Presented by

Fethiye İrmak Doğan

---

Central Challenges of AI

- Teaching machine to learn new programs
Central Challenges of AI

- Teaching machine to learn new programs
- Execute these programs automatically
Neural Programmer-Interpreters (NPI)

Neural Programmer-Interpreters is a recurrent and compositional neural network that learns how to

- represent a program
Neural Programmer-Interpreters (NPI)

Neural Programmer-Interpreters is a recurrent and compositional neural network that learns how to

- represent a program
- execute a program (as an interpreter)
Neural Programmer-Interpreters is a recurrent and compositional neural network that learns how to

- represent a program
- execute a program (as an interpreter)
- generate new program embeddings (as a programmer)
Task agnostic recurrent core: LSTM based sequence model which is a single core module with the shared parameters across all tasks
Compositional architecture of NPI

**Task agnostic recurrent core**: LSTM based sequence model which is a single core module with the shared parameters across all tasks

**Persistant key-value program memory**: Learnable key-value memory of program embeddings which provides learning and reusing programs
Compositional architecture of NPI

Task agnostic recurrent core: LSTM based sequence model which is a single core module with the shared parameters across all tasks

Persistent key-value program memory: Learnable key-value memory of program embeddings which provides learning and reusing programs

Domain-specific encoders: encoder that enables NPI to operate in diverse environments
Curriculum Learning\(^2\): Start small, learn easier aspects of the task or easier subtasks, and then gradually increase the difficulty level.

Curriculum Learning\textsuperscript{2}: Start small, learn easier aspects of the task or easier subtasks, and then gradually increase the difficulty level.

Rich Supervision: Rather than using large number of relatively weak labels, exploit from the fewer fully supervised execution traces

Dynamically Programmable Networks

- activations of one network become the weights of a second network
Related Work

Dynamically Programmable Networks
- activations of one network become the weights of a second network

Neural Turing Machine
- learning and executing simple programs
Related Work

Dynamically Programmable Networks
- activations of one network become the weights of a second network

Neural Turing Machine
- learning and executing simple programs

Program Induction
- inducing a program given example input and output pairs
Novelties of NPI

- being trained on execution traces instead of input and output pairs
Novelties of NPI

- being trained on execution traces instead of input and output pairs
- incorporating *compositional structure* into the network using a program memory
Novelties of NPI

- being trained on execution traces instead of input and output pairs
- incorporating **compositional structure** into the network using a program memory
- learning new programs by combining sub-programs
NPI Core acts as a **router** between programs and there is a single inference core shared by arbitrary programs.

**Figure**: Example execution trace of single-digit addition
Figure: Example execution trace of single-digit addition

NPI Core is conditioned on

- current state observations:
Figure: Example execution trace of single-digit addition

NPI Core is conditioned on

- current state observations:
  - learnable program embedding, program arguments, feature representation of the environment
NPI Core is conditioned on

- current state observations:
  - learnable program embedding, program arguments, feature representation of the environment

- previous hidden unit states

**Figure:** Example execution trace of single-digit addition
NPI Core

Figure: Example execution trace of single-digit addition

NPI Core outputs

- key indicating what program to call next
**Figure**: Example execution trace of single-digit addition

NPI Core outputs

- key indicating what program to call next
- probability of ending the current program
**Figure:** Example execution trace of single-digit addition

NPI Core outputs

- key indicating what program to call next
- probability of ending the current program
- argument for the following program (passed by reference or value)
Different programs correspond to different embeddings stored in a persistent memory.

Figure: Example execution trace of single-digit addition
Feed-Forward steps of program inference

\[ s_t = f_{\text{enc}}(e_t, a_t) \]
Feed-Forward steps of program inference

\[ s_t = f_{\text{enc}}(e_t, a_t) \]
\[ h_t = f_{\text{lstm}}(s_t, p_t, h_{t-1}) \]
Feed-Forward steps of program inference

\[ s_t = f_{\text{enc}}(e_t, a_t) \]
\[ h_t = f_{\text{lstm}}(s_t, p_t, h_{t-1}) \]
\[ r_t = f_{\text{end}}(h_t), \quad k_t = f_{\text{prog}}(h_t), \quad a_{t+1} = f_{\text{arg}}(h_t) \]
Program Embedding

\[ k_t : \text{program key embedding} \quad i: \text{program ID} \quad p_{t+1} : \text{next program embedding} \]

\[ M^{\text{key}} : \text{key embeddings which stores all the program keys} \quad M^{\text{prog}} : \text{program embeddings} \]

\[ i^* = \arg \max_{i=1..N} (M^{\text{key}})_{i,:}^T k_t, \quad p_{t+1} = M^{\text{prog}}_{i^*, :} \]
Environmental State

\[ e_t : \text{environment observation at time } t \quad p_t : \text{program embedding} \quad a_t : \text{output arguments at time } t \]
\[ f_{env} : \text{domain specific transition mapping} \quad e_{t+1} : \text{next environmental state} \]

\[ e_{t+1} \sim f_{env}(e_t, p_t, a_t) \]
Algorithm 1 Neural programming inference

1: **Inputs**: Environment observation $e$, program id $i$, arguments $a$, stop threshold $\alpha$
2: **function** RUN($i$, $a$)
3: \quad $h \leftarrow 0$, $r \leftarrow 0$, $p \leftarrow M_i^{prog}$
4: \quad **while** $r < \alpha$ **do**
5: \quad \quad $s \leftarrow f_{enc}(e, a)$, $h \leftarrow f_{lstm}(s, p, h)$
6: \quad \quad $r \leftarrow f_{end}(h)$, $k \leftarrow f_{prog}(h)$, $a_2 \leftarrow f_{arg}(h)$
7: \quad \quad $i_2 \leftarrow \text{arg max}(M_j^{key})$\text{^T}\text{k}_{j=1..N}$
8: \quad **if** $i == \text{ACT}$ **then** $e \leftarrow f_{env}(e, p, a)$
9: \quad **else** RUN($i_2$, $a_2$)

$\triangleright$ Init LSTM and return probability.

$\triangleright$ Feed-forward NPI one step.

$\triangleright$ Decide the next program to run.

$\triangleright$ Update the environment based on ACT.

$\triangleright$ Run subprogram $i_2$ with arguments $a_2$
Inference Algorithm

Algorithm 1 Neural programming inference

1: **Inputs:** Environment observation \( e \), program id \( i \), arguments \( a \), stop threshold \( \alpha \)
2: **function** \( \text{RUN}(i, a) \)
3: \( h \leftarrow 0, r \leftarrow 0, p \leftarrow M^\text{prog}_{i,i} \) \( \triangleright \) Init LSTM and return probability.
4: **while** \( r < \alpha \) **do**
5: \( s \leftarrow f_{\text{enc}}(e, a), h \leftarrow f_{\text{lstm}}(s, p, h) \) \( \triangleright \) Feed-forward NPI one step.
6: \( r \leftarrow f_{\text{end}}(h), k \leftarrow f_{\text{prog}}(h), a_2 \leftarrow f_{\text{arg}}(h) \)
7: \( i_2 \leftarrow \arg \max(M_{j,:}^{\text{key}})^T k \) \( j=1...N \) \( \triangleright \) Decide the next program to run.
8: **if** \( i == \text{ACT} \) **then** \( e \leftarrow f_{\text{env}}(e, p, a) \) \( \triangleright \) Update the environment based on ACT.
9: **else** \( \text{RUN}(i_2, a_2) \) \( \triangleright \) Run subprogram \( i_2 \) with arguments \( a_2 \)

- actions are encapsulated into ACT program shared across tasks and indicated by the NPI-generated arguments \( a_t \)
Inference Algorithm

Algorithm 1 Neural programming inference

1: **Inputs**: Environment observation $e$, program id $i$, arguments $a$, stop threshold $\alpha$
2: **function** RUN($i, a$)
3: $h \leftarrow 0$, $r \leftarrow 0$, $p \leftarrow M_{i,t}^{prog}$
4:  **while** $r < \alpha$ **do**
5:    $s \leftarrow f_{enc}(e, a)$, $h \leftarrow f_{lstm}(s, p, h)$  **▷** Feed-forward NPI one step.
6:    $r \leftarrow f_{end}(h)$, $k \leftarrow f_{prog}(h), a_2 \leftarrow f_{arg}(h)$  **▷** Decide the next program to run.
7:    $i_2 \leftarrow \arg\max_j (M_{j,t}^{key})^T k$
8:  **if** $i == \text{ACT}$ **then** $e \leftarrow f_{env}(e, p, a)$  **▷** Update the environment based on ACT.
9:  **else** RUN($i_2, a_2$)  **▷** Run subprogram $i_2$ with arguments $a_2$

- actions are encapsulated into ACT program shared across tasks and indicated by the NPI-generated arguments $a_t$
- core module is completely agnostic to the data modality used in the state encoding
Training

\[ \varepsilon_t^{inp} : \{e_t, i_t, a_t\} \text{ and } \varepsilon_t^{out} : \{i_{t+1}, a_{t+1}, r_t\} \] are the execution traces
Training

\[ \varepsilon_t^{\text{inp}} : \{e_t, i_t, a_t\} \text{ and } \varepsilon_t^{\text{out}} : \{i_{t+1}, a_{t+1}, r_t\} \text{ are the execution traces} \]

\[ i_t \text{ and } i_{t+1} \text{ are program IDs and row indices in } M^{\text{key}} M^{\text{prog}} \text{ of the programs to run at time } t \text{ and } t+1 \]
Training

\[ \xi^\text{inp}_t : \{ e_t, i_t, a_t \} \text{ and } \xi^\text{out}_t : \{ i_{t+1}, a_{t+1}, r_t \} \text{ are the execution traces} \]

\( i_t \text{ and } i_{t+1} \) are program IDs and row indices in \( M^\text{key} \) \( M^{\text{prog}} \) of the programs to run at time \( t \) and \( t+1 \)

\[ \theta^* = \arg \max_\theta \sum_{(\xi^\text{inp},\xi^\text{out})} \log P(\xi^\text{out}|\xi^\text{inp}; \theta) \]
Training

\( \varepsilon^\text{inp}_t : \{e_t, i_t, a_t\} \) and \( \varepsilon^\text{out}_t : \{i_{t+1}, a_{t+1}, r_t\} \) are the execution traces.

\( i_t \) and \( i_{t+1} \) are program IDs and row indices in \( M^\text{key} \) and \( M^\text{prog} \) of the programs to run at time \( t \) and \( t+1 \).

\[
\theta^* = \arg \max_{\theta} \sum_{(\varepsilon^\text{inp}, \varepsilon^\text{out})} \log P(\xi^\text{out} | \xi^\text{inp}; \theta)
\]

since traces are variable length above equation can be written as:

\[
\theta^* = \arg \max_{\theta} \sum_{(\xi^\text{inp}, \xi^\text{out})} \log P(\xi^\text{out} | \xi^\text{inp}; \theta)
\]
Training

\[ \varepsilon^\text{inp}_t : \{e_t, i_t, a_t\} \text{ and } \varepsilon^\text{out}_t : \{i_{t+1}, a_{t+1}, r_t\} \text{ are the execution traces} \]

\[ i_t \text{ and } i_{t+1} \text{ are program IDs and row indices in } M^\text{key} \text{ of the programs to run at time } t \text{ and } t+1 \]

\[ \theta^* = \arg \max_\theta \sum_{(\varepsilon^\text{inp}, \varepsilon^\text{out})} \log P(\varepsilon^\text{out} | \varepsilon^\text{inp}; \theta) \]

since traces are variable length above equation can be written as:

\[ \log P(\varepsilon^\text{out} | \varepsilon^\text{inp}; \theta) = \sum_{t=1}^{T} \log P(\varepsilon^\text{out}_t | \varepsilon^\text{inp}_1, ..., \varepsilon^\text{inp}_t; \theta) \]
Training

\( \varepsilon_t^{\text{inp}} : \{e_t, i_t, a_t\} \) and \( \varepsilon_t^{\text{out}} : \{i_{t+1}, a_{t+1}, r_t\} \) are the execution traces.

\( i_t \) and \( i_{t+1} \) are program IDs and row indices in \( M^{\text{key}} \ M^{\text{prog}} \) of the programs to run at time \( t \) and \( t+1 \).

\[ \theta^* = \arg \max_{\theta} \sum_{(\xi^{\text{inp}}, \xi^{\text{out}})} \log P(\xi^{\text{out}}|\xi^{\text{inp}}; \theta) \]

since traces are variable length above equation can be written as:

\[ \log P(\xi^{\text{out}}|\xi^{\text{inp}}; \theta) = \sum_{t=1}^{T} \log P(\xi^{\text{out}}_t|\xi^{\text{inp}}_1, \ldots, \xi^{\text{inp}}_t; \theta) \]

since hidden unit activations are capable of capturing temporal dependencies, right hand side can be written as:
Training

\( \xi^\text{inp}_t : \{e_t, i_t, a_t\} \) and \( \xi^\text{out}_t : \{i_{t+1}, a_{t+1}, r_t\} \) are the execution traces.

\( i_t \) and \( i_{t+1} \) are program IDs and row indices in \( M^{\text{key}} M^{\text{prog}} \) of the programs to run at time \( t \) and \( t+1 \)

\[
\theta^* = \arg \max_{\theta} \sum_{(\xi^\text{inp}, \xi^\text{out})} \log P(\xi^\text{out}|\xi^\text{inp}; \theta)
\]

since traces are **variable length** above equation can be written as:

\[
\log P(\xi^\text{out}|\xi^\text{inp}; \theta) = \sum_{t=1}^T \log P(\xi^\text{out}_t|\xi^\text{inp}_1, \ldots, \xi^\text{inp}_t; \theta)
\]

since **hidden unit activations** are capable of capturing temporal dependencies, right hand side can be written as:

\[
\log P(\xi^\text{out}_t|\xi^\text{inp}_1, \ldots, \xi^\text{inp}_t) = \log P(i_{t+1}|h_t) + \log P(a_{t+1}|h_t) + \log P(r_t|h_t)
\]
- **Adaptive curriculum**: sample frequency of a program is determined by model’s current prediction error in that program
Adaptive curriculum: sample frequency of a program is determined by model’s current prediction error in that program
- forces the model to focus on learning the program worst in execution
**Adaptive curriculum**: sample frequency of a program is determined by model’s current prediction error in that program

- forces the model to focus on learning the program worst in execution

Memory advantage thanks to **parallel execution** in sub-programs
**Task:** read in the digits of two base-10 numbers and produce the digits of the answer

---

**Addition**

- **Task:** read in the digits of two base-10 numbers and produce the digits of the answer
- Four pointers: one for each of the two input numbers, one for the carry, and another to write the output

---

**Task:** read in the digits of two base-10 numbers and produce the digits of the answer

Four pointers: one for each of the two input numbers, one for the carry, and another to write the output

Model sees the current values at each pointer locations as 1-of-K encodings \(^3\) (K=10)

---

\(^3\) https://en.wikipedia.org/wiki/One-hot
Addition

- **Task:** read in the digits of two base-10 numbers and produce the digits of the answer
- Four pointers: one for each of the two input numbers, one for the carry, and another to write the output
- Model sees the current values at each pointer locations as 1-of-K encodings $^3 (K=10)$

$$f_{enc}(Q, i_1, i_2, i_3, i_4, a_t) = MLP([Q(1, i_1), Q(2, i_2), Q(3, i_3), Q(4, i_4), a_t(1), a_t(2), a_t(3)])$$

$Q \in \mathbb{R}^{4 \times N \times K}$ is the scratch pad, first dimension of $Q$ corresponds to scratch pad rows, $N$ is the number of columns (digits) and $K$ is the one-hot encoding dimension

$^3$https://en.wikipedia.org/wiki/One-hot
### Task and Environment Descriptions

**Sample Complexity and Generalization**

Learning New Programs with a Fixed Core

Solving Multiple Tasks with a Single Network

### Addition

<table>
<thead>
<tr>
<th>Program</th>
<th>Descriptions</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Perform multi-digit addition</td>
<td>ADD1, LSHIFT</td>
</tr>
<tr>
<td>ADD1</td>
<td>Perform single-digit addition</td>
<td>ACT, CARRY</td>
</tr>
<tr>
<td>CARRY</td>
<td>Mark a 1 in the carry row one unit left</td>
<td>ACT</td>
</tr>
<tr>
<td>LSHIFT</td>
<td>Shift a specified pointer one step left</td>
<td>ACT</td>
</tr>
<tr>
<td>RSHIFT</td>
<td>Shift a specified pointer one step right</td>
<td>ACT</td>
</tr>
<tr>
<td>ACT</td>
<td>Move a pointer or write to the scratch pad</td>
<td>-</td>
</tr>
</tbody>
</table>

(a) Example scratch pad and pointers used for computing “96 + 125 = 221”. Carry step is being implemented.

(b) Actual trace of addition program generated by our model on the problem shown to the left. Note that we substituted the ACT calls in the trace with more human-readable steps.

**Figure:** Illustration of the addition environment
Task and Environment Descriptions
Sample Complexity and Generalization
Learning New Programs with a Fixed Core
Solving Multiple Tasks with a Single Network

Sorting

- **Task**: comparing each pair of adjacent items and swaps them if they are in the wrong order (Bubble Sort \(^4\))

---

\(^4\)https://en.wikipedia.org/wiki/Bubble_sort
Task: comparing each pair of adjacent items and swaps them if they are in the wrong order (Bubble Sort \(^4\))

\[
f_{enc}(Q, i_1, i_2, a_t) = MLP([Q(1, i_1), Q(1, i_2), a_t(1), a_t(2), a_t(3)])
\]

\(Q \in \mathbb{R}^{1 \times N \times K}\) is the scratch pad, \(N\) is the array length and \(K\) is the array entry embedding dimension.

---

\(^4\)https://en.wikipedia.org/wiki/Bubble_sort
### Sorting

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Related Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLESORT</td>
<td>Perform bubble sort (ascending order)</td>
<td>BUBBLE, RESET</td>
</tr>
<tr>
<td>BUBBLE</td>
<td>Perform one sweep of pointers left to right</td>
<td>ACT, BSTEP</td>
</tr>
<tr>
<td>RESET</td>
<td>Move both pointers all the way left</td>
<td>LSHIFT</td>
</tr>
<tr>
<td>BSTEP</td>
<td>Conditionally swap and advance pointers</td>
<td>COMPSWAP, RSHIFT</td>
</tr>
<tr>
<td>COMPSWAP</td>
<td>Conditionally swap two elements</td>
<td>ACT</td>
</tr>
<tr>
<td>LSHIFT</td>
<td>Shift a specified pointer one step left</td>
<td>ACT</td>
</tr>
<tr>
<td>RSHIFT</td>
<td>Shift a specified pointer one step right</td>
<td>ACT</td>
</tr>
<tr>
<td>ACT</td>
<td>Swap two values at pointer locations or move a pointer</td>
<td>-</td>
</tr>
</tbody>
</table>

(a) Example scratch pad and pointers used for sorting. Several steps of the BUBBLE subprogram are shown.

(b) Excerpt from the trace of the learned bubblesort program.

**Figure:** Illustration of the sorting environment
**Task:** learn a visual program that canonicalizes the model with respect to its pose
Canonicalizing 3D Models

- **Task**: learn a visual program that canonicalizes the model with respect to its pose
- Nontrivial problem: different starting positions and different car models
Task: learn a visual program that canonicalizes the model with respect to its pose

Nontrivial problem: different starting positions and different car models

\[ f_{enc}(Q, x, i_1, i_2, a_t) = MLP([Q(1, i_1), Q(2, i_2), f_{CNN}(x), a_t(1), a_t(2), a_t(3)]) \]

\( x \in \mathbb{R}^{H \times W \times 3} \) is the car rendering and \( Q \in \mathbb{R}^{2 \times 1 \times K} \) is the scratch pad, first dimension of \( Q \) corresponds to \( i_1, i_2 \) (fixed at 1) which are the pointer locations of the azimuth and elevation and \( K(=24) \) is the one-hot encoding dimension of pose coordinates.
Canonicalizing 3D Models

<table>
<thead>
<tr>
<th>GOTO</th>
<th>Change 3D car pose to match the target</th>
<th>HGOTO, VGOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>HGOTO</td>
<td>Move horizontally to the target angle</td>
<td>LGOTO, RGOTO</td>
</tr>
<tr>
<td>LGOTO</td>
<td>Move left to match the target angle</td>
<td>ACT</td>
</tr>
<tr>
<td>RGOTO</td>
<td>Move right to match the target angle</td>
<td>ACT</td>
</tr>
<tr>
<td>VGOTO</td>
<td>Move vertically to the target elevation</td>
<td>UGOTO, DGOTO</td>
</tr>
<tr>
<td>UGOTO</td>
<td>Move up to match the target elevation</td>
<td>ACT</td>
</tr>
<tr>
<td>DGOTO</td>
<td>Move down to match the target elevation</td>
<td>ACT</td>
</tr>
<tr>
<td>ACT</td>
<td>Move camera 15° up, down, left or right</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure: canonicalization of several different test set cars
Sample Complexity on Bubble Sort Problem

- Memory requirements is reduced from $O(n^2)$ to $O(n)$ thanks to compositional structure of the model.
Memory requirements is reduced from $O(n^2)$ to $O(n)$ thanks to compositional structure of the model.

Number of required training samples are also reduced:

**Figure:** Test accuracy by the varying sample complexity
Generalization on Bubble Sort Problem

- Training the model with variable-sized input (single-digit numbers from length 2 to length 20)
Generalization on Bubble Sort Problem

- Training the model with variable-sized input (single-digit numbers from length 2 to length 20)
- Adding a third pointer that acts as a counter to handle variable-sized inputs
Generalization on Bubble Sort Problem

- Training the model with variable-sized input (single-digit numbers from length 2 to length 20)
- Adding a third pointer that acts as a counter to handle variable-sized inputs
- Checking the success of the model on the inputs of previously unseen size to check how much the problem is learned
Generalization on Bubble Sort Problem

- Training the model with variable-sized input (single-digit numbers from length 2 to length 20)
- Adding a third pointer that acts as a counter to handle variable-sized inputs
- Checking the success of the model on the inputs of previously unseen size to check how much the problem is learned

Figure: Strong vs. weak generalization
Generalization on 3D Canonicalization Problem

- NPI is able to canonicalize cars of varying appearance from multiple starting positions
Generalization on 3D Canonicalization Problem

- NPI is able to canonicalize cars of varying appearance from multiple starting positions
- NPI can generalize to car appearances not encountered in the training
Generalization on 3D Canonicalization Problem

- NPI is able to canonicalize cars of varying appearance from multiple starting positions
- NPI can generalize to car appearances **not encountered in the training**

**Figure:** canonicalization of several different test set cars
Learning New Programs with a Fixed Core

- Fixing all the weights of core routing module
Learning New Programs with a Fixed Core

- Fixing all the weights of core routing module
- Only updating memory slots of the new programs
Prevent Existing Programs from Calling Subsequently Added Programs

- Looking back at the training data for known programs
Prevent Existing Programs from Calling Subsequently Added Programs

- Looking back at the training data for known programs
- Allowing addition of new programs
Solving Multiple Tasks with a Single Network

<table>
<thead>
<tr>
<th>Task</th>
<th>Single</th>
<th>Multi</th>
<th>+ Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>100.0</td>
<td>97.0</td>
<td>97.0</td>
</tr>
<tr>
<td>Sorting</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Canon. seen car</td>
<td>89.5</td>
<td>91.4</td>
<td>91.4</td>
</tr>
<tr>
<td>Canon. unseen</td>
<td>88.7</td>
<td>89.9</td>
<td>89.9</td>
</tr>
<tr>
<td>Maximum</td>
<td>-</td>
<td>-</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Per-sequence % accuracy

- NPI learns MAX perfectly **without forgetting the other tasks**
Solving Multiple Tasks with a Single Network

<table>
<thead>
<tr>
<th>Task</th>
<th>Single</th>
<th>Multi</th>
<th>+ Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>100.0</td>
<td>97.0</td>
<td>97.0</td>
</tr>
<tr>
<td>Sorting</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Canon. seen car</td>
<td>89.5</td>
<td>91.4</td>
<td>91.4</td>
</tr>
<tr>
<td>Canon. unseen</td>
<td>88.7</td>
<td>89.9</td>
<td>89.9</td>
</tr>
<tr>
<td>Maximum</td>
<td>-</td>
<td>-</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Per-sequence % accuracy

- NPI learns MAX perfectly **without forgetting the other tasks**
- One multi-task NPI can learn all three programs with comparable accuracy compared to each single-task NPI
Neural Programmer-Interpreters (NPI)

- learns several programs by using a **single core model**
Conclusion

Neural Programmer-Interpreters (NPI)
- learns several programs by using a **single core model**
- reduces sample complexity
Neural Programmer-Interpreters (NPI)

- learns several programs by using a **single core model**
- reduces sample complexity
- provides strong generalization
Neural Programmer-Interpreters (NPI)

- learns several programs by using a **single core model**
- reduces sample complexity
- provides strong generalization
- works for dissimilar environments
Neural Programmer-Interpreters (NPI)

- learns several programs by using a **single core model**
- reduces sample complexity
- provides strong generalization
- works for dissimilar environments
- learns new programs without forgetting already learned ones
Thank you!