best of our knowledge did not exist as open source. Finally, we tested the performance of NPI with simple task specification on Pick-and-Place using a modification of the NTP environment. While we observed robustness against changing initial conditions, NPI’s failed to generalize to unseen task specifications. A good follow-up project is to implement Neural Task Programming [7], and check whether the complex hierarchical task specification can be simplified while still being able to generalize.

7 Citations

- [1] William Falcon and The PyTorch Lightning team. *PyTorch Lightning*. Version 1.4. Mar. 2019. DOI: [10.5281/zenodo.3828935](https://doi.org/10.5281/zenodo.3828935). URL: <https://github.com/Lightning-AI/lightning>.
- [2] De-An Huang et al. “Neural task graphs: Generalizing to unseen tasks from a single video demonstration”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 8565–8574.
- [3] Eric Jang et al. “Bc-z: Zero-shot task generalization with robotic imitation learning”. In: *Conference on Robot Learning*. PMLR. 2022, pp. 991–1002.
- [4] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: <http://arxiv.org/abs/1412.6980>.
- [5] Ken Morishita. *keras_npi*. Nov. 2016. URL: https://github.com/mokemokechicken/keras_npi.
- [6] Scott E. Reed and Nando de Freitas. “Neural Programmer-Interpreters”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: <http://arxiv.org/abs/1511.06279>.
- [7] Danfei Xu et al. “Neural Task Programming: Learning to Generalize Across Hierarchical Tasks”. In: May 2018. DOI: [10.1109/ICRA.2018.8460689](https://doi.org/10.1109/ICRA.2018.8460689).

8 Appendix

8.1 Architecture details

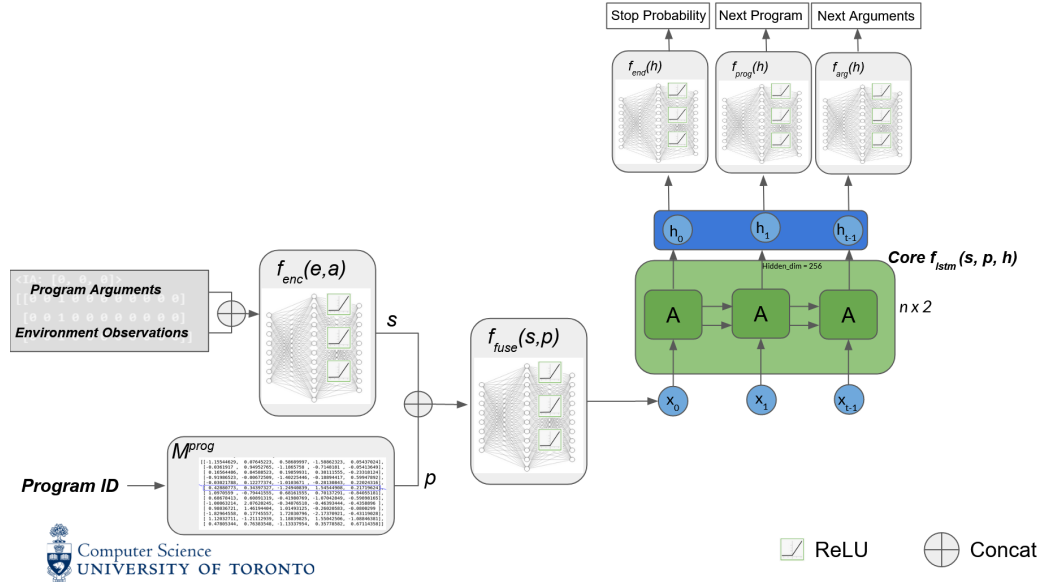


Figure 6: Graphical depiction of Pytorch NPI architecture

Function	Layers	Hidden Size	Activation
f_{enc}	2	128, 128	ReLU
f_{fuse}	2	256, 256	ReLU
f_{LSTM}	2	256	—
M_{prog}^i	—	256, 256	—
f_{end}	2	128, 1	ReLU
f_{prog}	2	128, 64	ReLU
f_{arg}	2	128, $\text{arg_depth} * \text{max_arg_num}$	ReLU

Table 4: Architecture layers, hidden sizes and activations

8.2 Backward NPI Graph



Figure 7: PyTorch NPI backward graph of inputs to end probability of program

8.3 Contribution

1. Aniket Gujarathi : Sorting environment and sub-task
2. Miguel Rogel : Pytorch model, training, addition sub-task, sort sub-task, pick-and-place sub-task
3. Michael (Spok) Szpakowicz : NTP environment, pick-and-place sub-task, video editing and presentation

8.4 Code

1. PyTorch NPI: https://github.com/migooll/pytorch_npi
2. Updated NTP VAT: <https://github.com/spok7/NTP-vat-release>