NEURAL PROGRAMMER-INTERPRETERS [1]

Presented by

Fethiye Irmak Doğan

[1] Reed, S., and De Freitas, N. (2015). Neural programmer-interpreters. arXiv.

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 (NPI)

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**)

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

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

Persistant 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.

[2] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In Proceedings of the 26th annual international conference on machine learning, pages 41-48

Curriculum Learning[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

[2] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In Proceedings of the 26th annual international conference on machine learning, pages 41-48

Related Work

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

NPI Core acts as a **router** between programs and there is a single inference core shared by arbitrary programs

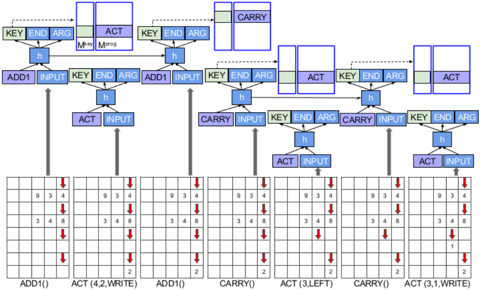


Figure: Example execution trace of single-digit addition

NPI Core

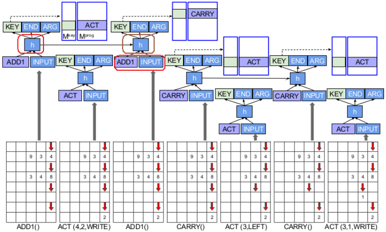


Figure: Example execution trace of single-digit addition

NPI Core is conditioned on

– current state observations:

NPI Core

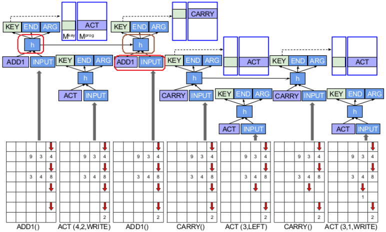


Figure: Example execution trace of single-digit addition

NPI Core is conditioned on

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

NPI Core

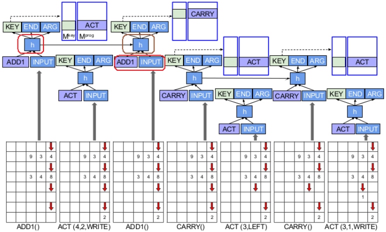


Figure: Example execution trace of single-digit addition

NPI Core is conditioned on

- current state observations:
  - learnable program embedding, program arguments, feature representation of the environment
- previous hidden unit states

NPI Core

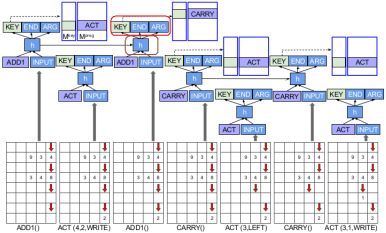


Figure: Example execution trace of single-digit addition

NPI Core outputs

- key indicating what program to call next

NPI Core

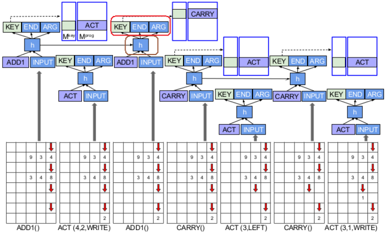


Figure: Example execution trace of single-digit addition

NPI Core outputs

- key indicating what program to call next
- probability of ending the current program

NPI Core

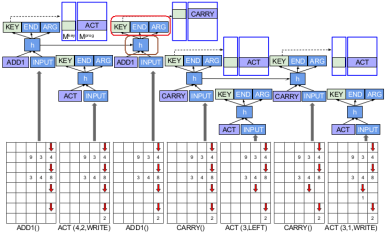


Figure: Example execution trace of single-digit addition

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)

Program Embedding Memory

Different programs correspond to different embeddings stored in a persistent memory

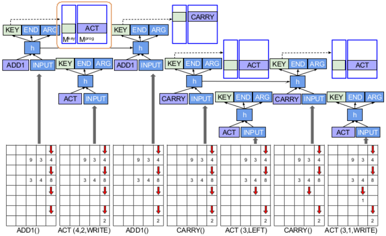


Figure: Example execution trace of single-digit addition

Feed-Forward steps of program inference

$\mathbf{e_t}$: environment observation at time t $\mathbf{a_t}$: current program argument $\mathbf{s_t}$: state encoding
$\mathbf{p_t}$: program embedding $\mathbf{h_{t-1}}$: previous hidden unit $\mathbf{c_{t-1}}$: previous cell unit
$\mathbf{r_t}$:end of program probability $\mathbf{k_t}$: program key embedding $\mathbf{a_t}$:output arguments at time t
$\mathbf{f_{enc}}$: domain specific encoder $\mathbf{f_{lstm}}$: LSTM mapping
$\mathbf{f_{end}}$: probability of finishing the program $\mathbf{f_{prog}}$: key embedding for next program $\mathbf{f_{arg}}$: arguments to next program

$$s_t = f_{enc}(e_t, a_t)$$

Feed-Forward steps of program inference

$\mathbf{e_t}$: environment observation at time t $\mathbf{a_t}$: current program argument $\mathbf{s_t}$: state encoding
$\mathbf{p_t}$: program embedding $\mathbf{h_{t-1}}$: previous hidden unit $\mathbf{c_{t-1}}$: previous cell unit
$\mathbf{r_t}$:end of program probability $\mathbf{k_t}$: program key embedding $\mathbf{a_t}$:output arguments at time t
$\mathbf{f_{enc}}$: domain specific encoder $\mathbf{f_{lstm}}$: LSTM mapping
$\mathbf{f_{end}}$: probability of finishing the program $\mathbf{f_{prog}}$: key embedding for next program $\mathbf{f_{arg}}$: arguments to next program

$$s_t = f_{enc}(e_t, a_t)$$

$$h_t = f_{lstm}(s_t, p_t, h_{t-1})$$

Feed-Forward steps of program inference

$\mathbf{e_t}$: environment observation at time t $\mathbf{a_t}$: current program argument $\mathbf{s_t}$: state encoding
$\mathbf{p_t}$: program embedding $\mathbf{h_{t-1}}$: previous hidden unit $\mathbf{c_{t-1}}$: previous cell unit
$\mathbf{r_t}$:end of program probability $\mathbf{k_t}$: program key embedding $\mathbf{a_t}$:output arguments at time t
$\mathbf{f_{enc}}$: domain specific encoder $\mathbf{f_{lstm}}$: LSTM mapping
$\mathbf{f_{end}}$: probability of finishing the program $\mathbf{f_{prog}}$: key embedding for next program $\mathbf{f_{arg}}$: arguments to next program

$$s_t = f_{enc}(e_t, a_t)$$

$$h_t = f_{lstm}(s_t, p_t, h_{t-1})$$

$$r_t = f_{end}(h_t),\ k_t = f_{prog}(h_t),\ a_{t+1} = f_{arg}(h_t)$$

Program Embedding

$\mathbf{k_t}$: program key embedding **i:** program ID $\mathbf{p_{t+1}}$: next program embedding
$\mathbf{M^{key}}$: key embeddings which stores all the program keys $\mathbf{M^{prog}}$: program embeddings

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

Environmental State

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

$$e_{t+1} \sim f_{env}(e_t, p_t, a_t)$$

Inference Algorithm

Algorithm 1 Neural programming inference

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

Inference Algorithm

Algorithm 1 Neural programming inference

Inputs: Environment observation e, program id i, arguments a, stop threshold α
function RUN(i, a)
  $h \leftarrow \mathbf{0}, r \leftarrow 0, p \leftarrow M^{\text{prog}}_{i,:}$ ▷ Init LSTM and return probability.
  while $r < \alpha$ **do**
    $s \leftarrow f_{enc}(e, a), h \leftarrow f_{lstm}(s, p, h)$ ▷ Feed-forward NPI one step.
    $r \leftarrow f_{end}(h), k \leftarrow f_{prog}(h), a_2 \leftarrow f_{arg}(h)$
    $i_2 \leftarrow \arg\max_{j=1..N} (M^{\text{key}}_{j,:})^T k$ ▷ Decide the next program to run.
    if $i ==$ ACT **then** $e \leftarrow f_{env}(e, p, a)$ ▷ Update the environment based on ACT.
    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

ε_t^{inp} : $\{e_t, i_t, a_t\}$ and ε_t^{out} : $\{i_{t+1}, a_{t+1}, r_t\}$ are the execution traces

Training

ε_t^{inp} : $\{e_t, i_t, a_t\}$ and ε_t^{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^{key} M^{prog} of the programs to run at time t and t+1

Training

ε_t^{inp} : $\{e_t, i_t, a_t\}$ and ε_t^{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^{key} M^{prog} of the programs to run at time t and t+1

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

Training

ε_t^{inp} : $\{e_t, i_t, a_t\}$ and ε_t^{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^{key} M^{prog} of the programs to run at time t and t+1

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

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

Training

ε_t^{inp} : $\{e_t, i_t, a_t\}$ and ε_t^{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^{key} M^{prog} of the programs to run at time t and t+1

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

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

$$\log P(\xi_{out}|\xi_{inp};\theta) = \sum_{t=1}^{T} \log P(\xi_t^{out}|\xi_1^{inp},...,\xi_t^{inp};\theta)$$

Training

ε_t^{inp} : $\{e_t, i_t, a_t\}$ and ε_t^{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^{key} M^{prog} of the programs to run at time t and t+1

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

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

$$\log P(\xi_{out} | \xi_{inp}; \theta) = \sum_{t=1}^{T} \log P(\xi_t^{out} | \xi_1^{inp}, ..., \xi_t^{inp}; \theta)$$

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

Training

ε_t^{inp} : $\{e_t, i_t, a_t\}$ and ε_t^{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^{key} M^{prog} of the programs to run at time t and t+1

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

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

$$\log P(\xi_{out}|\xi_{inp};\theta) = \sum_{t=1}^{T} \log P(\xi_t^{out}|\xi_1^{inp},...,\xi_t^{inp};\theta)$$

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

$$\log P(\xi_t^{out}|\xi_1^{inp},...,\xi_t^{inp}) = \log P(i_{t+1}|h_t) + \log P(a_{t+1}|h_t) + \log P(r_t|h_t)$$

Training

- **Adaptive curriculum :** sample frequency of a program is determined by model's current prediction error in that program

Training

- **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

Training

- **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

Addition

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

[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

[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)

[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)

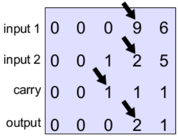
$$f_{enc}(Q, i_1, i_2, i_3, i_4, a_t) = MLP([Q(1, i_1), Q(2, i_2), Q(3, i_3), Q(4, i_4), a_t(1), a_t(2), a_t(3)])$$

$Q \in R^{4xNxK}$ 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

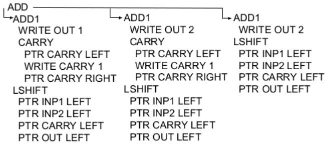
[3] https://en.wikipedia.org/wiki/One-hot

Addition

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



(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

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

Sorting

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

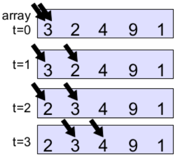
$$f_{enc}(Q, i_1, i_2, a_t) = MLP([Q(1, i_1), Q(1, i_2), a_t(1), a_t(2), a_t(3)])$$

Q $\in R^{1xNxK}$ 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

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



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

```
BUBBLESORT
 BUBBLE                RESET        ...   BUBBLE        ...
   PTR 2 RIGHT         LSHIFT               PTR 2 RIGHT
   BSTEP                 PTR 1 LEFT         BSTEP
     COMPSWAP            PTR 2 LEFT           COMPSWAP
       SWAP 1 2        LSHIFT                   SWAP 1 2
     RSHIFT              PTR 1 LEFT           RSHIFT
       PTR 1 RIGHT       PTR 2 LEFT             PTR 1 RIGHT
       PTR 2 RIGHT     ...                      PTR 2 RIGHT
   ...                 LSHIFT               ...
   BSTEP                 PTR 1 LEFT         BSTEP
     COMPSWAP            PTR 2 LEFT           COMPSWAP
     RSHIFT                                   RSHIFT
       PTR 1 RIGHT                              PTR 1 RIGHT
       PTR 2 RIGHT                              PTR 2 RIGHT
```

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

Figure: Illustration of the sorting environment

Canonicalizing 3D Models

- **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

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

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

x $\in R^{HxWx3}$ is the car rendering and Q $\in R^{2x1xK}$ 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

GOTO	Change 3D car pose to match the target	HGOTO, VGOTO
HGOTO	Move horizontally to the target angle	LGOTO, RGOTO
LGOTO	Move left to match the target angle	ACT
RGOTO	Move right to match the target angle	ACT
VGOTO	Move vertically to the target elevation	UGOTO, DGOTO
UGOTO	Move up to match the target elevation	ACT
DGOTO	Move down to match the target elevation	ACT
ACT	Move camera 15° up, down, left or right	-

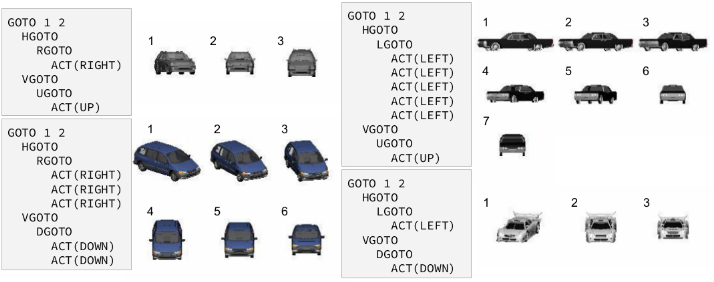


Figure: canonicalization of several different test set cars

Sample Complexity on Bubble Sort Problem

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

Sample Complexity on Bubble Sort Problem

- Memory requirements is reduced from $\mathbf{O}(\mathbf{n}^2)$ to $\mathbf{O}(\mathbf{n})$ thanks to compositional structure of the model
- Number of required training samples are also reduced:

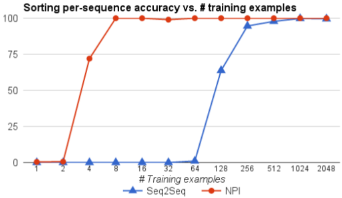


Figure: Test accuracy by the varying sample complexity

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

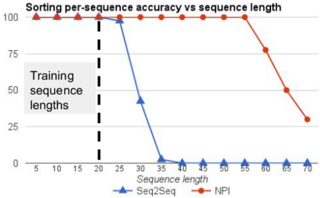


Figure: Strong vs. weak generalization

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**

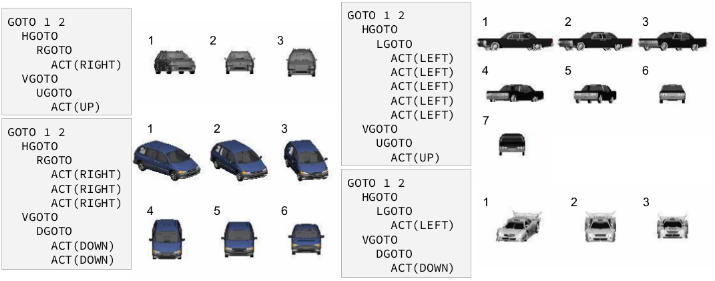


Figure: canonicalization of several different test set cars

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

Task	Single	Multi	+ Max
Addition	100.0	97.0	97.0
Sorting	100.0	100.0	100.0
Canon. seen car	89.5	91.4	91.4
Canon. unseen	88.7	89.9	89.9
Maximum	-	-	100.0

Per-sequence % accuracy

- NPI learns MAX perfectly **without forgetting the other tasks**

Solving Multiple Tasks with a Single Network

Task	Single	Multi	+ Max
Addition	100.0	97.0	97.0
Sorting	100.0	100.0	100.0
Canon. seen car	89.5	91.4	91.4
Canon. unseen	88.7	89.9	89.9
Maximum	-	-	100.0

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

Conclusion

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

Conclusion

Neural Programmer-Interpreters (NPI)

- learns several programs by using a **single core model**
- reduces sample complexity
- provides strong generalization

Conclusion

Neural Programmer-Interpreters (NPI)

- learns several programs by using a **single core model**
- reduces sample complexity
- provides strong generalization
- works for dissimilar environments

Conclusion

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!