IMPROVING THE UNIVERSALITY AND LEARNABILITY OF NEURAL PROGRAMMER-INTERPRETERS WITH COMBINATOR ABSTRACTION

Anonymous authors
Paper under double-blind review

ABSTRACT

To overcome the limitations of Neural Programmer-Interpreters (NPI) in its universality and learnability, we propose the incorporation of combinator abstraction into neural programing and a new NPI architecture to support this abstraction, which we call Combinatory Neural Programmer-Interpreter (CNPI). Combinator abstraction can dramatically reduce the number and complexity of programs that need to be interpreted by the core controller of CNPI, while still allowing the CNPI to represent and interpret arbitrary complex programs by the collaboration of the core with the other components. We propose a small set of four combinators to capture the most pervasive programming patterns. Due to the finiteness and simplicity of this combinator set and the offloading of some burden of interpretation from the core, we are able construct a CNPI that is is universal with respect to the set of all combinatorizable programs, which is adequate for solving most algorithmic tasks. Moreover, it is possible to train the CNPI by policy gradient reinforcement learning with an appropriately designed curriculum.

1 INTRODUCTION

Teaching machines to learn programs is a challenging task. Numerous models have been proposed for learning programs, e.g. Neural Turing Machine (Graves et al. (2014)), Differentiable Neural Computer (Graves et al. (2016)), Neural GPU (Kaiser & Sutskever (2015)), Neural Programmer (Neelakantan et al. (2015)), Hierarchical Attentive Memory (Andrychowicz & Kurach (2016)), Neural Random Access Machine (Kurach et al. (2015)) and Neural Programmer-Interpreter (Reed & De Freitas (2016)). These models are usually equipped with some form of memory components with differentiable access. Most of these models are trained on program input-output pairs and the neural network effectively learns to become the particular target program, mimicking a particular Turing machine.

Of these models one notable exception is Neural Programmer-Interpreters (NPI) (Reed & De Freitas (2016)) and its extension that supports recursion (Cai et al. (2017)) (referred to in this paper as RNPI). NPI has three components: a core controller that is typically implemented by a recurrent neural network, a program memory that stores embeddings of learned programs, and domain-specific encoders that enable a single NPI to operate in diverse environments. Instead of learning to become any particular program, the core module learns to *interpret* arbitrary programs represented as program embeddings. This integration of the core (interpreter) and a learned program memory (programmer) mimics a universal Turing machine and offers NPIs with better flexibility and composability by allowing the model to learn new programs by combining subprograms. Despite these merits, the NPI model bears some theoretical and practical limitations that hinder its application in real world problems.

One hypothetical theoretical property of the NPI model that makes it appealing for multi-task, transfer, and life-long learning settings is its *universality*, i.e. the capability to represent and interpret *any* program. As the NPI relies solely on the core to interpret programs, universality requires a fixed core to interpret potentially many programs. A universal fixed core is critical for learning and re-using learned programs in a continual manner, because a core with changing weights may fail to interpret old learned programs after learning new ones. Although the original NPI paper shows empirically

that a single shared core can interpret 21 programs to solve five tasks, and that a trained NPI with fixed core can learn a new simple program MAX, it is unclear whether a universal NPI exists or how universal it could be. Specifically, given the infinite set of all possible programs, the subset of programs that can be interpreted by a fixed core is not explicitly defined. Even though a universal NPI exists, it may still intractable to provable guarantee of universality by the verification method proposed in Cai et al. (2017), because there may be infinite programs to verify.

Practically, as proposed in Reed & De Freitas (2016), the training of an NPI model relies on a very strong form of supervision, i.e. the example execution traces of programs. This form of training data is typically more costly to obtain than input-output examples. Training with weaker form of supervision is desirable to unlock the NPI model's full potential.

In this paper, we propose to overcome these limitations of NPIs by incorporating combinator abstraction into the NPI model and augmenting the original NPI architecture with necessary components and mechanisms to support this abstraction. We refer to this new architecture as the Combinatory Neural Programmer-Interpreter (CNPI). Combinators, a.k.a. higher-order functions are import abstraction techniques in functional programming that are used to express some common programming patterns shared across different programs. We find that combinator abstraction can dramatically reduce the number and complexity of programs (i.e. combinators) that need to be interpreted by the core, while still allowing the CNPI to represent and interpret arbitrary complex programs by the collaboration of the core with the other components. We propose a small set of four combinators to capture the most pervasive programming patterns and demonstrate how to compose combinatory programs with these combinators to solve common algorithmic tasks. Due to the finiteness and simplicity of this combinator set and the offloading of some burden of interpretation from the core, we are able construct a CNPI with a fixed core that can represent and interpret an infinite set of programs which is adequate for solving most algorithmic tasks. This CNPI is universal with respect to the set of all combinatorizable programs. Moreover, we show empirically that besides supervised training on execution traces, it is possible to train the CNPI by policy gradient reinforcement learning with an appropriately designed curriculum.

2 Overview of Combinator Abstraction

2.1 Review of NPI with its Limitations

In this section, we give a brief review of the NPI architecture from Reed & De Freitas (2016) and Cai et al. (2017). Then we analyze its limitations to motivate our combinator abstraction. The NPI model has three learnable components: a task-agnostic core controller, a program memory, and domain-specific encoders that allow the NPI to operate in diverse environments. The core controller is a long short-term memory (LSTM) network (Hochreiter & Schmidhuber (1997)) that acts as a router between programs. At each time step, the core can decide either to select another programs to call with certain arguments, or to end the current program. When the program returns, control is returned to the caller by popping the callers LSTM hidden units and program embedding off of a program call stack and resuming execution in this context.

A description of NPI's inference procedure is given in Algorithm 1. It is adapted from Reed & De Freitas (2016) with slight modification for clarity and ease of comparison with our inference procedure int Algorithm 2. At time step t, an encoder f_{enc} takes in the environment observation e_t and arguments a_t and generates a state s_t. The core LSTM f_{lstm} takes in the state s_t, a program embedding $p_t \in \mathbb{R}^P$ and the previous hidden state h_{t-1} to update its hidden state h_t. From the top LSTM hidden state several decoders generate the outputs. The return probability r_t is computed by f_{end}. The next program's key embedding k_{t+1} is computed by f_{prog}. The arguments to the next program $a_{t+1}$are generated by f_{arg}. The outputs r_t, k_{t+1} and a_{t+1} are used to determine the next action, as described in Algorithm 1 The next program's embedding is obtained by comparing the key embedding k_{t+1} to each row of key memory M_{key}. Then the program embedding is retrieved from program memory M_{prog} as follows:

$$i^* = \underset{j=1..N}{\arg\max}(M_j^{key})^T k_{t+1} \ , \ p_{t+1} = M_{i^*}^{prog} \tag{1}$$

where N is the current number of programs in memory.

Algorithm 1 Neural programming inference

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

The above-described NPI architecture bears two limitations. First, as shown in Algorithm 1 and equation (1), at each time step, a decision must be made by the core to select the next program to call out of *all* N currently learned programs in the program memory. As N grows large, e.g. to hundreds or thousands, interpreting programs correctly becomes a more and more difficult task for a single core. What makes things worse is that the core has to learn to interpreting new programs without forgetting old ones. Second, it it common that programs with different names and functionalities share some common underlying programming patterns. We take two programs used in Reed & De Freitas (2016) and Cai et al. (2017) as example (also see Figure 1): the ADD1 program in grade school addition, and the BSTEP program to perform one pass of bubble sort. Their non-recursive forms are first described in Figure 3 and 4 of Reed & De Freitas (2016). Cai et al. (2017) describes their recursive form in Figure 1 and 2 of their paper. The two program share a very common looping pattern: If there are still items left (A[P2] $\neq$ END), first do some operation on the current item (ADD1 or COMPSWAP), then move to the next item (LSHIFT or RSHIFT), and finally recursively call itself to repeat the process (ADD or BSTEP) [1]. However, the core needs to learn each of these programs separately without taking any advantage of their shared patterns. The total number of programs that need to be learned by the core thus become infinite. We argue that these two limitations make it very challenging, if not impossible, to construct a universal NPI.

2.2 Our approach using combinator abstraction

To overcome the limitations of the NPI, we propose to incorporate combinator abstraction into the NPI architecture. In functional programming, combinators are a special kind of higher-order functions that server as power abstraction mechanisms, increasing the expressive power of programming languages. We adapt the concept to neural programming and make it play the central role in improving the universality of NPI.

Conceptually, a *combinator* is a "program template" with blanks as *formal* arguments that are callable as subprograms. An actual program can be formed by wrapping a combinator with another program called an *applier*, which invokes the combinator and passes the *actual* arguments to be called when executing the combinator. Alternatively, an applier applies a combinator to a set of actual programs as callable arguments. Note that the callable arguments themselves can also be wrapped programs (i.e. appliers), and programs with increasing complexity can thus be built up. As in the original NPI, the interpretation of a combinator is conditioned on the output of a lightweight domain-specific encoder which we call an *detector*. It is also provided on the fly by the applier. Figure 1 illustrates the usage of combinator abstraction in the NPI architecture.

In the CNPI architecture, combinators are the only type of programs that need to be interpret by the core. By prohibiting a combinator to call programs other than those passed to it as arguments, the selection range for next program to call at each time step is reduced from a growing N to a constant K, which is the maximum number of arguments for combinators (le 9 in our proposed model). Meanwhile, compared to the infinity of all possible programs, the number of useful combinators is finite and typically small. In practice, we construct a small set of four combinators to express four most pervasive programming patterns. Therefore, the core only needs to interpret a small number

[1] In this paper we always use the recursive version of looping programs, as proposed in Cai et al. (2017)

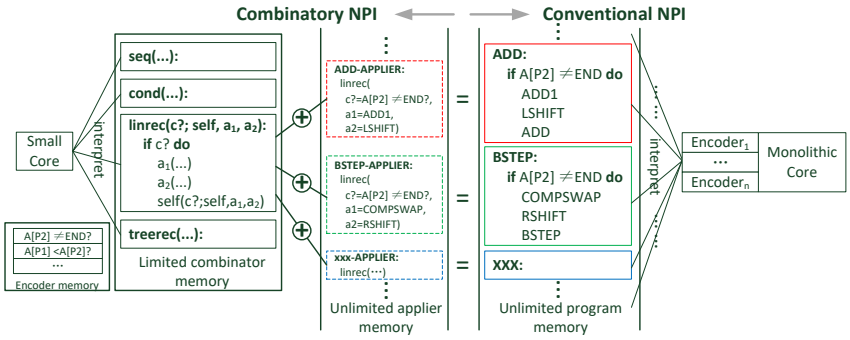


Figure 1: NPI with combinator abstraction.

```
# sequential pattern
def seq(c?; self,
    a1, a2, a3):
    a1()
    a2()
    a3()
```

```
# conditional pattern
def cond(c?; self,
    a1, a2, a3):
   if  c?():
      a1()
      a2()
   else:
      a3()
```

```
# linear recursion pattern
def linrec(divisible?/c?; self,
      do/a1, next/a2, base/a3):
  if  divisible?():
      do()
      next()
      self(divisible?; self,
           do, next, base)
  else:
      base()
```

```
# tree recursion pattern
def treerec(divisible?/c?; self,
      pre/a1, divide/a2, post/a3):
  if divisible?():
      pre()
      _push_sentinel
      divide()
      _mapself(divisible?; self,
                pre, divide, post)
      post()
```

Figure 2: Pseudo-code for the set of combinators.

of simple programs. We will show that a quite small core suffices for this job, and that by the collaboration of this core and the other components a universal CNPI can be constructed.

3 Combinatory NPI Model

3.1 Combinators and Combinatory Programs

We propose a set of four combinators to express four most pervasive programming patterns: sequential, conditional, linear recursion and tree recursion (i.e. multi-recursion). The pseudo-code for these combinator are shown in Figure 2. Each combinator has four callable arguments $self$, $a1$, $a2$ and $a3$ and one detector argument $c?$. $self$ is a *default* argument referring to the combinator itself and is used for recursive call. For $linrec$ and $treerec$, we give more readable aliases to $a1$, $a2$ and $a3$ to hint their typical roles. The detector argument provides signals for the combinator to condition its execution on. A detector detects a certain condition (e.g. a pointer P2 reaching the end of array) in the environment. It outputs 0 if the condition satisfies, otherwise 1. For seq a default blind detector is passed, which always outputs 0. Although not directly callable, detectors can also be viewed as a special kind of programs that act as perception modules. We append a '?' in the name of detectors to differentiate them from callable programs. In this paper, the condition to detect is often used to name the detector. Like primitive actions, we could also define primitive detectors (DETs) for specific tasks. Note that this combinator set is by no means unique or minimal, and the proposed combinators are not all in their simplest forms. In fact, two callable arguments are sufficient for expressing the sequential and conditional patterns. They take their current forms mainly for ease of use and learning.

The four combinators are classified into two categories. seq, $cond$ and $linrec$ are basic combinators, which only call their callable arguments during execution. $treerec$ is an advanced combinator. Besides callable arguments, an advanced combinator can also call built-in programs, such as $_push_sentinel$ and $_mapself$ in $treerec$. These built-in programs are used to facilitate multi-

```
def COMPSWAP:
  if A[P1]>A[P2]:
    SWAP_12
def RSHIFT:
  P1_RIGHT
  P2_RIGHT
def BSTEP:
  if A[P2]≠END:
    COMPSWAP
    RSHIFT
    BSTEP
```

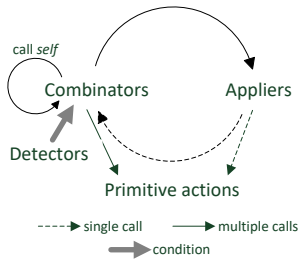
(a) Normal program.

```
def COMPSWAP:
  cond(A[P1]>A[P2]?;
       SWAP_12, NOP, NOP)
def RSHIFT:
  seq(; P1_RIGHT, P2_RIGHT)
def BSTEP:
  linrec(A[P2]≠END?;
       COMPSWAP, RSHIFT, NOP)
```

(b) Combinatory program.

```
BSTEP
 linrec(..)
  a1->COMPSWAP
   cond(..)
    a1->SWAP_12
  a2->RSHIFT
   seq(..)
    a1->P1_RIGHT
    a2->P2_LEFT
  self->linrec(..)
   a1->COMPSWAP
   ...
```

(c) Trace of combinatory program.



(d) Interaction between programs and detectors.

Figure 3: Example combinatory program of BSTEP. NOP is special ACT which does nothing.

ple recursive calls to $self$ in $treerec$ combinator. Basically, $divide$ prepares states necessary for each recursive call and push these states to a stack. The built-in combinator $_mapself$ shares a similar structure with $linrec$. It loads the states one by one from the stack and makes the recursive call with each state until a sentinel is met (The sentinel is pushed to the stack before $divide$ by $_push_sentinel$, which is a built-in primitive action). More details on built-in programs and $treerec$ are given in Appendix 7.1, and examples of using them to compose the topological sort and quick sort programs can be found in Appendix 7.3.

We now describe how to compose combinator programs using combinators by taking the BSTEP program (i.e. one pass of bubble sort) as example. The normal and combinatory version of the program are shown in Figure 3 (a) and (b) respectively. Recall that an applier applies a combinator to a set of actual programs (primitive actions or other predefined appliers) to form a new actual program. Composing a combinatory program amounts to defining appliers iteratively. As shown in Figure 3 (c), during the execution of a combinatory programs, combinators and appliers call each other and form an alternating call sequence until reaching a primitive action (ACT). Combinators, appliers and detectors are all highly constrained programs, and thus are all easily interpretable and learnable. Nevertheless, they can collaborate to build arbitrarily complex programs.

3.2 CNPI ARCHITECTURE AND ALGORITHM

Having introduced combinators and how to use them to compose combinatory programs, we now describe how these programs are interpreted by the CNPI and the necessary augmentations to the original NPI architecture to enable the interpretation. The complete inference procedure for CNPI is given in Algorithm 2. An example execution of the BSTEP program is illustrated in Figure 4.

Appliers are effectively one-line programs that apply a program $prog$, which could be either a combinator or an ACT, to a set of arguments. To interpret an applier $appl$ we just need to identify the program to be called and its arguments, prepare environment for the invocation, and make the invocation. For easy of interpretation, we propose to store the key embeddings of $prog$ and its detector and callable arguments $c?$, $a1$, $a2$ and $a3$ directly in the applier's program embedding p_{appl}:

$$p_{appl} = k_{prog}|k_{c?}|k_{a1}|k_{a2}|k_{a3} \tag{2}$$

where $|$ denotes concatenation [2]. We use a parser to extract the key embeddings from the applier's embedding. Then the combinator ID i is computed by comparing the key k_{comb} to each row of memory M_{key} and finding the best match. The callable arguments' IDs are computed similarly. In CNPI architecture, the models for detectors are stored in a detector memory (W^{key}, W^{weight}) which has the same key-value structure as the program memory. The detector argument ID i' is computed by comparing $k_{c?}$ to each row of W_{key}. Note that the core LSTM does not participate in the interpretation of appliers. As the format for storing these key embeddings is predefined, a fixed parser can parse *any* applier's embedding.

[2]For the seq combinator and ACTs which do not need detector arguments, the blind detector's key embedding is stored. For ACTs which need arguments we just store the arguments' values in place of the key embeddings.

After the IDs of arguments are computed, they need to be passed to the combinator before it is called. We use *frames* to pass arguments to combinators. Each frame is a table of K bindings which associate formal callable arguments with their corresponding actual IDs, with K the number of callable arguments for combinators. When calling a combinator, a new frame is created. The IDs of the combinator's callable arguments (including the combinator's ID i as it corresponds to the $self$ argument) are filled into the frame [3]. As the number of arguments for a combinator is fixed, in practice we do not use a key-value structure for frames. Instead the frame only stores values, i.e. the arguments' IDs in a fixed order of $self$, $a1$, $a2$ and $a3$. Once the frame has been prepared, the combinator is called.

The interpretation of combinators is in general similar to the inference procedure in Algorithm 1. Here we highlight three key differences. The first difference is in the initialization stage. Besides retrieving combinator embedding from the program memory, the detector model is also loaded from the detector memory. Then instead of using the program embedding as input to the LSTM at every time step, we use the combinator embedding to initialize the LSTM's state, i.e. each layer's hiddens and cells. We find that this parameterization has better efficiency and accuracy for our combinators, as is shown in Section 5.1. The second difference is that we binarize the output of detector f_det to get a binary condition c before feeding it to the LSTM:

$$c \leftarrow \mathbb{1}(f_{det}(e) \geq \beta) \ , \ h \leftarrow f_{lstm}(c, h) \tag{3}$$

where $\mathbb{1}()$ is an indicator function. This operation effectively decouples the detector from the core LSTM. This enables us to verify the core's interpretation of combinators separately without considering any specific detectors, given that the correct condition is provided. This is difficult to achieve in the original NPI architecture because there the core and the program embedding are trained jointly with the encoders.

The third and most important difference is on how the next subprogram to call is computed. We use a decoder f_{prog} to compute a score vector $S \in \mathbb{R}^{\mathbb{K}}$ to assign a score for each formal callable argument. The argument with the maximum score is selected and its actual program ID is retrieved from the frame F. This ID is used in turn to retrieve the program embedding from the program memory M^{prog} when the next program is executed:

$$z^* = \underset{j=1..K}{\arg\max}\ S_j \ , \ i^* \leftarrow F[z^*] \ , \ p_{t+1} = M^{prog}_{i^*} \tag{4}$$

where K is the maximum number of callable arguments for combinators. We consider this indirection of subprogram embedding retrieval, together with the dynamic binding of formal arguments to actual programs in the frames, to be the key to the superior universality and learnability of CNPI.

When calling subprogram the same detector ID and frame are re-used, which is equivalent to passing the combinator's arguments to all of its subprograms. This facilitates recursion as these arguments are needed by $linrec$ and $treerec$ combinators when calling $self$ (see Figure 2). For other subprogram calls, which should be calls to either appliers or ACTs, these arguments are safely ignored. Another optimization we make, which is not described in Algorithm 2 but shown in Figure 4, is on the usage of the detector's output which we call *transient condition*: We allow the condition signal produced by the detector to take effect only at the first time step during a combinator's execution and set the condition to zero at subsequent steps. In other words, we require the branching of execution, if any, to happen only at the beginning the of execution. This conforms to conventional programming practice that a branching condition does not take effect once inside the branch, and we show empirically in Section 5.3 that it facilitates learning.

3.3 TRAINING

CNPI has four components: the core, the detector memory, the program (combinator and applier) memories, and the parser, of which the first three are learnable. The combinators are trained jointly with the core. Detectors and appliers are trained separately.

Supervised learning (SL) of CNPI uses execution traces of combinatory programs. A single element of an execution trace consist of a step input-step out pair, which takes one of the two forms: ξ^{comb}_t :

[3] If the combinator is an advanced one, i.e. $treerec$, the IDs of built-in programs also need to be appended to the end of the frame in some predefined order. In this case the frame's size is $K + B$, with B the total number of built-in programs.

Algorithm 2 Combinatory neural programming inference

Inputs: Environment observation e, program ID i, detector ID i', frame F, stop threshold α, condition threshold β, number of arguments (including $self$) K for combinators
function RUN(i, i', F)
  $r \leftarrow 0,\ p \leftarrow M_i^{prog}$
  if p is an applier **then**
    $i_2,\ i_2',\ a_2 \leftarrow$ _PARSE(p) ▷ Get the next program to run with its detector and args.
    $F_2 \leftarrow$ FRAME(K), $F_2[1] \leftarrow i_2$ ▷ New an empty frame and fill in $self$ arg.
    for $j = 1$ to $K - 1$ **do** $F_2[j+1] \leftarrow a_2[j]$ ▷ Fill in the other args.
    RUN(i_2, i_2', F_2) ▷ Run subprogram i_2 with detector i_2' and frame F_2.
    FREE(F_2) ▷ Free frame F_2's space.
  else if p is a combinator **then**
    $f_{det} \leftarrow W_{i'}^{weight},\ h \leftarrow p$ ▷ Load detector from detector memory and init LSTM.
    while $r < \alpha$ **do**
      $c \leftarrow \mathbb{1}(f_{det}(e) \geq \beta),\ h \leftarrow f_{lstm}(c, h)$ ▷ c is a binary condition.
      $r \leftarrow f_{end}(h),\ S \leftarrow f_{prog}(h)$ ▷ S is a K-dim score vector.
      $z_2 \leftarrow \arg\max_{j=1..K} S_j$ ▷ Decide the argument ID of the next program to run.
      $i_2 \leftarrow F[z_2]$ ▷ Retrieve the ID of the next program to run from frame.
      RUN(i_2, i', F) ▷ Run subprogram i_2 with the same detector and frame.
  else ▷ p is a primitive action.
    $a \leftarrow F[2:K],\ e \leftarrow f_{env}(e, p, a)$ ▷ Unpack args from F and do the action.

function _PARSE(p) ▷ Helper function for interpreting appliers.
  $k, k', a \leftarrow$ SPLIT(p) ▷ Get program, detector and arg keys from applier embedding.
  $i \leftarrow \arg\max_{j=1..N}(M_j^{key})^T k,\ i' \leftarrow \arg\max_{j=1..M}(W_j^{key})^T k'$ ▷ Program key to id.
  if i is not a primitive action **then**
    for $j = 1$ to $K - 1$ **do** $a[j] \leftarrow \arg\max_{j'=1..N}(M_{j'}^{key})^T a[j]$ ▷ Arg keys to IDs.
  return i, i', a

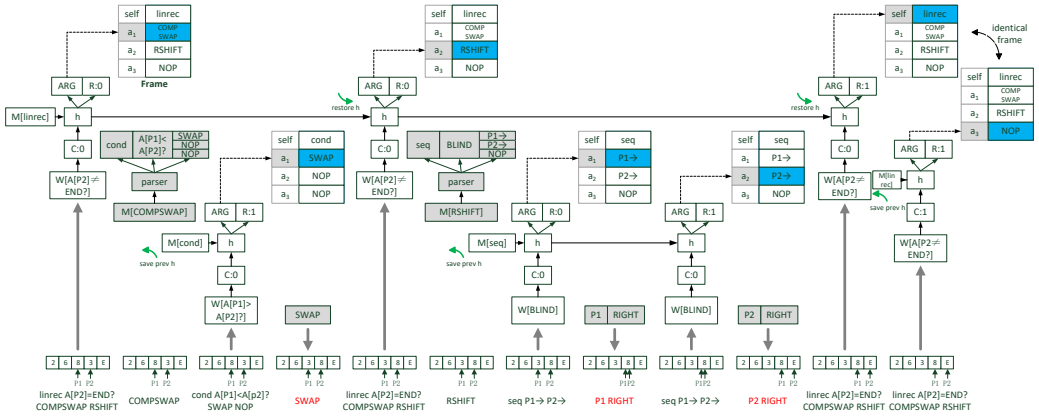


Figure 4: Example execution of the combinator BSTEP program. SWAP, P1_RIGHT and P2_RIGHT are in fact appliers which need to be parsed as COMPSWAP and RSHIFT. We omit this step and treat them as ACTs in the figure for brevity. Combinatory/applier embeddings M[..] and detector models W[..] are retrieved from the program memory and detector memory respectively, which are not shown. The core LSTM's hidden state is pushed to the program call stack only when a combinator is called. Appliers have no stack effect just as ACTs.

$\{e_t, i_t\} \rightarrow \{c_t, z_{t+1}\}$ for combinator execution and $\xi_t^{appl} : \{i_t\} \rightarrow \{i_{t+1}, i'_{t+1}, a_{t+1}\}$ for applier execution. z_{t+1} is the formal callable argument ID to be called by the applier at time step $t+1$. c_t is correct condition at time step t and is used as the output target for detectors. $it+1$ and r_t provide targets for the core. Detectors and the core are trained on the ξ_t^{comb} elements of the trace, using stochastic gradient ascent to maximize the likelihood of their corresponding targets.

$$\Delta w \propto \sum_{t=1}^{T} (\nabla_w \log p_w(c_t \mid e_t)) \ , \ \Delta\theta \propto \sum_{t=1}^{T} (\nabla_\theta \log p_\theta(i_{t+1} \mid c_t) + \nabla_\theta \log p_\theta(r_t \mid c_t)) \tag{5}$$

where w are parameters of the detector model, θ are the collective parameters of the core and the combinator embedding, T is the length of the sequence of ξ_t^{comb} elements. The probability $p(i_{t+1} \mid c_t)$ of calling subprogram i is computed by applying a softmax to the scores produced by f_{prog}: $p(i_{t+1} \mid c_t) = \exp s(i_{t+1} \mid c_t) / \sum_j \exp s(j_{t+1} \mid c_t)$. In SL the applier embeddings do not need to be trained; they are just generated from ξ_t^{appl} elements of the trace. Specifically, the program embedding of applier i_t is set according to equation (2).

CNPI can also be trained by policy gradient reinforcement learning (RL) [4]. No execution trace is given and the core tries to complete the task by making program calls following the probabilities $p(i_{t+1} \mid c_t)$ and feeding-forward the LSTM. An episode ends if the task is completed or the number of steps reaches MAX_NSTEP. In our experiments, MAX_NSTEP = $K \cdot n$, where K is the number of callable arguments for combinators, n is the complexity of the problem to be solved. A reward R_T is given when an episode ends at step T:

$$R_T = \begin{cases} +1 - 0.1 \times T & \text{if task is completed} \\ -1 - 0.1 \times T & \text{if task not is completed} \end{cases} \tag{6}$$

At each time step t, a condition $\tilde{c}_t$ is sampled from a Bernoulli distribution defined by the output of the detector. the next program to be called is identified as $F[\tilde{i}_{t+1}]$, where $\tilde{i}_{t+1}$ is sampled from $\{p(i_{t+1} \mid \tilde{c}_t)\}$. The core is trained using stochastic gradient ascent on a mixed objective with parts: an RL objective of maximizing expected reward, plus an SL objective of maximizing the likelihood of correct flag of program return:

$$\Delta\theta \propto \sum_{t=1}^{T} (\nabla_\theta \log p_\theta(\tilde{i}_{t+1} \mid \tilde{c}_t) R_T + \nabla_\theta \log p_\theta(r_t \mid \tilde{c}_t) \mathbb{1}(R_T > 0)) \tag{7}$$

The RL objective is derived from the REINFORCE algorithm Williams (1992). Note that the SL objective only takes effect on episodes where a positive reward is received on task completion. This combination of an RL objective and an SL objective to optimize a policy is also used in Oh et al. (2017) to learn parameterized skills. The detector is also trained to maximize expected reward using REINFORCE:

$$\Delta w \propto \sum_{t=1}^{T} (\nabla_w \log p_w(\tilde{c}_t \mid e_t) R_T) \tag{8}$$

Once the detector and the core have been learned, applier embeddings can also be learned using RL. After the program $\tilde{i}$ is called with detector and callable arguments $\tilde{i}'$ and $\tilde{a}_j, j = 1..K-1$, a reward $R \in \{-1, +1\}$ is given according to whether the task has been completed. The applier embedding parameterized by ϕ is updated as:

$$\Delta\phi \propto \nabla_\phi \left(\log p_\phi(\tilde{i}) + \log p_\phi(\tilde{i}') + \sum_{j=1}^{K} \log p_\phi(\tilde{a}_j) \right) R \tag{9}$$

where the identifiers $\tilde{i}$, $\tilde{i}'$ and $\tilde{a}_j$ are sampled respectively from the distributions derived from the corresponding keys stored in the applier's embedding: $\tilde{i} \sim \text{softmax}(M^{key} k_{prog})$, $\tilde{i}' \sim \text{softmax}(W^{weight} k_{c?})$, $\tilde{a}_j \sim \text{softmax}(M^{key} k_{a_j})$, $j = 1..K-1$.

Note that in both SL and RL, detectors are trained separately from the core. This decoupling facilitates the sharing of detectors across programs and the verification of the behavior of the core.

[4] In our experiments we only train CNPI by RL on tasks that can be solved by the three basic combinators in Figure 2.

Table 1: Qualitative comparison of CNPI with NPI (Reed & De Freitas (2016)) and RNPI (Cai et al. (2017)).

Model	general-ization	univer-sality	# verifications of		# trainings of	
			prog. / comb.	encoders / detectors	prog. / comb.	encoders / detectors
NPI	×	×	–	–	per task	per task
RNPI	✓	×	per prog.	per task	per task	per task
CNPI	✓	✓	**once**	**per cond.**	**once**	**per cond.**

4 ANALYSIS

Training CNPI with supervised learning to solve algorithmic tasks consists of the following three steps. First, train and verify the core jointly with the combinators with synthetic *abstract traces*, i.e. sequences of ξ_t^{comb} elements corresponding to the correct execution of the combinators given correct conditions. After the core has been trained and verified for correct behavior, the core and the combinator embeddings are fixed. This step is done only once before solving any specific task. Second, for any specific algorithmic task, identify the conditions needed to solve the task, then train and verify detectors to detect these conditions. Finally, solve the task by iteratively defining appliers (i.e. generate their embeddings) according to the ξ_t^{appl} elements of the traces generated by the combinatory program of the task. Note that once the three steps are done, the correct behavior of the learned program on any input complexity of the problem, i.e. perfect generalization, is automatically guaranteed.

We call a program *combinatorizable* if it can be converted to a combinatory equivalent. From the above-mentioned steps we know that a CNPI with a fixed trained core can represent and interpret any combinatorizable program with perfect generalization, given that appropriate detectors for the task have been trained and verified. In other words, the CNPI is universal with respect to the set of all combinatorizable programs. This set of programs is adequate for solving most algorithmic tasks, considering the proposition proposed in Appendix 7.3 that any recursive program is combinatorizable and the fact that most, if not all, algorithmic tasks have a recursive solution.

We argue that universality is a property harder to achieve than the generalization property discussed in Cai et al. (2017). The paper provides provable guarantees of perfect generalization for several programs. However, the authors did not consider the problem of universality with a fixed core. In fact, although RNPI can be trained on a particular task and verified for perfect generalization, after training on a new task causing changes to the parameters of the core, the property of perfect generalization on old tasks may not hold any more. In contrast, CNPI provides both generalization and universality. Table 1 qualitatively compares CNPI with NPI and RNPI.

Due to the decomposition of programs into combinators and appliers, and the decoupling of detectors from the core, we can verify the perfect generalization of a particular program using much fewer test inputs than RNPI. Table 2 compares the verification set sizes of RNPI and CNPI for four tasks.

5 EXPERIMENTS

While the previous section analyzes the universality of CNPI, this section shows results on the empirical evaluation of its learnability via both SL and RL experiments. We mainly report results on learning the core and the combinators, assuming that detectors for the tasks have been trained. Learning a detector in our CNPI architecture is a standard binary classification problem, which can be trivially solved by training a classifier.

To evaluate how CNPI improves learnability over the original NPI architecture, in some experiments we use the RNPI model as a baseline. It has the same architecture as NPI and allows recursive calls. For a CNPI with K callable arguments (denoted as CNPI-K), we construct a counterpart RNPI with $K \cdot n$ existing actual programs (either composite programs or ACTs) in the program memory as base programs (denoted as RNPI-Kxn). These base programs are divided into n sets corresponding to to n different tasks (e.g. grade school addition and bubble sort). Then new programs are learned over

Table 2: Verification set sizes of RNPI and CNPI for four tasks. For CNPI, combinators are verified only once and for each specific task only the verification of detectors is needed. As proposed in Cai et al. (2017), in Bubblesort encoders/detectors receive raw values at pointers as input while in Bubblesort2 they directly receive the results of relevant comparisons. So less verifications are needed. For $linrec$ and $_mapself$, verification set size with and without transient condition are both shown, separated by a '/'.

Model	Combinators		Addition		Bubblesort		Bubblesort2		Toposort traverse		Quicksort	
RNPI			ADD:	20003	BUBBLESORT:	261	BUBBLESORT:	3	TRAVERSE:	3	QUICKSORT:	11
			ADD1	200	BSTEP:	900	BSTEP:	10	CHECK_CHILD:	9	COMPSWAP	
			CARRY:	100	COMPSWAP:	100	COMPSWAP:	3	EXPLORE:	15	_LOOP:	18
			LSHIFT:	20002	RSHIFT:	55	RSHIFT:	2	**Total:**	**27**	COMPSWAP:	6
			Total:	**40305**	RESET:	52	RESET:1	1			PARTITION:	4
					LSHIFT:	710	LSHIFT:	11			**Total:**	**39**
					Total:	**2078**	**Total:**	**30**				
CNPI	seq:	1	(in ADD)		(in BUBBLESORT)		(in BUBBLESORT)		(in DIVIDE)		(in COMPSWAP)	
	cond:	2	A[P2]≠END?:	11	A[P3]≠END?:	11	A[P3]≠END?:	2	$Q_{color}(DAG[v]$		A[P_j]≤A[P_{hi}]?:	2
	linrec:	2/3	(in ADD1)		(in BSTEP)		(in BSTEP)		$[childList[v]])$		(in COMPSWAP	
	treerec:	2	CARRY?:	200	A[P2]≠END?:	11	A[P2]≠END?:	2	is valid?:	4	_LOOP)	
	_mapself:	2/3	**Total:**	**211**	(in LSHIFT)		(in LSHIFT)		(in TRVERSE)		A[P_j]≠A[P_{hi}]?:	2
	Total:	**9/11**			A[P1]≠BEGIN?:	11	A[P1]≠BEGIN?:	2	$Q_{color}(v)$ is	4	(in QUICKSORT)	
					(in COMPSWAP)		(in COMPSWAP)		WHITE?:		$P_{lo}<P_{hi}$:	2
					A[P1]>A[P2]?:	90	A[P1]>A[P2]?:	2	**Total:**	**8**	**Total:**	**6**
					Total:	**123**	**Total:**	**8**				

each set by calling the corresponding K base programs as subprograms. Note that some of these new programs may share same patterns (e.g. ADD1 and BSTEP, we call them isomorphic programs), but in RNPI they are treated as different programs and the core needs to learn all of them. For fair comparison, the counterpart RNPI uses the same detector as the CNPI.

For all experiments, we used a one-layer LSTM for the core. We trained the CNPI using plain SGD with batch size 1, and learning rate of 0.5 and 0.1 for SL and RL experiments respectively. For the SL experiments, the learning rate was decayed by a factor of 0.1 if prediction accuracy did not improve for 10 epochs.

5.1 Supervised learning results

We found that, as expected, a CNPI can be trained to learn the the set of four combinators using synthetic abstract traces without any difficulty. From Section 4 we know that this CNPI is able to learn all combinatorizable programs (including the four in Appendix 7.3) with perfect generalization. To further stress the learning capacity of the core, we enlarge the small set of four combinators to a full set of all possible combinators with the following two constraints: 1) branching can only happen at the beginning of the execution; 2) call to the $self$ argument, i.e. recursive call, can only be made at the end of the execution (i.e. only tail recursion is allowed). For $K = 4$, this full set has 57 combinators (including the three basic combinators).

Cores with different number of LSTM cells were trained to learn this full set of combinators. We compare two methods of feeding the combinators embedding to the core LSTM: use the embedding as input to the LSTM at every time step (Emb-as-Input), as is done in Reed & De Freitas (2016), and using it as the initial state (i.e. hiddens and cells) of the LSTM (Emb-as-State0). For both methods the combinator embedding size is set to be equal to the LSTM size. Note that the Emb-as-State0 model has fewer parameters than Emb-as-Input with the same number of cells. We also trained a sequence-to sequence model from Sutskever et al. (2014) where an encoder LSTM takes in the text code representation of the combinator (a simplified version of the pseudo-code in Figure 2) and the last state of the encoder is used as the combinator embedding to initialize the core LSTM's state. This seq2seq model can be seen as a miniature of an instruction-to-action architecture. We see in Figure 5 that the Emb-as-State0 model achieves better prediction accuracy than the Emb-as-Input model with the same size. Particularly, the Emb-as-State0 model with only 5 cells can learn all the 57 combinators with 100%. This LSTM is much smaller than the one used in Reed & De Freitas (2016) which has two layers of size 256. The seq2seq model can also achieve 100% accuracy with 7 cells. In subsequent experiments we used a core LSTM of size 16 if not mentioned otherwise.

We compare the abilities of CNPI and RNPI to learn new combinators/programs with a fixed core. The models were first trained on a combinator/program set to get 100% accuracy, then trained on

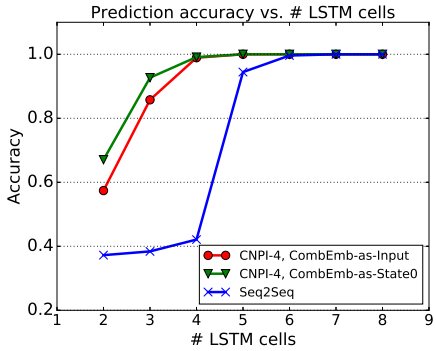


Figure 5: Prediction accuracy on the full combinator set with 4 callable arguments

Table 3: % accuracy of learning new programs/combinators and remembering old ones with a fixed core. The maximum accuracy obtained when training on new set are 100% and 97.7% for RNPI and CNPI respectively.

Model	Old / New set	Train old	Train new (with fixed core)	Test old
RNPI-4x2	50% 1st full prog. set /	100	> 90	12.9
	50% 2nd full prog. set	100	> 97	3.3
	100% 1st full prog. set /	100	> 90	6.5
	100% 2nd full prog. set	100	> 97	1.7
CNPI-4	50% full comb. set / the other 50%	100	97.7	**100**

a new set with the core fixed. Finally the models were tested on the old set to see if they are still remembered by the models. For the CNPI-4 model the old and new combinators were generated by a random even split of the full combinator set. For the RNPI-4x2 model with two sets of base programs, we constructed a full set of all possible composite programs for each set of base programs, as with combinators. Then old and new programs were randomly sampled from the two sets respectively. Note that the old and new programs generated this way have certain proportion of isomorphic programs, and this proportion grows with the percentage of random sampling. In the RNPI experiment, the program key embeddings need to be learned jointly with the program embeddings, otherwise the model would not be able to learn the new programs. As shown in Table 3, although both models can be trained on the new set with high accuracy, when tested on old ones, RNPI shows catastrophic forgetting, which becomes more severe as there are more isomorphic programs between the old and new set. In contrast, CNPI remembers old combinators perfectly.

5.2 Curriculum for reinforcement learning

We find that curriculum learning is necessary for training CNPI with RL. Table 4 shows the curriculum we used for training CNPI for the sorting task. For each subtask, the programs to be learned (including detectors) are bolded and colored. The curriculum has two stages. In the first stage, the combinators were trained with simple auxiliary tasks, using ACTs and DETs as arguments. The learned combinators are then used as prerequisite for solving the actual tasks. The tasks for each combinator is designed to ensure that the task will be completed if and only if the combinator is correctly executed as defined in Figure 2. In the second stage, detectors and appliers can be learned in two forms: we can either define a sketch for solving the task (similar to the policy sketches in Andreas et al. (2017)), with some learnable arguments (as in the Compare and swap and Output max tasks), or learn an applier to solve the task using already learned arguments (as in the Sort task). Note that for brevity we define some appliers for resetting pointers directly without any learning after at the end of Stage 1. In fact, they could also be learned the same way as the appliers in Stage 2.

Table 4: Curriculum for training CNPI for the sorting task using RL. The programs to be learned (including detectors) are bolded and colored. Several ACTs are added to help learn the tasks: OUT_x: write the element at pointer x to position pointed by P3 and advanced P3 one step. CLEAR_x: set the element at pointer x to -1. OUTCLEAR_x: output then clear x.

Subtask	**Description**	**Prerequisite**	**Program to learn**
Stage 1: Learning the core and combinators			
Swap and output easy	Output A[P1]	ACTs, DETs	**seq0**(; OUT_1, NOP, NOP)
Swap and output	Output A[P1], then swap A[P1] and A[P2], finally output A[P2]	ACTs, DETs	**seq**(; OUT_1, SWAP_12, OUT_2)
Conditional output easy	Output A[P2] if P2 is in array, otherwise output 'OK'	ACTs, DETs	**cond0**(A[P2]≠END?; OUT_2, NOP, OUT_OK)
Conditional output	Output and clear A[P2] by setting it to -1 if P2 is in array, otherwise output 'OK'	ACTs, DETs	**cond**(A[P2]≠END?; OUT_2, CLEAR_2, OUT_OK)
Copy easy	Output the first element if array is empty, otherwise output 'OK'	ACTs, DETs	**linrec0**(A[P2]≠END?; OUT_2, NOP, OUT_OK)
Copy	Output elements sequentially till the end of array, then output 'OK'	ACTs, DETs	**linrec**(A[P2]≠END?; OUT_2, P2_RIGHT, OUT_OK)
Reset pointers	Reset P1 and P2 to the appropriate beginning position	ACTs, DETs, combinators	RESET_1: linrec(A[P1]≠END?; P1_LEFT, NOP, NOP) RESET_2: linrec(A[P2]≠END?; P2_LEFT, NOP, P2_RIGHT) RESET: seq(; RESET_1, RESET_2, NOP)
Stage 2: Learning detectors and appliers			
Conditional swap	Conditionally swap two elements	ACTs, DETs, combinators	COMPSWAP: cond(**A[P1]>A[P2]?**, SWAP_12, NOP, NOP)
Output max	Find and output the max element in the array then clear it	ACTs, DETs, combinators A[P1]>A[P2]? COMPSWAP	MAX: linrec(A[P2]≠END?; **STEP**, P2_RIGHT, OUTCLEAR_1)
Sort	Sort the array by repeatedly outputting the current max element	ACTs, DETs, combinators A[P1]>A[P2]? COMPSWAP, MAX	**SORT**: linrec(A[P3]≠END?; MAX, RESET, NOP)

Table 5: % success of training the core with the combinators with RL

Sampling method	No curriculum				Mixed curriculum				Gradual curriculum			
	all	seq	cond	linrec	all	seq	cond	linrec	all	seq	cond	linrec
Uniform	**7**	11	10	7	**17**	33	28	17	**49**	94	49	49
Adaptive	**31**	74	41	31	**78**	87	80	78	**91**	92	91	91

Table 6: % success of training CNPI and RNPI with RL

Stage	CNPI-4	RNPI-4x2	RNPI-4x3
Easy	**99**	25	0
Final	**91**	0	0

5.3 Reinforcement learning results

As shown in the curriculum in Tabel 4, the three basic combinators are learned by trying to complete the three auxiliary tasks Swap and output, conditional output and Copy. Though being quite simple programs, we find that the three basic combinators are still difficult to learn with plain policy gradient RL. To facilitate learning. we use the adaptive sampling technique proposed in Reed & De Freitas (2016). Example traces are fetched with frequency proportional to the models current prediction error. We set the sampling frequency using a softmax over a moving average of prediction error over last 10 episodes, with temperature 1. Besides, for each combinator's auxiliary task we design an easy version of the task, which corresponds to a partial completion of the true task (see Table 4). Then a curriculum can be formed for each combinator by either mixing the easy and true task (mixed), or complete the easy task first before going to the true one (gradual). For each different use of adaptive sampling and curriculum we ran 100 experiments with a maximum number of 5000 episodes for each experiment. Table 5 shows the success rate that all three auxiliary tasks are completed along with success rates for completing the task for each combinator. As shown in Table 5, both adaptive sampling and the curriculum help training considerably. A relatively high success rate of 91% can be obtained with adaptive sampling and the gradual curriculum. We can also know from Table 5 that of the three combinators seq is the easiest to learn while $linrec$ is the hardest.

We compare the success rate of training CNPI (CNPI-4) with its counterpart RNPI models RNPI-4xn. For each combinator's auxiliary task, we constructed n different versions of the task by providing different ACTs and DETs. For example, for the Copy task we can use different pointers (e.g. P1) to move in different directions (e.g. to left), and output different symbols when finished (e.g. 'DONE'). This results in a total of $3 \times n$ tasks. Then we trained the RNPI to learn $3 \times n$ actual programs in parallel to solve these tasks. The RNPI-4xn models were trained with adaptive sampling and gradual curriculum for a maximum number of 10000 $\times n$ episodes, and for each model the success rate over 100 trials are shown in Table 6. Due to the enlarged candidate set of the next program to call from 4 to 4 $\times n$ and the increased number of programs to be learned from 3 to 3 $\times n$, it is much more difficult to train RNPI with RL. RNPI-4x2 finishes the first stage of the curriculum to complete the easy tasks with a success rate of 25% while fails completely on the final stage to complete the true tasks. RNPI-4x3 can not even finish the easy stage.

In CNPI detectors are pretrained to output a binary condition, which is used as input when training the core. This is different from the NPI architecture, where decoders are trained jointly with the core by feeding the decoder's vector output to the core as input. To evaluate the impact of different detectors on learning the core and combinators, we compare the success rate on the Copy task to learn $linrec$ with a pretrained detector to that with a jointly trained detector. The jointly trained detector uses the one-hot encoding of the array element as input and has a linear output layer of size 4. As an ablation test, for both the pretrained and jointly trained detector, we also trained the core without using transient condition proposed in Section 3.2. As shown in Table 7, $linrec$ can be learned jointly with a detector that detects the termination condition of the loop, with success rate comparable to that of the pretrained detector. However, when transient condition is not used, both success rates drop, indicating that transient condition is an effective mechanism that facilitate learning.

Table 7: % success of learning linrec with different types of detectors

transient condition	pretrained detector	jointly trained detector
w/o	60	78
w/	91	91

We trained the A[P1]>A[P2]? detector in the Conditional swap task with the RL objective. The input to the detector is the one-hot encoding of the two elements. Then the STEP applier was learned in the context of the Output max task by maximizing the expected reward of completing the task; see equation (9). The embedding of STEP was learned successfully in 79% out of the 100 experiments we ran. In each successfully trial one of the two appliers, seq(; COMPSWAP, P1_LEFT, NOP) and cond(A[P1]>A[P2]?; NOP, NOP, MOVE_12) was learned, which was equivalent to finding the max element by a pass of bubble sort and selection sort respectively. MOVE_12 is a primitive applier we defined to move P1 forward until reaching P2. Finally, the Sort applier was learned to complete the Sort task, with success rate of 62% over 100 experiments. Both bubble sort and selection sort were learned by calling the two learned STEP appliers respectively [5].

6 Conclusion

The problem of universality of NPI architectures is addressed for the first time. We have shown that CNPI overcomes two major limitations of NPI by incorporating an important construct from functional programming literature: combinator abstraction. CNPI can be trained successfully with both strong and weak supervision. It offers the opportunity of applying neural programming to broader domains.

References

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 166–175, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

Marcin Andrychowicz and Karol Kurach. Learning efficient algorithms with hierarchical attentive memory. *arXiv preprint arXiv:1602.03218*, 2016.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations (ICLR)*, 2017.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[5]Here we use a slightly different form of sort compared with Reed & De Freitas (2016) and Cai et al. (2017). Instead of sorting in-place, the max element found in each pass is written to a second array pointed to by P3

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

Junhyuk Oh, Singh Satinder, Lee Honglak, and Kholi Pushmeet. Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*, 2016.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

7 APPENDIX

7.1 BUILT-IN PROGRAMS TO SUPPORT TREE RECURSION

We use some built-ins facilities, including a state stack, a combinator, four ACTs and a detector, to support tree recursion. They are listed in Table 8. The pseudo-code for the built-in combinator _map_self is shown in Figre 6. The built-in ACT _load_state need to be overloaded for each specific task because the state needed to for a recursive call may be different for each task. [6] See Figure 8 and 9.

Table 8: Built-ins to support tree recursion

Type	Name	Descriptions
data structure	state state	A stack to hold states for recursive calls
combinator	_mapself	Make recursive call for each state on stack
ACT	_push_sentinel	Push a sentinel to stack to terminate recursive call loop
	_push	Push a state to stack
	_pop	Pop a state from stack
	_load_state	Load the state on top of stack before recursive call
detector	_top≠SENTINEL?	Detect the termination condition for recursive call loop

```
# used by treerec
def _mapself(divisible?/c?; self,
    pre/a1, divide/a2, post/a3):
  if _top≠SENTINEL?():
    _load_state()
    self(divisible?; self,  pre, divide, post)
    _pop()
    _mapself(divisible?; self, pre, divide, post)
  else:
    _pop()  # pop the sentinel
    _load_state()
```

Figure 6: Built-in combinator _mapself used by treerec

[6] The state can be seen as arguments for a recursive call but NPI do not support passing these arguments to the calleee.

7.2 LISTING OF LEARNED PROGRAMS

We list the programs learned by CNPI in Table 9, 10 and 11. Note that for brevity appliers that call ACTs directly (e.g. P1_RIGHT) are treated as "primitive appliers", i.e. they are defined and used as ACTs without learning.

Appliers/ACTs	**Descriptions**	**Calls**
BUBBLESORT	Perform bubble sort (ascending order)	BSTEP, RESET
BSTEP	Conditionally swap and advance pointers	COMPSWAP, RSHIFT
COMPSWAP	Conditionally swap two elements	SWAP_12
RSHIFT	Moves P_1 and P_2 one step right	P1_RIGHT, P2_RIGHT
RESET	Reset three pointers	LSHIFT, P3_RIGHT
LSHIFT	Move both pointers all the way left	P1_LEFT, P2_LEFT
SWAP_12	SWAP(P_1, P_2)	ACT
P1_RIGHT	MOVE(P_1, UP)	ACT
P2_RIGHT	MOVE(P_2, UP)	ACT
P3_RIGHT	MOVE(P_3, UP)	ACT
P1_LEFT	MOVE(P_1, DOWN)	ACT
P2_LEFT	MOVE(P_2, DOWN)	ACT
MOVE_12	MOVE(P_1, P_2)	ACT
ACT	SWAP, MOVE, OUT, OUT_OK, CLEAR, OUTCLEAR	-
DET	$P_1 \neq$END?, $P_2 \neq$END?, $P_3 \neq$END?	-

Table 9: Programs learned for bubble sort

Appliers/ACTs	**Descriptions**	**Calls**
QUICKSORT	Perform quick sort (ascending order)	DIVIDE
DIVIDE	Partition the array and prepare for recursive calls	PARTITION, SAVE_STATE2, SAVE_STATE1
PARTITION	Run the partition function	PRE_COMPSWAP_LOOP, COMPSWAP_LOOP, POST_COMPSWAP_LOOP
PRE_COMPSWAP_LOOP	Prepare for COMPSWAP_LOOP	SET_PIVOT_LO, SET_J_LO
COMPSWAP_LOOP	Run the loop inside the partition function	COMPSWAP, PJ_RIGHT
COMPSWAP	Conditionally swap two elements and advanced the pointer	SWAP_PIVOTJ, PPIVOT_RIGHT
POST_COMPSWAP_LOOP	Set A[P_{pivot}] and P_j after COMPSWAP_LOOP	SWAP_PIVOTHI, SET_J_NULL
SWAP_PIVOTJ	SWAP(P_{pivot}, P_j)	ACT
PPIVOT_RIGHT	MOVE(P_{pivot}, UP)	ACT
PJ_RIGHT	MOVE(P_j, UP)	ACT
SWAP_PIVOTHI	SWAP(A[P_{pivot}], A[P_{hi}])	ACT
SAVE_STATE1	Push state for 1st recursive call to state stack	built-in ACT
SAVE_STATE2	Push state for 2nd recursive call to state stack	built-in ACT
ACT	SWAP, MOVE, SET_PIVOT_LO, SET_J_LO, SET_J_NULL	-
built-in ACT	_push, _load_state	-

Table 10: Programs learned for quick sort

Appliers/ACTs	**Descriptions**	**Calls**
TRAVERSE	Traverse the graph by DFS	PRE, DIVIDE, POST
PRE	Prepare for traverse	WRITE(COLOR_CURR, COLOR_GREY)
DIVIDE	Save the children of v to state stack	SAVE_STATE, MOVE(ChilsdList[v], UP)
SAVE_STATE	Write the current child to v and pushe v to state stack	WRITE(ACTIVATE_NEIGHB), built-in ACT
WRITE(ACTIVATE_NEIGHB)	Write the current child to v	ACT
POST	Write result after traverse	WRITE(COLOR_CURR, COLOR_BLACK), WRITE(RESULT)
MOVE(ChilsdList[v], UP)	Move to the next child of v	ACT
WRITE(COLOR_CURR, COLOR_GREY)	Color v with grey	ACT
WRITE(COLOR_CURR, COLOR_BLACK)	Color v with black	ACT
WRITE(RESULT)	Insert current vertex into result list	ACT
ACT	MOVE, WRITE	-
built-in ACT	_push, _load_state	-

Table 11: Programs learned for traverse in topological sort

7.3 COMBINATORY PROGRAMS FOR ALGORITHMIC TASKS

Figure 7 8 and 9 show the combinatory programs compared with the corresponding normal programs for three algorithmic tasks bubble sort, quick sort and traverse in topological sort. Bubble sort has a nested two levels of linear recursion which are expressed by $linrec$. Quick sort uses bi-recursion and traverse in topological sort uses multi-recursion. They are all expressed by $treerec$ together with SAVE_STATE and _load_state.

A general algorithm for converting any program set expressing an recursive algorithm to a combinatory one is given in Algorithm 3. For a program it first removes any multiple recursive calls by using _push_state and _mapself, then removes any loop by replacing them with tail recursion. Finally an iterative maximum matching procedure is used to convert the program to a set of appliers iteratively. We put forward a proposition that any recursive program can be converted to a combinatory counterpart in this way. We leave formal proof of this proposition to future work. Note that non-recursive programs, (e.g. the stack-based iterative program for topological sort used in Cai et al. (2017)) may still be combinatorized by Algorithm 3, but the process is less straightforward.

```
def COMPSWAP():
  if A[P1]>A[P2]:
    SWAP(P1,P2)

def RSHIFT():
  MOVE(P1,UP)
  MOVE(P2,UP)

def BSTEP():
  if A[P2] ≠ END:
    COMPSWAP()
    RSHIFT()
    BSTEP()

def LSHIFT():
  if A[P1]≠ END:
      MOVE(P1,DOWN)
      MOVE(P2,DOWN)
    LSHIFT()

def RESET():
  LSHIFT()
  MOVE(P3,UP)

def BUBBLESORT():
  if A[P3]≠END:
    BSTEP()
    RESET()
    BUBBLESORT()
```

(a) Normal Program

```
def COMPSWAP:
  cond(A[P1]>A[P2]?; SWAP_12, NOP, NOP)

def RSHIFT:
  seq(; P1_RIGHT, P2_RIGHT, NOP)

def BSTEP:
  linrec(A[P2]≠END?; COMPSWAP, RSHIFT,
      NOP)

def LSHIFT:
  linrec(A[P1]≠END?; P1_LEFT, P2_LEFT,
      NOP)

def RESET:
  seq(; LSHIFT, P3_RIGHT, NOP)

def BUBBLESORT:
  linrec(A[P3]≠END?; BSTEP, RESET, NOP)
```

(b) Combinatory Program

Figure 7: Normal and combinatory programs for bubble sort.

```
def COMPSWAP():
  if A[Pj]≤A[Phi]:
    SWAP(Ppivot, Pj)
    MOVE(Ppivot, UP)

def COMPSWAP_LOOP():
  if Pj≠Phi:
    COMPSWAP()
    MOVE(Pj, UP)
    COMPSWAP_LOOP()

def PARTITION():
  SET_PIVOT_LO()
  SET_J_LO()
  COMPSWAP_LOOP()
  SWAP(Ppivot ,Phi)
  SET_J_NULL()

def QUICKSORT():
   if Plo<Phi:
      PARTITION()
      STACK(STACK_PUSH_CALL2)
      STACK(STACK_PUSH_CALL1)
      WRITE(Phi, ENV_STACK_HI_PEEK)
      WRITE(Plo, ENV_STACK_LO_PEEK)
      QUICKSORT
      STACK(STACK_POP)
      WRITE(Phi, ENV_STACK_HI_PEEK)
      WRITE(Plo, ENV_STACK_LO_PEEK)
      QUICKSORT
      STACK(STACK_POP)
```

(a) Normal Program

```
def PRE_COMPSWAP_LOOP:
  seq(; SET_PIVOT_LO, SET_J_LO, NOP)

def COMPSWAP:
  cond(A[Pj]≤A[Phi]?; SWAP_PIVOTJ,
      PPIVOT_RIGHT, NOP)

def COMPSWAP_LOOP:
  linrec(Pj≠Phi?; COMPSWAP, PJ_RIGHT, NOP)

def POST_COMPSWAP_LOOP:
  seq(; SWAP_PIVOTHI, SET_J_NULL, NOP)

def PARTITION:
  seq(; PRE_COMPSWAP_LOOP, COMPSWAP_LOOP,
      POST_COMPSWAP_LOOP)

def SAVE_STATE2:
   _push(Ppivot+1, Phi)

def SAVE_STATE1:
   _push(Plo, Ppivot-1)

def DIVIDE:
  seq(; PARTITION, SAVE_STATE2,
      SAVE_STATE1)

def _load_state:
  Write value pair on top of the state
      stack to Phi and Plo

def QUICKSORT:
  treerec(Plo<Phi?; NOP, DIVIDE, NOP)
```

(b) Combinatory Program

Figure 8: Normal and combinatory programs for quick sort.

```
def TRAVERSE():
 if Q_color(v) is WHITE:
  WRITE(COLOR_CURR, COLOR_GREY)
  while Q_color(DAG[v][childList[v]]) is
       valid:
   WRITE(ACTIVATE_NEIGHB)
   TRAVERSE()
   MOVE(ChildList[v], UP)
  WRITE(COLOR_CURR, COLOR_BLACK)
  WRITE(RESULT)
```

(a) Normal Program

```
def PRE:
 WRITE(COLOR_CURR, COLOR_GREY)

def SAVE_STATE:
 seq(; WRITE(ACTIVATE_NEIGHB), _push(v),
      NOP)

def DIVIDE:
 linrec(Q_color(DAG[v][childList[v]]) is
      valid?; SAVE_STATE,
      MOVE(ChildList[v], UP), NOP)

def _load_state:
 Write value on top of the state stack
      to v

def POST:
 seq(; WRITE(COLOR_CURR, COLOR_BLACK),
      WRITE(RESULT), NOP)

def TRAVERSE:
 treerec(Q_color(v) is WHITE?; PRE,
      DIVIDE, POST)
```

(b) Combinatory Program

Figure 9: Normal and combinatory programs for traverse in topological sort.

Algorithm 3 Convert a recursive program set to a combinatory one

Inputs: A program set P for solving a task by a recursive algorithm
Outputs: A combinatory program set Q equivalent to P
function CONVERT(P)
    $Q \leftarrow \{\}$
    for all subprogram $p \in P$ **do**
        Find any multiple recursive calls to the same function (including recursive calls in a loop) and replace them with a _push_state(). Then add a _mapself() at the end of these calls
        Find any loop (without recursive calls in the body) and replace it with a tail recursion
        while p is not an applier **do**
            MATCHANDREPLACE(p, treerec, Q)
            MATCHANDREPLACE(p, linrec, Q)
            MATCHANDREPLACE(p, cond, Q)
            MATCHANDREPLACE(p, seq, Q)
    return Q

function MATCHANDREPLACE($p, comb, Q$)
    for all block $b \in p$ **do**
        if b matches the pattern expressed by $comb$ **then**
            replace b with an applier $appl$ calling $comb$
            $Q \leftarrow Q \cup \{appl\}$