SEQUENTIAL COORDINATION OF DEEP MODELS FOR LEARNING VISUAL ARITHMETIC

ABSTRACT

Achieving machine intelligence requires a smooth integration of perception and reasoning. Yet the models we have developed to date tend to specialize in one or the other; sophisticated manipulation of symbols acquired from rich perceptual spaces has so far proved elusive. Consider a visual arithmetic task, where an agent must learn to solve mathematical expressions, captured in natural conditions (e.g. hand-written, with background). We propose a two-tiered architecture for tackling this problem. At the lower level we leverage a collection of pre-trained deep perceptual models that can be used to detect and extract representations of characters in the image. At the higher level, we use reinforcement learning to learn when to apply the perceptual networks and what transformations to apply to their outputs. The resulting model is able to solve a variety of tasks in the Visual Arithmetic domain, and has several advantages over standard convolutional models, including greatly improved sample efficiency.

1 INTRODUCTION

Recent successes in machine learning have shown that difficult perceptual tasks can be tackled efficiently using deep neural networks, while complex decision making problems can be addressed with reinforcement learning. However, many challenging tasks involve combining both perception and reasoning. Consider for example the task of visual arithmetic where one has to perform arithmetic operations on digits from a natural image where the operation to perform is specified by some special symbol within the image (e.g. + or *). In order to solve this task, one has to first segment the image into its relevant parts and categorize them (perception) before performing the specified operation (basic arithmetic and logical reasoning). In this work, we propose a reinforcement learning algorithm to train an agent to perform high level tasks by using a pool of available modules performing simple individual tasks. For example in the visual arithmetic task, our algorithm can be used to train an agent to learn how to sequentially combine an attention mechanism, a network classifying digits and arithmetic symbols, and modules performing basic arithmetic to solve this task.

More generally, we investigate in this paper how different modules, each designed to solve a specific task, can be sequentially combined in an automated fashion to solve a higher-level task relying on the individual abilities of the available modules. The motivation for this approach is two-fold. First, cognitive science theories accounting for the entire range of human behavior often hypothesize that the brain itself has such a modular architecture. At least one prominent theory of brain function hypothesizes that the brain contains discrete modules, each performing some versatile information processing function that can be used as components of algorithms for many different tasks (Anderson, 2010). Marblestone et al. (2016) also suggest that the brain is composed of modules, and that both the individual structure and the loss function minimized by each module (assuming such modules are minimizing loss functions) depend on the information processing role it will eventually play, in contrast with the alternative of a large, architecturally homogeneous network optimizing a global loss function. Second, the modularity of our approach makes it very appealing to machine learning practitioners, limiting the need for the domain expert to decide which modules are relevant to handle the task at hand without having to explicitly specify how these modules should interact together. Furthermore, we believe that the large pool of
pre-trained models made available in the recent years could constitute an invaluable source of crucial information about the world to intelligent agents. Indeed, an agent with access to a wide range of complementary networks would be able to combine them in order to perform even more sophisticated computations, amplifying the utility of the individual networks. The agent would also be able to select the most suited network for a given task, while each network may play a useful role in solving multiple tasks, opening up the possibility of systems capable of solving a wide range of tasks while being efficient in terms of computational resources.

In this work, we primarily consider two kinds of modules: pre-trained deep neural networks and explicit logical/arithmetic modules, but our approach can be applied to the combination of arbitrary individual modules designed (or pre-trained) to achieve some specific sub-tasks. We propose a reinforcement learning algorithm to train an agent to figure out how to solve a given high-level task from training data by sequentially combining models from a pool of available modules.

**Contributions.** We propose a novel recipe for constructing agents capable of solving complex tasks by sequentially combining provided modules. The role of the system designer is limited to choosing a pool of modules and gathering training data in the form of input-output examples for the task at hand, which we believe makes our approach particularly relevant to the machine learning practitioner. To achieve this, we provide the agent with representations of the world that are well suited to the task at hand by specifying a pool of relevant modules, and train it using an actor critic algorithm. We evaluate our model on visual arithmetic tasks where the agent has to perform arithmetic reduction operations on digits in an image. Our experiments show that the proposed model can efficiently learn to solve tasks in this domain, including a difficult task whose solution requires many sequential steps.

## 2 Technical Methods

### 2.1 Markov Decision Processes

Our approach makes use of standard reinforcement learning formalisms Sutton and Barto (1998). The external world is modelled as a Partially Observable Markov Decision Process (POMDP), $\mathcal{E}$. Each timestep $\mathcal{E}$ is in a state $s_t$, based upon which it omits an observation $o_t$ that is sent to the learning agent. The agent responds with an action $a_t$, which causes the environment to emit a reward $r_t$. Finally, the state is stochastically updated according to $\mathcal{E}$’s dynamics, $s_{t+1} \sim P(\cdot|s_t, a_t)$ and the process repeats. The agent is assumed to choose $a_t$ according to a parameterized policy that maps from observation-action histories to distributions over actions, i.e. $a_t \sim \pi_\theta(\cdot|h_t)$, where $h_t = o_0, a_0, \ldots, o_t$ is the sequence of observable events up to time $t$, and $\pi_\theta$ is assumed to be differentiable with respect to its parameter vector $\theta$.

We also borrow the notion of an interface as proposed in Zaremba et al. (2016). An interface is a designed, task-specific machine that mediates the learning agent’s interaction with the external world, providing the agent with a representation (observation and action spaces) which is intended to be more conducive to learning than the raw representations. In this work we formalize an interface as a separate POMDP $\mathcal{I}$ with its own state, observation and action spaces. Each time step $\mathcal{E}$ sends an observation to $\mathcal{I}$, which potentially alters its state, and $\mathcal{I}$ emits its own observation to the agent. When the agent responds with an action, it is first processed by $\mathcal{I}$, once again possibly changing its state, after which $\mathcal{I}$ sends an action to $\mathcal{E}$. The agent may thus be regarded as interacting with a POMDP $\mathcal{C}$ constituting the combination of $\mathcal{E}$ and $\mathcal{I}$. $\mathcal{C}$’s observation and action spaces are the same as $\mathcal{I}$, its state is the concatenation of the states of $\mathcal{I}$ and $\mathcal{E}$, and its dynamics are determined by the nature of the interaction between $\mathcal{I}$ and $\mathcal{E}$.

Zaremba et al. (2016) learn to control interfaces in order to solve purely algorithmic tasks, such as copying lists of abstractly (rather than perceptually) represented digits. One of the main insights of our paper is that the idea of interfaces can be extended to tasks with rich perceptual domains by incorporating pre-trained deep networks to handle the perceptual components.
2.2 Actor-Critic Reinforcement Learning

We train agents using a variant of the actor critic algorithm (see e.g. (Sutton and Barto, 1998; Degris et al., 2012; Mnih et al., 2016)). The agent’s goal is to maximize its expected sum of rewards:

\[ J(\theta) = E_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} r_t \right], \]

where \( \tau \) is a trajectory sampled using \( \pi_\theta \) and \( r_0, \ldots, r_T \) are the rewards experienced along that trajectory. We maximize this objective using gradient ascent. It is not immediately clear how to take its gradient, since the probability of a trajectory depends on the environment dynamics \( P(s_{t+1}|s_t, a_t) \) which are unknown. It can be shown that an unbiased estimate of \( \nabla_\theta J(\cdot) \) can be obtained by differentiating a surrogate loss function:

\[ L(\theta) = E_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} \log(\pi_\theta(a_t|h_t)) \left( \sum_{i=t}^{T} r_i \right) \right]. \]

(2)

This is now tractable, since all required quantities are either known or can be estimated from sample trajectories. The standard REINFORCE algorithm consists in first sampling a batch of trajectories using \( \pi_\theta \), from which an estimate \( \hat{L}(\theta) \) of \( L(\theta) \) is computed (Williams, 1992). \( \nabla_\theta \hat{L}(\theta) \) is then computed, and the parameter vector \( \theta \) updated using standard gradient ascent or one of its more sophisticated counterparts.

In order to decrease the variance of the gradient estimates, a baseline function \( b_t(h_t) \) can be introduced into Equation (2), yielding an improved surrogate loss function:

\[ L(\theta) = E_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} \log(\pi_\theta(a_t|h_t)) \left( \sum_{i=t}^{T} r_i - b_t(h_t) \right) \right]. \]

(3)

It can be shown that including \( b_t(\cdot) \) does not bias the gradient estimate and may lower its variance if chosen appropriately. The state-value function for the current policy, \( V^\pi_\theta(h) = E_{\tau \sim \pi_\theta} [r_t|h_t = h] \), is a good choice for \( b_t \) (Sutton et al., 2000) — however this will rarely be known. A typical compromise is to learn a parametric estimate \( V^\omega(h) \) of \( V^\pi_\theta(h) \) concurrently with learning our policy. Here we follow an additional standard technique of having \( V^\omega(h) \) share the majority of its parameters with \( \pi_\theta \), i.e. \( \omega = \theta \) (Mnih et al., 2016). This allows the agent to learn useful internal representations even in the absence of reward, which often speeds up learning. The value function is trained by minimizing an objective of the form \( \sum_{t=0}^{T} \| \sum_{i=t}^{T} r_i - V^\omega(h_t) \|^2 \) estimated from trajectories.

2.3 Task Domain: Visual Arithmetic

We consider a set of visual arithmetic tasks (see Fig. 2, right) in which each input example is an image containing a random number of MNIST digits and possibly a symbol indicating a reduction operation to perform on the digits (e.g. \( \cdot \) means take the product , \( + \) means take the sum, etc.).

2.4 Sequential Coordination of Deep Models

We construct our solution as follows. We first identify a collection of basic information processing modules that we predict will be useful to a sequential agent attempting to solve the task. We can immediately see that it may be useful to have modules performing the following functions:

1. Detect and attend to salient locations in the image.
2. Classify digits and other mathematical symbols in the attended region.
3. Manipulate symbols to produce an answer.
We can select basic modules, including pre-trained deep perceptual models, to construct an interface, which will be controlled by a reinforcement learning agent. A sample interface for Visual Arithmetic is depicted in Figure 2.

Figure 1: Example input images from the Visual Arithmetic tasks studied in the experiments section. Top: Any of the 4 single-operation tasks (Sum, Prod, Max, Min). Correct answer depends on what task they are being used in. Bottom: Combined task. Correct answers from left to right are: 2, 5, 27, 10.

3 Experiments

In this section, we consider a number of different tasks within the Visual Arithmetic domain, all of which make use of the interface described previously.

The interface for this domain includes 3 pre-trained deep neural networks. Two of these are instances of LeNet (LeCun et al., 1998), each consisting of two convolutional/max-pool layers followed by a fully connected layer with 128 hidden units and RELU activations. One, the op classifier, is pretrained to recognize capital letters from the Extended MNIST dataset (Cohen et al., 2017) (which, in the some tasks we consider, are used as symbols to indicate the operation to perform on the digits). The other, the digit classifier, is pretrained to recognize MNIST digits. The third network is the salience detector, a multilayer perceptron with 3 hidden layers of 100 units each and RELU as the non-linearity. The salience network is pre-trained to output a salience map when given as input scenes consisting of randomly scattered Extended MNIST characters (both letters and digits).

For the policy $\pi_{\theta}$, we employ a Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) with 128 hidden...
units. This network accepts observations provided by the interface (see Figure ??) as input, and yields as output a distribution over the interface's action space as well as an estimate of the expected future return from the current state under the current policy (i.e. a value estimate). The weights of the LSTM are updated according to the actor-critic algorithm discussed in Section 2.2.

In all experiments, we compare against convolutional networks trained using cross-entropy loss, the de facto standard approach for solving visual tasks with deep learning. In particular, we experiment with instances of LeNet with 32, 128 or 512 units in the fully connected layer. More complex feedforward networks of course exist, but these will likely require much larger amounts of data to train, and here we are primarily concerned with performance at very small sample sizes. We treat all tasks as classification tasks with 101 possible output labels. The first 100 classes correspond to the integers 0-99. All integers greater than or equal to 100 (e.g. when multiplying 3 digits) are subsumed under the 101st class.

Our experiments look primarily at how performance is influenced by the number of training samples. For all sample sizes, training the RL algorithm incorporates experience replay, performing many more experiences using its interface to solve the problem, but all of those experiences operate on the small provided set of input-output training samples, and does not assume access to a simulator of the external environment.

3.1 Single Operation Tasks

In the initial set of tasks, the input image contains a randomly selected number of MNIST digits (2 or 3) placed randomly on a 2x2 grid, and the learner is required to output an answer that is a function of the displayed digits as well as the task being performed. We consider 4 variants: Sum, Product, Maximum and Minimum. Example input images are shown in Figure 1, Row A. The RL agent runs for $T = 30$ timesteps, after which the contents of the interface's store field (see Figure 2) is taken to be its answer. During training, the agent receives a reward of 0 for each time step on which its store field contains the correct answer, and $-1/T$ otherwise. This serves to encourage the agent to obtain the answer quickly, and provides a dense training signal. Additionally, on the final timestep a reward of 0 is provided if store contains the correct answer, and $-1$ otherwise. No reward is provided during testing. Experiments showing the sample efficiency of the candidate models are shown in Figure 3. The RL approach is able to quickly leverage control structure over pre-trained models in order to achieve good performance at significantly smaller sample sizes.

3.2 Combined Task

We next consider a task that combines the four tasks from the previous section. Each input example now contains a capital Extended MNIST character in addition to the 2 to 3 digits. The character indicates the operation that must be performed on the digits: $A$ indicates add/sum, $M$ indicates multiplication/product, $X$ indicates maximum, $N$ indicates minimum. Example input images are shown in Figure 1, Row B. Solving this task requires much more sophisticated control; the agent needs to learn not only how to perform the required operations, but also how to determine what the required operation is, which intuitively favours a form of sequential processing. In Figure 4 we probe the sample efficiency of the candidate models, and once again find that the sequential solution outperforms the convolutional networks by a large margin.

4 Related Work

Production Systems & Cognitive Architectures. The idea of using a central controller to co-ordinate the activity of a collection of information processing components has its roots in research on production systems in classical artificial intelligence, and cognitive architectures in cognitive science. Production systems have a rich history in cognitive science and artificial intelligence, dating back to Newell and Simon’s pioneering work on the study of high-level human problem solving, manifested most clearly in the General Problem Solver (GPS;
Figure 3: Probing sample efficiency on the four single operation tasks (Sum, Product, Maximum, Minimum). At small sample sizes, the reinforcement learning agent is able to leverage the provided interface to achieve significantly lower error than the convolutional neural networks.
Newell and Simon, 1963). GPS sought to explain human cognition, but was also taken
up enthusiastically by the AI community, evidenced by its central role in Expert Systems
research, a mainstay of AI in the 1980’s (Boden, 2006). While production systems have fallen
out of favour in mainstream AI, they still enjoy a strong following in the cognitive science
community, forming the core of nearly every prominent cognitive architecture including
ACT-R (Anderson, 2009), SOAR (Newell, 1994; Laird, 2012), and EPIC (Kieras and Meyer,
1997). We see our contribution as taking a computational mechanism that has been shown to
be sufficient for solving some of the high-level problems that humans are capable of solving
(Newell and Simon, 1972), and adapting it to take advantage of the wealth of advances
made in recent years in optimizing the parameters of computation graphs (e.g. automatic
differentiation, an arsenal of effective heuristics, improved optimization theory, vastly more
powerful hardware).

Sequential Models for Supervised Learning. A related body of work concerns
sequential models applied to supervised learning problems, such as Mnih et al.
(2014), which typically have an algorithmic flavour while still involving a signif-
icant perceptual component. Our framework may be regarded as providing a recipe
for building similar models while placing greater emphasis on modularity, multi-task
learning, and efficiency via module reuse.

Neural Abstract Machines. One obvious point of comparison to the current work is
recent research on deep neural networks designed to learn to carry out algorithms
on sequences of discrete symbols. Some of these frameworks, including the Differenti-
table Forth Interpreter (Riedel and Rocktäschel, 2016) and TerpreT (Gaunt et al., 2016b), achieve this by explicitly generating code, while others, including the Neural Turing Machine (NTM; Graves et al., 2014), Neural Random-Access Machine (NRAM; Kurach et al., 2015), Neural Programmer (NP; Neelakantan et al., 2015), Neural Programmer-Interpreter (NPI; Reed and De Freitas, 2015) and work in Zaremba et al. (2016) on learning algorithms using reinforcement learning, avoid generating code and generally consist of a controller network that learns to perform actions in a (sometimes differentiable) external computational medium in order to carry out an algorithm.

Our approach is most similar to the latter category, the main difference being that we have
elected not to require the external computational medium (the interface in the vocabulary of
this article) to be differentiable. Our work can be regarded as generalizing the form of the
external computational medium, and placing greater emphasis on tasks that require sophis-
ticated perception and perceptual control (i.e. attention) in addition to posing algorithmic
challenges. Indeed, while past work in this space has largely focused on sequence-to-
sequence learning on sequences of discrete symbols (though see Gaunt et al. (2016a) for an
exception that employs code-generation), our framework targets tasks wherein the inputs
are presented in a less direct matter (e.g. visually).

Modular Neural Networks. Work outside the neural computation literature has focused
more directly on the use of neural modules and adaptively choosing groups of modules
to apply depending on the input. End-to-End Module Networks (Hu et al., 2017) use
reinforcement learning to train a recurrent neural network to lay out a feedforward neural
network composed of elements of a stored library of neural modules (which are themselves
learnable). Our work differs in that rather than having the layout of the network depend
solely on the input, the module applied at each stage (i.e. the topology of the network) may
depend on past module applications within the same episode, since a decision about which
module/rule to apply is made at every time step. Systems built using our framework can,
for example, use their modules to gather information about the environment in order to decide which modules to apply at a later time.

In the Deep Sequential Neural Networks (DSNN Denoyer and Gallinari, 2014), each edge of a fixed directed acyclic graph (DAG) is a trainable neural module. Running the network on an input consists in moving through the DAG starting from the root while maintaining a feature vector which is repeatedly transformed by the neural modules associated with the traversed edges. At each node of the DAG, an outgoing edge is stochastically selected for traversal by a learned controller which takes the current features as input. This differs from our work, where each module may be applied many times rather than just once as is the case for the entirely feedforward DSNNs (where no module appears more than once in any path through the DAG connecting the input to the output).

Finally, there is the recent PathNet (Fernando et al., 2017). An important difference is that different modules are recruited for the entire duration of the task, rather than on a more fine-grained step-by-step basis.

5 Discussion

There are number of possible future directions related to the current work, including potential benefits that were not explored here. These include the ability to take advantage of conditional computation; in principle, only the subset of the interface needed to carry out the chosen action needs to be executed every time step. If the interface contains many large networks or other computationally intensive modules, large speedups could likely be realized along these lines. A related idea is that of adaptive computation time; in the current work, all episodes ran for a fixed number of time steps, but it should be possible to have the agent decide when it has the correct answer and stop computation at that point, saving valuable computational resources. Furthermore, it may be beneficial to train the neural modules concurrently with training the controller agent, so that they may adapt to better perform the uses that the agent finds for them. Finally, the generality of reinforcement learning and its ability to make use of discrete and non-differentiable modules opens up a vast array of possible modules that could be included in the interface; for instance, discrete knowledge bases may serve as a long term memory.

Because deep networks are differentiable with respect to their inputs, an alternative to reinforcement learning would be to make the entire system differentiable and train it end-to-end using backpropagation. A differentiable approximation to discrete action selection can be achieved using the Gumbel-Softmax trick (Maddison et al., 2016; Jang et al., 2016), and in some cases smooth approximations may be found for non-differentiable internal transformations. We briefly experimented with this approach, but were unable to make it work. When using Gumbel-Softmax, a stochastic convex combination of the available actions is executed each time step. We found that this made it difficult to design interfaces, in comparison to the discrete action selection used in reinforcement learning, since each time step sees multiple actions being executed, each to differing degrees, making actions interfere with one another and making their effects difficult to reason about. Moreover, in training our models, we found that choosing appropriate exploration strategies was a key factor in making things work, and, going forward, it will be useful to be able to leverage the considerable range of sophisticated exploration strategies that have been developed for reinforcement learning (e.g. Bellemare et al. (2016); Osband et al. (2016)).

6 Conclusion

We have suggested a novel approach for solving tasks that require both sophisticated perception and multiple computational/algorithmic steps. This approach consists in designing an interface between the world and a learning agent that contains pre-trained deep neural networks for processing perceptual data, as well as modules for manipulating stored symbolic representations. Reinforcement learning is then used to train a controller agent to use the interface to solve tasks.
Any generally intelligent system will need many individual competencies at its disposal, both perceptual and algorithmic; in this work we have proposed one path by which a system may learn to coordinate such competencies.

REFERENCES


