Differentiable Programs with Neural Libraries

Alexander L. Gaunt [1] **Marc Brockschmidt** [1] **Nate Kushman** [1] **Daniel Tarlow** [2]

Abstract

We develop a framework for combining differentiable programming languages with neural networks. Using this framework we create end-to-end trainable systems that learn to write interpretable algorithms with perceptual components. We explore the benefits of inductive biases for strong generalization and modularity that come from the program-like structure of our models. In particular, modularity allows us to learn a library of (neural) functions which grows and improves as more tasks are solved. Empirically, we show that this leads to lifelong learning systems that transfer knowledge to new tasks more effectively than baselines.

1. Introduction

Recently, there has been much work on learning algorithms using neural networks. Following the idea of the Neural Turing Machine (Graves et al., 2014), this work has focused on extending neural networks with interpretable components that are differentiable versions of traditional computer components, such as external memories, stacks, and discrete functional units. However, trained models are not easily interpreted as the learned algorithms are embedded in the weights of a monolithic neural network. In this work we flip the roles of the neural network and differentiable computer architecture. We consider *interpretable* controller architectures which express algorithms using differentiable programming languages (Gaunt et al., 2016; Riedel et al., 2016; Bunel et al., 2016). In our framework, these controllers can execute discrete functional units (such as those considered by past work), but also have access to a library of trainable, uninterpretable neural network functional units. The system is end-to-end differentiable such that the source code representation of the algorithm is jointly induced with the parameters of the neural function library. In this paper we explore potential advantages of this class of hybrid model over purely neural systems, with a particular emphasis on lifelong learning systems that learn from weak supervision.

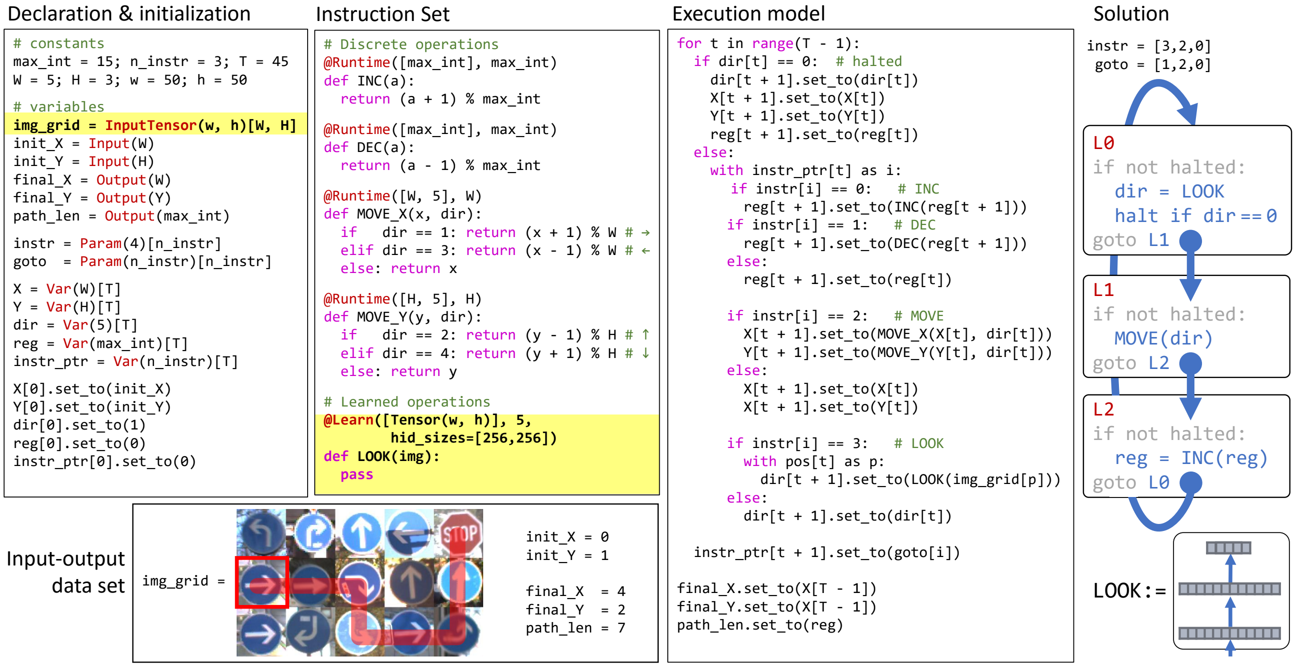
We concentrate on *perceptual programming by example* (PPBE) tasks that have both algorithmic and perceptual elements to exercise the traditional strengths of program-like and neural components. Examples of this class of task include navigation tasks guided by images or natural language (see Fig. 1) or handwritten symbol manipulation (see Sec. 3). Using an illustrative set of PPBE tasks we aim to emphasize two specific benefits of our hybrid models:

First, the source code representation in the controller allows modularity: the neural components are small functions that specialize to different tasks within the larger program structure. It is easy to separate and share these functional units to transfer knowledge between tasks. In contrast, the absence of well-defined functions in purely neural solutions makes effective knowledge transfer more difficult, leading to problems such as catastrophic forgetting in multitask and lifelong learning (McCloskey & Cohen, 1989; Ratcliff, 1990). In our experiments, we consider a lifelong learning setting in which we train the system on a *sequence* of PPBE tasks that share perceptual subtasks.

Second, the source code representation enforces an inductive bias that favors learning solutions that exhibit strong generalization. For example, once a suitable control flow structures (e.g., a `for` loop) for a list manipulation problem was learned on short examples, it trivially generalizes to lists of arbitrary length. In contrast, although some neural architectures demonstrate a surprising ability to generalize, the reasons for this generalization are not fully understood (Zhang et al., 2017) and generalization performance invariably degrades as inputs become increasingly distinct from the training data.

This paper is structured as follows. We first present a language, called NEURAL TERPRET (NTPT), for specifying hybrid source code/neural network models (Sec. 2), and then introduce a sequence of PPBE tasks (Sec. 3). Our NTPT models and purely neural baselines are described in Sec. 4 and 5 respectively. The experimental results are presented in Sec. 6.

[1] Microsoft Research, Cambridge, UK [2] Google Brain, Montréal, Canada (work done while at Microsoft). Correspondence to: Alexander L. Gaunt <algaunt@microsoft.com>.



Declaration & initialization

```
# constants
max_int = 15; n_instr = 3; T = 45
W = 5; H = 3; w = 50; h = 50

# variables
img_grid = InputTensor(w, h)[W, H]
init_X = Input(W)
init_Y = Input(H)
final_X = Output(W)
final_Y = Output(Y)
path_len = Output(max_int)

instr = Param(4)[n_instr]
goto  = Param(n_instr)[n_instr]

X = Var(W)[T]
Y = Var(H)[T]
dir = Var(5)[T]
reg = Var(max_int)[T]
instr_ptr = Var(n_instr)[T]

X[0].set_to(init_X)
Y[0].set_to(init_Y)
dir[0].set_to(1)
reg[0].set_to(0)
instr_ptr[0].set_to(0)
```

Instruction Set

```
# Discrete operations
@Runtime([max_int], max_int)
def INC(a):
  return (a + 1) % max_int

@Runtime([max_int], max_int)
def DEC(a):
  return (a - 1) % max_int

@Runtime([W, 5], W)
def MOVE_X(x, dir):
  if   dir == 1: return (x + 1) % W # →
  elif dir == 3: return (x - 1) % W # ←
  else: return x

@Runtime([H, 5], H)
def MOVE_Y(y, dir):
  if   dir == 2: return (y - 1) % H # ↑
  elif dir == 4: return (y + 1) % H # ↓
  else: return y

# Learned operations
@Learn([Tensor(w, h)], 5,
        hid_sizes=[256,256])
def LOOK(img):
  pass
```

Execution model

```
for t in range(T - 1):
  if dir[t] == 0:  # halted
    dir[t + 1].set_to(dir[t])
    X[t + 1].set_to(X[t])
    Y[t + 1].set_to(Y[t])
    reg[t + 1].set_to(reg[t])
  else:
    with instr_ptr[t] as i:
      if instr[i] == 0:   # INC
        reg[t + 1].set_to(INC(reg[t + 1]))
      if instr[i] == 1:   # DEC
        reg[t + 1].set_to(DEC(reg[t + 1]))
      else:
        reg[t + 1].set_to(reg[t])

      if instr[i] == 2:   # MOVE
        X[t + 1].set_to(MOVE_X(X[t], dir[t]))
        Y[t + 1].set_to(MOVE_Y(Y[t], dir[t]))
      else:
        X[t + 1].set_to(X[t])
        X[t + 1].set_to(Y[t])

      if instr[i] == 3:   # LOOK
        with pos[t] as p:
          dir[t + 1].set_to(LOOK(img_grid[p]))
      else:
        dir[t + 1].set_to(dir[t])

  instr_ptr[t + 1].set_to(goto[i])

final_X.set_to(X[T - 1])
final_Y.set_to(X[T - 1])
path_len.set_to(reg)
```

Solution

```
instr = [3,2,0]
 goto = [1,2,0]
```

```
L0
if not halted:
  dir = LOOK
  halt if dir == 0
goto L1

L1
if not halted:
  MOVE(dir)
goto L2

L2
if not halted:
  reg = INC(reg)
goto L0
```

Input-output data set

```
img_grid =

init_X = 0
init_Y = 1

final_X  = 4
final_Y  = 2
path_len = 7
```

Figure 1: Components of an illustrative NTPT program for learning loopy programs that measure path length (`path_len`) through a maze of street sign images. The learned program (parameterized by `instr` and `goto`) must control the position (`X`, `Y`) of an agent on a grid of (`W`×`H`) street sign images each of size (`w`×`h`). The agent has a single register of memory (`reg`) and learns to interpret street signs using the `LOOK` neural function. A solution consists of a correctly inferred program and a trained neural network. Learnable components are shown in blue and the NTPT extensions to the TERPRET language are highlighted. The red path on the `img_grid` shows the desired behavior and is not provided at training time.

2. Building hybrid models

The TERPRET language (Gaunt et al., 2016) provides a system for constructing differentiable program interpreters that can induce source code operating on basic data types (e.g. integers) from input-output examples. We extend this language with the concept of learnable neural functions. These can either be embedded inside the differentiable interpreter as mappings from integer to integer or (as we emphasize in this work) can act as learnable interfaces between perceptual data represented as floating point `Tensors` and the differentiable interpreter's integer data type. Below we briefly review the TERPRET language and describe the NEURAL TERPRET extensions.

2.1. TERPRET

TERPRET programs specify a differentiable interpreter by defining the relationship between `Inputs` and `Outputs` via a set of inferrable `Params` (that define an executable program) and `Vars` (that store intermediate results). TERPRET requires all of these variables to range over bounded integers. The model is made differentiable by a compilation step that lifts the relationships between integers specified by the TERPRET code to relationships between marginal distributions over integers in finite ranges. Fig. 1 illustrates an example application of the language.

TERPRET can be translated into a TensorFlow (Abadi et al., 2015) computation graph which can then be trained using standard methods. For this, two key features of the language need to be translated:

- **Function application**. The statement `z.set_to(foo(x, y))` is translated into $\mu_i^z = \sum_{jk} I_{ijk}\mu_j^x\mu_k^y$ where $\boldsymbol{\mu}^a$ represents the marginal distribution for the variable a and $\boldsymbol{I}$ is an indicator tensor $\mathbb{1}[i = \texttt{foo}(j,k)]$. This approach extends to all functions mapping any number of integer arguments to an integer output.
- **Conditional statements** The statements `if x == 0: z.set_to(a); elif x == 1: z.set_to(b)` are translated to $\boldsymbol{\mu}^z = \mu_0^x\boldsymbol{\mu}^a + \mu_1^x\boldsymbol{\mu}^b$. More complex statements follow a similar pattern, with details given in (Gaunt et al., 2016).

2.2. NEURAL TERPRET

To handle perceptual data, we relax the restriction that all variables need to be finite integers. We introduce a new `Tensor` type whose dimensions are fixed at declaration, and which is suitable for storing perceptual data. Additionally, we introduce *learnable functions* that can process integer or tensor variables. A learnable func-

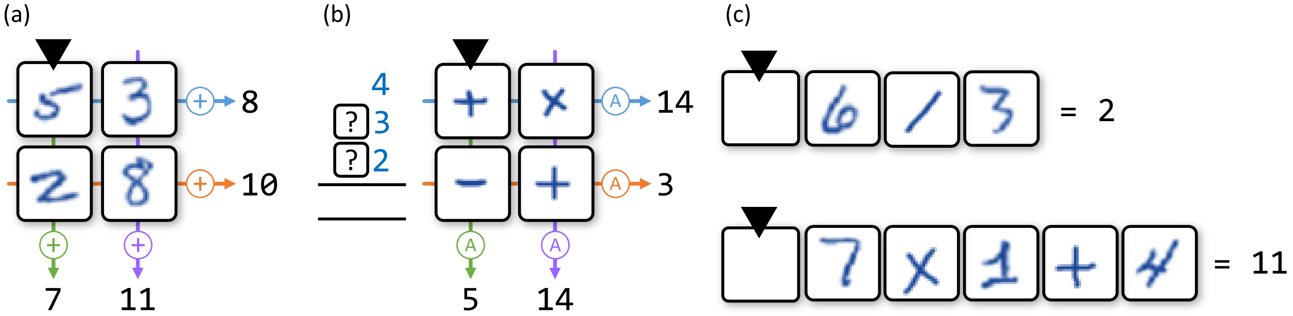


Figure 2: Overview of tasks in the (a) ADD2X2, (b) APPLY2X2 and (c) MATH scenarios. 'A' denotes the APPLY operator which replaces the ? tiles with the selected operators and executes the sum. We show two MATH examples of different length.

tion is declared using `@Learn(`$[d_1,\ldots,d_D]$`,` d_{out}`, hid_sizes=`$[\ell_1,\ldots,\ell_L]$`)`, where the first component specifies the dimensions (resp. ranges) $d_1,\ldots,d_D$ of the input tensors (resp. integers) and the second specifies the dimension of the output. NTPT compiles such functions into a fully-connected feed-forward neural network whose layout is controlled by the `hid_sizes` component (specifying the number neurons in each layer). The inputs of the function are simply concatenated. Tensor output is generated by learning a mapping from the last hidden layer, and finite integer output is generated by a softmax layer producing a distribution over integers up to the declared bound. Learnable parameters for the generated network are shared across every use of the function in the NTPT program, and as they naturally fit into the computation graph for the remaining TERPRET program, the whole system is trained end-to-end. We illustrate an example NTPT program for learning navigation tasks in a maze of street signs (Stallkamp et al., 2011) in Fig. 1.

3. A Lifetime of PPBE Tasks

Motivated by the hypothesis that the modularity of the source code representation benefits knowledge transfer, we devise a *sequence* of PPBE tasks to be solved by sharing knowledge between tasks. Our tasks are based on algorithmic manipulation of handwritten digits and mathematical operators.

In early tasks the model learns to navigate simple 2×2 grids of images, and to become familiar with the concepts of digits and operators from a variety of weak supervision. Despite their simplicity, these challenges already pose problems for purely neural lifelong learning systems.

The final task in the learning lifetime is more complex and designed to test generalization properties: the system must learn to compute the results of variable-length mathematical expressions expressed using handwritten symbols. The algorithmic component of this task is similar to arithmetic tasks presented to contemporary Neural GPU models (Kaiser & Sutskever, 2016; Price et al., 2016). The complete set of tasks is illustrated in Fig. 2 and described in detail below.

ADD2X2 scenario: The first scenario in Fig. 2(a) uses of a 2 × 2 grid of MNIST digits. We set 4 tasks based on this grid: compute the sum of the digits in the (1) top row, (2) left column, (3) bottom row, (4) right column. All tasks require classification of MNIST digits, but need different programs to compute the result. As training examples, we supply *only* a grid and the resulting sum. Thus, we *never* directly label an MNIST digit with its class.

APPLY2X2 scenario: The second scenario in Fig. 2(b) presents a 2 × 2 grid of of handwritten arithmetic operators. Providing three auxiliary random integers $\mathtt{d}_1$, $\mathtt{d}_2$, $\mathtt{d}_3$, we again set 4 tasks based on this grid, namely to evaluate the expression[1] $\mathtt{d}_1\ \mathtt{op}_1\ \mathtt{d}_2\ \mathtt{op}_2\ \mathtt{d}_3$ where ($\mathtt{op}_1$, $\mathtt{op}_2$) are the operators represented in the (1) top row, (2) left column, (3) bottom row, (4) right column. In comparison to the first scenario, the dataset of operators is relatively small and consistent[2], making the perceptual task of classifying operators considerably easier. However, the algorithmic part is more difficult, requiring non-linear operations on the supplied integers.

MATH scenario: The final task in Fig. 2(c) requires combination of the knowledge gained from the weakly labeled data in the first two scenarios to execute a handwritten arithmetic expression.

4. Models

We study two kinds of NTPT model. First, for navigating the introductory 2 × 2 grid scenarios, we create a model

[1] Note that for simplicity, our toy system ignores operator precedence and executes operations from left to right - i.e. the sequence in the text is executed as $((\mathtt{d}_1\ \mathtt{op}_1\ \mathtt{d}_2)\ \mathtt{op}_2\ \mathtt{d}_3)$.

[2] 200 handwritten examples of each operator were collected from a single author to produce a training set of 600 symbols and a test set of 200 symbols from which to construct random 2 × 2 grids.

(a)

```
# initialization:
R0 = READ
# program:
R1 = MOVE_EAST
R2 = MOVE_SOUTH
R3 = SUM(R0, R1)
R4 = NOOP
return R3
```

(b)

```
# initialization:
R0 = InputInt[0]
R1 = InputInt[1]
R2 = InputInt[2]
R3 = READ
# program:
R4 = MOVE_EAST
R5 = MOVE_SOUTH
R6 = APPLY(R0, R1, R4)
R7 = APPLY(R6, R2, R5)
return R7
```

Figure 3: Example solutions for the tasks on the right columns of the (a) ADD2X2 and (b) APPLY2X2 scenarios. The read head is initialized READing the top left cell and any auxiliary InputInts are loaded into memory. Instructions and arguments shown in black must be learned.

which learns to write simple straight-line code. Second, for the MATH scenario we ask the system to use a more complex language which supports loopy control flow (note that the baselines will also be specialized between the 2×2 scenarios and the MATH scenario). Knowledge transfer is achieved by defining a library of 2 neural network functions shared across all tasks and scenarios. Training on each task should produce a task-specific source code solution (from scratch) and improve the overall usefulness of the shared networks. Below we outline further details of the models.

4.1. Shared components

We refer to the 2 networks in the shared library as net_0 and net_1. Both networks have similar architectures: they take a 28×28 monochrome image as input and pass this sequentially through two fully connected layers each with 256 neurons and ReLU activations. The last hidden vector is passed through a fully connected layer and a softmax to produce a 10 dimensional output (net_0) or 4 dimensional output (net_1) to feed to the differentiable interpreter (the output sizes are chosen to match the number of classes of MNIST digits and arithmetic operators respectively).

One restriction that we impose is that when a new task is presented, no more than one new untrained network can be introduced into the library (i.e. in our experiments the very first task has access to only net_0, and all other tasks have access to both nets). This restriction is imposed because if a differentiable program tries to make a call to one of N untrained networks based on an unknown parameter net_choice = Param(N), then the system effectively sees the N nets together with the net_choice parameter as one large untrained network, which cannot usefully be split apart into the N components after training.

4.2. 2×2 model

For the 2×2 scenarios we build a model capable of writing short straight line algorithms with up to 4 instructions. The model consists of a read head containing net_0 and net_1 which are connected to a set of registers each capable of holding integers in the range $0, \ldots, M$, where $M = 18$. The head is initialized reading the top left cell of the 2×2 grid. At each step in the program, one instruction can be executed, and lines of code are constructed by choosing an instruction and addresses of arguments for that instruction. We follow (Feser et al., 2016) and allow each line to store its result in a separate immutable register. For the ADD2X2 scenario the instruction set is:

- NOOP: a trivial no-operation instruction.
- MOVE_NORTH, MOVE_EAST, MOVE_SOUTH, MOVE_WEST: translate the head (if possible) and return the result of applying the neural network chosen by net_choice to the image in the new cell.
- ADD$(\cdot, \cdot)$: accepts two register addresses and returns the sum of their contents.

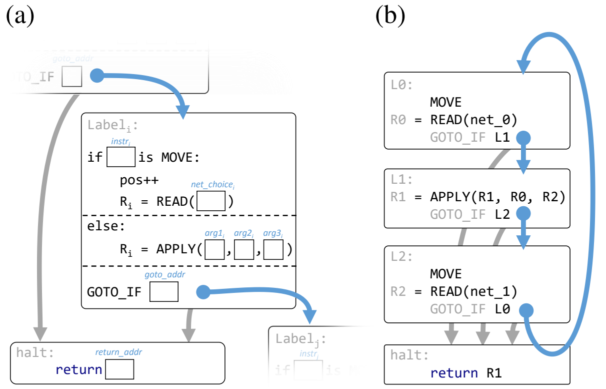
The parameter net_choice is to be learned and decides which of net_0 and net_1 to apply. In the APPLY2X2 scenario we extend the ADD instruction to APPLY(a, b, op) which interprets the integer stored at op as an arithmetic operator and computes[3] a op b. In addition, for the APPLY2X2 scenario we initialize three registers with the auxiliary integers supplied with each 2×2 operator grid [see Fig. 2(b)]. In total, this model exposes a program space of up to $\sim 10^{12}$ syntactically distinct programs.

4.3. MATH model

The final task investigates the synthesis of more complex, loopy control flow. A natural solution to execute the expression on the tape is to build a loop with a body that alternates between moving the head and applying the operators [see Fig. 4(b)]. This loopy solution has the advantage that it generalizes to handle arbitrary length arithmetic expressions.

Fig. 4(a) shows the basic architecture of the interpreter used in this scenario. We provide a set of blocks each containing the instruction MOVE or APPLY, an address, a register and a net_choice. A MOVE instruction increments the position of the head and loads the new symbol into a block's register using either net_0 or net_1 as determined by the block's net_choice. After executing the instruction, the interpreter executes a GOTO_IF statement which checks whether the head is over the end of the tape and if not then it passes control to the block specified by goto_addr, otherwise control passes to a halt block which returns a chosen register value and exits the program. This model describes a space of $\sim 10^6$ syntactically distinct programs.

[3] All operations are performed modulo $(M + 1)$ and division by zero returns M.



```
_IF ☐

Label_i:
if ☐ is MOVE:
    pos++
    R_i = READ(☐)
else:
    R_i = APPLY(☐,☐,☐)
GOTO_IF ☐

halt:
    return ☐

Label_j:
if ☐ is
```

```
L0:
     MOVE
R0 = READ(net_0)
     GOTO_IF L1
L1:
R1 = APPLY(R1, R0, R2)
     GOTO_IF L2
L2:
     MOVE
R2 = READ(net_1)
     GOTO_IF L0
halt:
     return R1
```

Figure 4: Overview of the MATH model. (a) The general form of a block in the model. Blue elements are learnable. (b) A loop-based solution to the task in the MATH scenario.

5. Baselines

To evaluate the merits of including the source code structure in NTPT models, we build baselines that replace the differentiable program interpreter with neural networks, thereby creating purely neural solutions to the lifelong PPBE tasks. As in the NTPT case, we specialize the neural baselines for either the 2×2 tasks (with emphasis on lifelong learning) or the MATH task (with emphasis on generalization).

5.1. 2×2 baselines

We define a *column* as the following neural architecture (see Fig. 5(a)):

- Each of the images in the 2×2 grid is passed through an embedding network with 2 layers of 256 neurons (cf. net_0/1) to produce a 10-dimensional embedding. The weights of the embedding network are shared across all 4 images.
- These 4 embeddings are concatenated into a 40-dimensional vector and for the APPLY2X2 the auxiliary integers are represented as one-hot vectors and concatenated with this 40-dimensional vector.
- This is then passed through a network consisting of 3 hidden layers of 128 neurons to produce a 19-dimensional output.

We construct 3 different neural baselines derived from this column architecture (see Fig. 5):

1. **Indep.**: Each task is handled by an independent column with no mechanism for transfer.
2. **Progressive Neural Network (PNN)**: We follow (Rusu et al., 2016) and build lateral connections linking each task specific column to columns from tasks appearing earlier in the learning lifetime. Weights in all columns except the active task's column are frozen during a training update. Note that the number of layers in each column must be identical to allow lateral connections, meaning we cannot tune the architecture separately for each task.
3. **Multitask neural network (MTNN)**: We split the column into a shared perceptual part and a task specific part. The perceptual part consists of net_0 and net_1 embedding networks (note that we use a similar symmetry breaking technique mentioned in Sec. 4.1 to encourage specialization of these networks to either digit or operator recognition respectively).

   The task-specific part consists of a neural network that maps the perceptual embeddings to a 19 dimensional output. Note that unlike PNNs, the precise architecture of the task specific part of the MTNN can be tuned for each individual task. We consider two MTNN architectures:

   (a) MTNN-1: All task-specific parts are 3 layer networks comparable to the PNN case.

   (b) MTNN-2: We manually tune the number of layers for each task and find best performance when the task specific part contains 1 hidden layer for the ADD2X2 tasks and 3 layers for the APPLY2X2 tasks.

5.2. MATH baselines

For the MATH task, we build purely neural baselines which (1) have previously been shown to offer competitive generalization performance for some tasks with sequential inputs of varying length (2) are able to learn to execute arithmetic operations and (3) are easily integrated with the library of perceptual networks learned in the 2×2 tasks. We consider two models fulfilling these criteria: an LSTM and a Neural GPU.

For the LSTM, at each image in the mathematical expression the network takes in the embeddings of the current symbol from net_0 and net_1, updates an LSTM hidden state and then proceeds to the next symbol. We make a classification of the final answer using the last hidden state of the LSTM. Our best performance is achieved with a 3 layer LSTM with 1024 elements in each hidden state and dropout between layers.

For the Neural GPU, we use the implementation from the original authors[4] (Kaiser & Sutskever, 2016).

[4] available at https://github.com/tensorflow/models/tree/master/neural_gpu

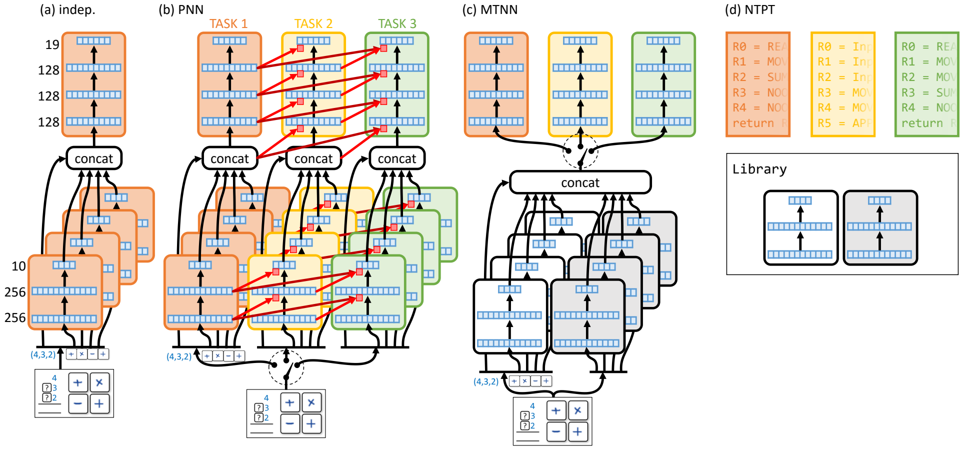


Figure 5: Cartoon illustration of all models used in the 2×2 experiments. See text for details.

6. Experiments

In this section we report results illustrating the key benefits of NTPT for the lifelong PPBE tasks in terms of knowledge transfer (Sec. 6.1) and generalization (Sec. 6.2).

6.1. Lifelong Learning

Demonstration of lifelong learning requires a series of tasks for which there is insufficient data to learn independent solutions to all tasks and instead, success requires transferring knowledge from one task to the next. Empirically, we find that training any of the purely neural baselines or the NTPT model on individual tasks from the ADD2X2 scenario with only 1k distinct 2×2 examples produces low accuracies of around $40 \pm 20\%$ (measured on a held-out test set of 10k examples). Since none of our models can satisfactorily solve an ADD2X2 task independently in this small data regime, we can say that any success on these tasks during a lifetime of learning can be attributed to successful knowledge transfer. In addition, we check that in a data rich regime (e.g. $\geq$4k examples) all of the baseline models and NTPT can independently solve each task with $>80\%$ accuracy. This indicates that the models all have sufficient capacity to represent satisfactory solutions, and the challenge is to find these solutions during training.

We train on batches of data drawn from a time-evolving probability distribution over all 8 tasks in the 2×2 scenarios (see the top of Fig. 6(a)). During training, we observe the following key properties of the knowledge transfer achieved by NTPT:

Reverse transfer: Fig. 6(a) focuses on the performance of NTPT on the first task (ADD2X2:top). The red bars indicate times where the the system was presented with an example from this task. Note that even when we have stopped presenting examples, the performance on this task continues to increase as we train on later tasks - an example of *reverse transfer*. We verify that this is due to continuous improvement of `net_0` in later tasks by observing that the accuracy on the ADD2X2:top task closely tracks measurements of the accuracy of `net_0` directly on the digit classification task.

Avoidance of catastrophic forgetting: Fig. 6(b) shows the performance of the NTPT on the remaining ADD2X2 tasks. Both Fig. 6(a) and (b) include results for the MTNN-2 baseline (the best baseline for these tasks). Note that whenever the dominant training task swaps from an ADD2X2 task to an APPLY2X2 task the baseline's performance on ADD2X2 tasks drops. This is because the shared perceptual network becomes corrupted by the change in task - an example of *catastrophic forgetting*. To try to limit the extent of catastrophic forgetting and make the shared components more robust, we have a separate learning rate for the perceptual networks in both the MTNN baseline and NTPT which is 100 fold smaller than the learning rate for the task-specific parts. With this balance of learning rates we find empirically that NTPT does not display catastrophic forgetting, while the MTNN does.

Final performance: Fig. 6(c) focuses on the ADD2X2:left and APPLY2X2:left tasks to illustrate the relative performance of all the baselines described in Sec. 5. Note that although PNNs are effective at avoiding catastrophic forgetting, there is no clear overall winner between the MTNN and PNN baselines. NTPT learns faster and to a higher accuracy than all baselines for all the tasks considered here. For clarity we only plot results for the *:left tasks: the other tasks show similar behavior and the accuracies for all tasks at the end of the lifetime of learning are presented in Fig. 7.

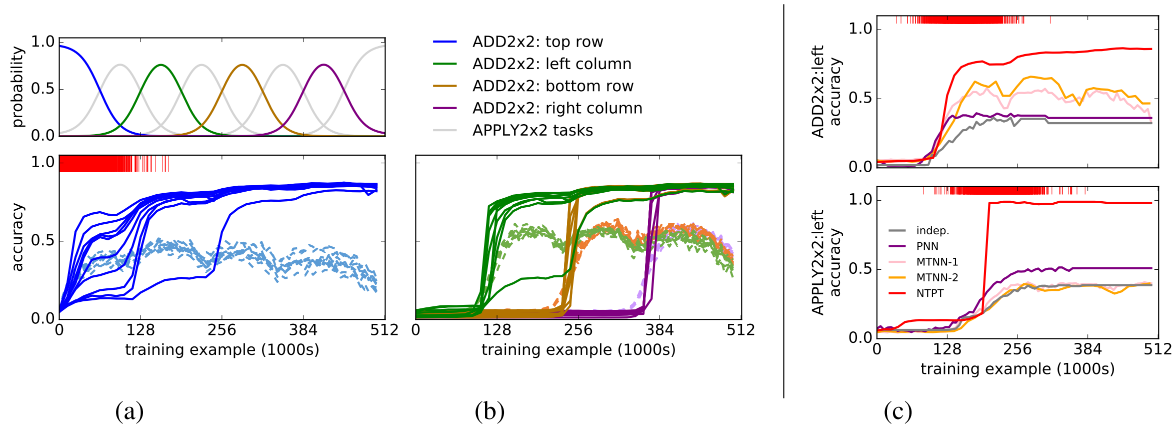


Figure 6: Lifelong learning with NTPT. (a) top: the sequential learning schedule for all 8 tasks, bottom: performance of NTPT (solid) and the MTNN-2 baseline (dashed) on the first ADD2X2 task. (b) performance on the remaining ADD2X2 tasks. (c) Performance of all the baselines on the *:left tasks.

	task	indep	PNN	MTNN-1	MTNN-2	NTPT
ADD2X2	top	35%	35%	26%	24%	87%
	left	32%	36%	38%	47%	87%
	bottom	34%	33%	40%	56%	86%
	right	32%	35%	44%	60%	86%
APPLY2X2	top	38%	39%	40%	38%	98%
	left	39%	51%	41%	39%	100%
	bottom	39%	48%	41%	40%	100%
	right	39%	51%	42%	37%	100%

Figure 7: Final accuracies on all 2×2 tasks for all models at the end of lifelong learning

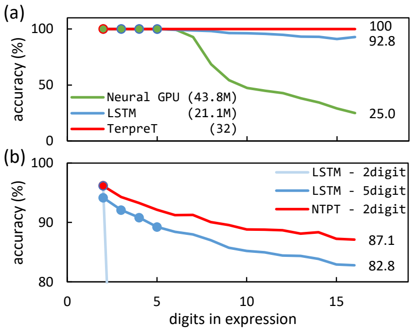


Figure 8: Generalization behavior on MATH expressions. Solid dots indicate expression lengths used in training. We show results on (a) a simpler non-perceptual MATH task (numbers in parentheses indicate parameter count in each model) and (b) the MATH task including perception.

6.2. Generalization

In the final experiment we take `net_0/1` from the end of the NTPT 2×2 training and start training on the MATH scenario. For the NTPT model we train on arithmetic expressions containing only 2 digits. The known difficulty in training differentiable interpreters with free loop structure (Gaunt et al., 2016) is revealed by the fact that only 2/100 random restarts converge on a correct program in a global optimum of the loss landscape. We detect convergence by a rapid increase in the accuracy on a validation set (typically occurring after around 30k training examples). Once the correct program is found, continuing to train the model mainly leads to further improvement in the accuracy of `net_0`, which saturates at 97.5% on the digit classification task. The learned source code provably generalizes perfectly to expressions containing any number of digits, and the only limitation on the performance on long expressions comes from the repeated application of the imperfect `net_0`.

To pick a strong baseline for the MATH problem, we first perform a preliminary experiment with two simplifications: (1) rather than expecting strong generalization from just 2-digit training examples, we train candidate baselines with supervision on examples of up to 5 digits and 4 operators, and (2) we remove the perceptual component of the task, presenting the digits and operators as one-hot vectors rather than images. Fig. 8(a) shows the generalization performance of the LSTM and Neural GPU (512-filter) baselines in this simpler setting after training to convergence.[5] Based on these results, we restrict attention to the LSTM baseline and return to the full task including the perceptual component. In the full MATH task, we initialize the embedding networks of each model using `net_0/1` from the end of the NTPT 2×2 training. Fig. 8(b) shows generalization of the NTPT and LSTM models on expressions of up to 16 digits (31 symbols) after training to convergence. We find that even though the LSTM shows surprisingly effective generalization when supplied supervision for up to 5 digits, NTPT trained on only 2-digit expressions still offers better results.

7. Related work

Lifelong Machine Learning. We operate in the paradigm of Lifelong Machine Learning (LML) (Thrun,

[5] Note that (Price et al., 2016) also find poor generalization performance for a Neural GPU applied to the similar task of evaluating arithmetic expressions involving binary numbers.

1994; 1995; Thrun & O'Sullivan, 1996; Silver et al., 2013; Chen et al., 2015), where a learner is presented a sequence of different tasks and the aim is to retain and re-use knowledge from earlier tasks to more efficiently and effectively learn new tasks. This is distinct from related paradigms of multitask learning (where a set of tasks is presented rather than in sequence (Caruana, 1997; Kumar & Daume III, 2012; Luong et al., 2015; Rusu et al., 2016)), transfer learning (transfer of knowledge from a source to target domain without notion of knowledge retention (Pan & Yang, 2010)), and curriculum learning (training a single model for a single task of varying difficulty (Bengio et al., 2009)).

The challenge for LML with neural networks is the problem of catastrophic forgetting: if the distribution of examples changes during training, then neural networks are prone to forget knowledge gathered from early examples. Solutions to this problem involve instantiating a knowledge repository (KR) either directly storing data from earlier tasks or storing (sub)networks trained on the earlier tasks with their weights frozen. This knowledge base allows either (1) rehearsal on historical examples (Robins, 1995), (2) rehearsal on virtual examples generated by the frozen networks (Silver & Mercer, 2002; Silver & Poirier, 2006) or (3) creation of new networks containing frozen sub networks from the historical tasks (Rusu et al., 2016; Shultz & Rivest, 2001)

To frame our approach in these terms, our KR contains partially-trained neural network classifiers which we call from learned source code. Crucially, we never freeze the weights of the networks in the KR: all parts of the KR can be updated during the training of all tasks - this allows us to improve performance on earlier tasks by continuing training on later tasks (so-called reverse transfer). Reverse transfer has been demonstrated previously in systems which assume that each task can be solved by a model parameterized by an (uninterpretable) task-specific linear combination of shared basis weights (Ruvolo & Eaton, 2013). The representation of task-specific knowledge as source code, learning from weak supervision, and shared knowledge as a deep neural networks distinguishes this work from the linear model used in (Ruvolo & Eaton, 2013).

Neural Networks Learning Algorithms. Recently, extensions of neural networks with primitives such as memory and discrete computation units have been studied to learn algorithms from input-output data (Graves et al., 2014; Weston et al., 2014; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Kurach et al., 2015; Kaiser & Sutskever, 2016; Reed & de Freitas, 2016; Bunel et al., 2016; Andrychowicz & Kurach, 2016; Zaremba et al., 2016; Graves et al., 2016; Riedel et al., 2016; Gaunt et al., 2016; Feser et al., 2016). A dominant trend in these works is to use a neural network controller to managing differentiable computer architecture. We flip this relationship, and in our approach, a differentiable interpreter acts as the controller that can make calls to neural network components.

The methods above, with the exception of (Reed & de Freitas, 2016) and (Graves et al., 2016), operate on inputs of (arrays of) integers. However, (Reed & de Freitas, 2016) requires extremely strong supervision, where the learner is shown all intermediate steps to solving a problem; our learner only observes input-output examples. (Reed & de Freitas, 2016) also show the performance of their system in a multitask setting. In some cases, additional tasks harm performance of their model and they freeze parts of their model when adding to their library of functions. Only (Bunel et al., 2016), (Riedel et al., 2016) and (Gaunt et al., 2016) aim to consume and produce source code that can be provided by a human (e.g. as sketch of a solution) to or returned to a human (to potentially provide feedback).

8. Discussion

We have presented NEURAL TERPRET, a framework for building end-to-end trainable models that structure their solution as a source code description of an algorithm which may make calls into a library of neural functions. Experimental results show that these models can successfully be trained in a lifelong learning context, and they are resistant to catastrophic forgetting; in fact, they show that even after instances of earlier tasks are no longer presented to the model, performance still continues to improve.

Our experiments concentrated on two key benefits of the hybrid representation of task solutions as source code and neural networks. First, the source code structure imposes modularity which can be seen as *focusing the supervision*. If a component is not needed for a given task, then the differentiable interpreter can choose not to use it, which shuts off any gradients from flowing to that component. We speculate that this could be a reason for the models being resistant to catastrophic forgetting, as the model either chooses to use a classifier, or ignores it (which leaves the component unchanged). The second benefit is that learning programs imposes a bias that favors learning models that exhibit strong generalization. Additionally, the source code representation has the advantage of being interpretable by humans, allowing verification and incorporation of domain knowledge describing the shape of the problem through the source code structure.

The primary limitation of this design is that it is known that differentiable interpreters are difficult to train on problems significantly more complex than those presented here (Kurach et al., 2015; Neelakantan et al., 2016; Gaunt et al., 2016). However, if progress can be made on more robust training of differentiable interpreters (perhaps extending ideas in (Neelakantan et al., 2016) and (Feser et al., 2016)), then we believe there to be great promise in using hybrid models to build large lifelong learning systems.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian, Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jozefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mané, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul, Vanhoucke, Vincent, Vasudevan, Vijay, Viégas, Fernanda, Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

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

Bengio, Yoshua, Louradour, Jérôme, Collobert, Ronan, and Weston, Jason. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, pp. 41–48, 2009.

Bunel, Rudy, Desmaison, Alban, Kohli, Pushmeet, Torr, Philip H. S., and Kumar, M. Pawan. Adaptive neural compilation. *CoRR*, abs/1605.07969, 2016. URL `http://arxiv.org/abs/1605.07969`.

Caruana, Rich. Multitask learning. *Machine Learning*, 28: 41–75, 1997.

Chen, Zhiyuan, Ma, Nianzu, and Liu, Bing. Lifelong learning for sentiment classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 750–756, 2015.

Feser, John K., Brockschmidt, Marc, Gaunt, Alexander L., and Tarlow, Daniel. Neural functional programming. 2016. Submitted to ICLR 2017.

Gaunt, Alexander L., Brockschmidt, Marc, Singh, Rishabh, Kushman, Nate, Kohli, Pushmeet, Taylor, Jonathan, and Tarlow, Daniel. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016. URL `http://arxiv.org/abs/1608.04428`.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL `http://arxiv.org/abs/1410.5401`.

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

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. In *Proceedings of the 28th Conference on Advances in Neural Information Processing Systems (NIPS)*, pp. 1828–1836, 2015.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pp. 190–198, 2015.

Kaiser, Łukasz and Sutskever, Ilya. Neural GPUs learn algorithms. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016. URL `http://arxiv.org/abs/1511.08228`.

Kumar, Abhishek and Daume III, Hal. Learning task grouping and overlap in multi-task learning. *arXiv preprint arXiv:1206.6417*, 2012.

Kurach, Karol, Andrychowicz, Marcin, and Sutskever, Ilya. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2015. URL `http://arxiv.org/abs/1511.06392`.

Luong, Minh-Thang, Le, Quoc V, Sutskever, Ilya, Vinyals, Oriol, and Kaiser, Lukasz. Multi-task sequence to sequence learning. In *International Conference on Learning Representations (ICLR)*, 2015.

McCloskey, Michael and Cohen, Neal J. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of learning and motivation*, 24: 109–165, 1989.

Neelakantan, Arvind, Le, Quoc V., and Sutskever, Ilya. Neural programmer: Inducing latent programs with gradient descent. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2016.

Pan, Sinno Jialin and Yang, Qiang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

Price, Eric, Zaremba, Wojciech, and Sutskever, Ilya. Extensions and limitations of the neural GPU. 2016. Submitted to ICLR 2017.

Ratcliff, Roger. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97(2):285, 1990.

Reed, Scott E. and de Freitas, Nando. Neural programmer-interpreters. 2016.

Riedel, Sebastian, Bosnjak, Matko, and Rocktäschel, Tim. Programming with a differentiable forth interpreter. *CoRR*, abs/1605.06640, 2016. URL `http://arxiv.org/abs/1605.06640`.

Robins, Anthony. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

Rusu, Andrei A, Rabinowitz, Neil C, Desjardins, Guillaume, Soyer, Hubert, Kirkpatrick, James, Kavukcuoglu, Koray, Pascanu, Razvan, and Hadsell, Raia. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Ruvolo, Paul and Eaton, Eric. Ella: An efficient lifelong learning algorithm. *ICML (1)*, 28:507–515, 2013.

Shultz, Thomas R and Rivest, Francois. Knowledge-based cascade-correlation: Using knowledge to speed learning. *Connection Science*, 13(1):43–72, 2001.

Silver, Daniel L and Mercer, Robert E. The task rehearsal method of life-long learning: Overcoming impoverished data. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pp. 90–101. Springer, 2002.

Silver, Daniel L and Poirier, Ryan. Machine life-long learning with csmtl networks. In *AAAI*, 2006.

Silver, Daniel L, Yang, Qiang, and Li, Lianghao. Lifelong machine learning systems: Beyond learning algorithms. In *AAAI Spring Symposium: Lifelong Machine Learning*, pp. 49–55, 2013.

Stallkamp, Johannes, Schlipsing, Marc, Salmen, Jan, and Igel, Christian. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pp. 1453–1460, 2011.

Thrun, Sebastian. A lifelong learning perspective for mobile robot control. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, 1994.

Thrun, Sebastian. Is learning the n-th thing any easier than learning the first? In *Advances in Neural Information Processing Systems 8 (NIPS)*, pp. 640–646, 1995.

Thrun, Sebastian and O'Sullivan, Joseph. Discovering structure in multiple learning tasks: The TC algorithm. In *Machine Learning, Proceedings of the Thirteenth International Conference (ICML)*, pp. 489–497, 1996.

Weston, Jason, Chopra, Sumit, and Bordes, Antoine. Memory networks. In *Proceedings of the 3rd International Conference on Learning Representations 2015*, 2014. URL `http://arxiv.org/abs/1410.3916`.

Zaremba, Wojciech, Mikolov, Tomas, Joulin, Armand, and Fergus, Rob. Learning simple algorithms from examples. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, pp. 421–429, 2016.

Zhang, Chiyuan, Bengio, Samy, Hardt, Moritz, Recht, Benjamin, and Vinyals, Oriol. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations*, 2017.