Learning Simple Algorithms from Examples

Wojciech Zaremba[§] WOJ.ZAREMBA@GMAIL.COM
New York University

Tomas Mikolov **Armand Joulin** **Rob Fergus** {TMIKOLOV,AJOULIN,ROBFERGUS}@FB.COM
Facebook AI Research

Abstract

We present an approach for learning simple algorithms such as copying, multi-digit addition and single digit multiplication directly from examples. Our framework consists of a set of *interfaces*, accessed by a *controller*. Typical interfaces are 1-D tapes or 2-D grids that hold the input and output data. For the controller, we explore a range of neural network-based models which vary in their ability to abstract the underlying algorithm from training instances and generalize to test examples with many thousands of digits. The controller is trained using Q-learning with several enhancements and we show that the bottleneck is in the capabilities of the controller rather than in the search incurred by Q-learning.

1. Introduction

Many every day tasks require a multi-step interaction with the world. For example, picking an apple from a tree requires visual localization of the apple; extending the arm and then fine muscle control, guided by visual feedback, to pluck it from the tree. While each individual procedure is not complex, the task nevertheless requires careful sequencing of operations across both visual and motor systems.

This paper explores how machines can learn algorithms involving a similar compositional structure. Since our emphasis is on learning the correct sequence of operations, we consider the domain of arithmetic where the operations themselves are very simple. For example, although learning to add two digits is straightforward, solving addition of two multi-digit numbers requires precise coordination of this operation with movement over the sequence and recording of the carry. We explore a variety of algorithms in this domain, including complex tasks involving addition and multiplication.

Our approach formalizes the notion of a central controller that interacts with the world via a set of interfaces, appropriate to the task at hand. The controller is a neural network model which must learn to control the interfaces, via a set of discrete actions (e.g. "move input tape left", "read", "write symbol to output tape", "write nothing this time step") to produce the correct output for given input patterns. Specifically, we train the controller from large sets of examples of input and output patterns using reinforcement learning. Our reward signal is sparse, only being received when the model emits the correct symbol on the output tape.

We consider two separate settings. In the first, we provide supervision in the form of ground truth actions. In the second, we train only with input-output pairs (i.e. no supervision over actions). While we are able to solve all the tasks in the latter case, the supervised setting provides insights about the model limitations and an upper bound on the performance. We evaluate our model on sequences far longer than those present during training. Surprisingly, we find that controllers with even modest capacity to recall previous states can easily overfit the short training sequences and not generalize to the test examples, even if the correct actions are provided. Even with an appropriate controller, off-the-shelf Q-learning fails on the majority of our tasks. We therefore introduce a series of modifications that dramatically improve performance. These include: (i) a novel dynamic discount term that makes the reward invariant to the sequence length; (ii) an extra penalty that aids generalization and (iii) the deployment of Watkins Q-lambda (Sutton & Barto, 1998).

2. Model

Our model consists of an RNN-based controller that accesses the environment through a series of pre-defined in-

[§]The author is now at OpenAI.

Proceedings of the 33^{rd} International Conference on Machine Learning, New York, NY, USA, 2016. JMLR: W&CP volume 48.

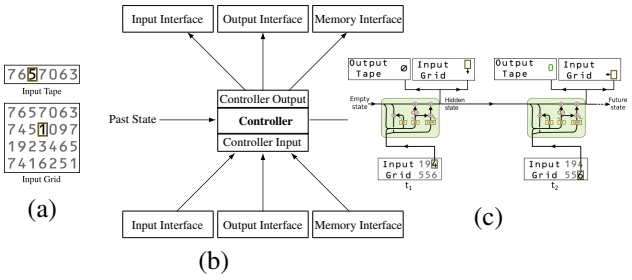

Figure 1. (a): The input tape and grid interfaces. Both have a single head (gray box) that reads one character at a time, in response to a *read* action from the controller. It can also move the location of the head with the *left* and *right* (and *up*, *down*) actions. (b) An overview of the model, showing the abstraction of controller and a set of interfaces (in our experiments the memory interface is not used). (c) An example of the model applied to the addition task. At time step t_1, the controller, a form of RNN, reads the symbol 4 from the input grid and outputs a no-operation symbol ($\oslash$) on the output tape and a *down* action on the input interface, as well as passing the hidden state to the next timestep.

terfaces. Each interface has a specific structure and set of actions it can perform. The interfaces are manually selected according to the task (see Section 3). The controller is the only part of the system that learns and has no prior knowledge of how the interfaces operate. Thus the controller must learn the sequence of actions over the various interfaces that allow it to solve a task. We make use of three different interfaces:

Input Tape: This provides access to the input data symbols stored on an "infinite" 1-D tape. A read head accesses a single character at a time through the *read* action. The head can be moved via the *left* and *right* actions.

Input Grid: This is a 2D version of the input tape where the read head can now be moved by actions *up*, *down*, *left* and *right*.

Output Tape: This is similar to the input tape, except that the head now writes a single symbol at a time to the tape, as provided the controller. The vocabulary includes a no-operation symbol (NOP) enabling the controller to defer output if it desires. During training, the written and target symbols are compared using a cross-entropy loss. This provides a differentiable learning signal that is used in addition to the sparse reward signal provided by the Q-learning.

Fig. 1(a) shows examples of the input tape and grid interfaces. Fig. 1(b) gives an overview of our controller–interface abstraction and Fig. 1(c) shows an example of this on the addition task (for two time steps).

For the controller, we explore several recurrent neural network architectures: two different sizes of 1-layer LSTM (Hochreiter & Schmidhuber, 1997), a gated-recurrent unit (GRU)(Cho et al., 2014) and a vanilla feed-forward network. Note that RNN-based models are able to remember previous network state, unlike the the feed-forward network. This is important because some tasks explicitly require some form of memory, e.g. the carry in addition.

The simple algorithms we consider (see Section 3) have deterministic solutions that can be expressed as a finite state automata. Thus during training we hope the controller will implicitly learn the correct automata from the training samples, since this would ensure generalization to sequences of arbitrary length. On some tasks like reverse, we observe a higher-order form of over-fitting: the model learns to solve the training tasks correctly and generalizes successfully to test sequences of the same length (thus is not over-fitting in the standard sense). However, when presented with longer test sequences the model fails completely. This suggests that the model has converged to an incorrect local minima, one corresponding to an alternate automata which have an implicit awareness of the sequence length of which they were trained. See Fig. 4 for an example of this on the reverse task. Note that this behavior results from the controller, not the learning scheme, since it is present in both the supervised (Section 5) and Q-learning settings (Section 6). These experiments show the need to carefully adjust the controller capacity to prevent it learning any dependencies on the length of training sequences, yet ensuring it has enough state to implement the algorithm in question.

As illustrated in Fig. 1(c), the controller passes two signals to the output tape: a discrete action (move left, move right, write something) and a symbol from the vocabulary. This symbol is produced by taking the max from the softmax output on the top of the controller. In training, two different signals are computed from this: (i) a cross-entropy loss is used to compare the softmax output to the target symbol and (ii) a discrete 1/0 reward if the symbol is correct/incorrect. The first signal gives a continuous gradient to update the controller parameters via backpropagation. Leveraging the reward requires reinforcement learning, since many actions might occur before a symbol is written to the output tape. Thus the action output of the controller is trained with reinforcement learning and the symbol output is trained by backpropagation.

3. Tasks

We consider six different tasks: copy, reverse, walk, multi-digit addition, 3 number addition and single digit multiplication. The input interface for copy and reverse is an input tape, but an input grid for the others. All tasks use an output tape interface. Unless otherwise stated, all arithmetic operations use base 10. Examples of the six tasks are shown in Fig. 2.

Copy: This task involves copying the symbols from the input tape to the output tape. Although simple, the model still has to learn the correspondence between input and output symbols, as well as executing the move right action on the input tape.

Reverse: Here the goal is to reverse a sequence of symbols on the input tape. We provide a special character "r" to indicate the end of the sequence. The model must learn to move right multiple times until it hits the "r" symbol, then move to the left, copying the symbols to the output tape.

Walk: The goal is to copy symbols, according to the directions given by an arrow symbol. The controller starts by moving to the right (suppressing prediction) until reaching one of the symbols $\uparrow, \downarrow, \leftarrow$. Then it should change it's direction accordingly, and copy all symbols encountered to the output tape.

Addition: The goal is to add two multi-digit sequences, provided on an input grid. The sequences are provided in two adjacent rows, with their right edges aligned. The initial position of the read head is the last digit of the top number (i.e. upper-right corner). The model has to: (i) memorize an addition table for pairs of digits; (ii) learn how to move over the input grid and (iii) discover the concept of a carry.

3 Number Addition: As for the addition task, but now three numbers are to be added. This is more challenging as the reward signal is less frequent (since more correct actions must be completed before a correct output digit can be produced). Also the carry now can take on three states (0, 1 and 2), compared with two for the 2 number addition task.

Single Digit Multiplication: This involves multiplying a single digit with a long multi-digit number. It is of similar complexity to the 2 number addition task, except that the carry can take on more values $\in [0, 8]$.

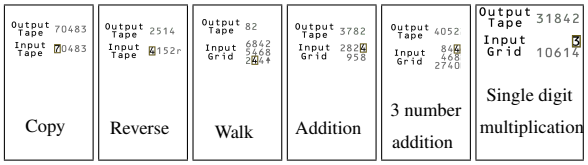

Figure 2. Examples of the six tasks, presented in their initial state. The yellow box indicates the starting position of the read head on the Input Interface. The gray characters on the Output Tape are **target** symbols used in training.

In Table 1, we examine the feasibility of solving them by exhaustively searching over all possibile automata. For tasks involving addition and multiplication, this approach is not practical. We thus explore a range of learning-based approaches.

Task	#states	#possible automata
Copying	1	1
Reverse	2	4
Walk	4	4096
Addition	30	10^{262}
3-rows Addition	50	10^{737}
Single Digit Multiplication	20	10^{114}

Table 1. All six of our tasks can be solved by a finite-state automata. We estimate size of the automata for each task, and the time necessary to find it by exhaustive search. The model is in a single state at any given time and the current input, together with model state, determines the output actions and new state. For instance, addition has to store: (i) the current position on the grid (up, down after coming from the top, down after coming from the right) and (ii) the previous number with accumulated curry. All combinations of these properties can occur, and the automata must have sufficient number of states to distinguish them. The number of possible directed graphs for a given number of states is $4^{n*(n-1)/2}$. Thus exhaustive search is impractical for all but the simplest tasks.

4. Related Work

A variety of recent work has explored the learning of simple algorithms. Many of them are different embodiments of the controller-interface abstraction formalized in our model. The Neural Turing Machine (NTM) (Graves et al., 2014) uses a modified LSTM (Hochreiter & Schmidhuber, 1997; Gers et al., 2003) as the controller, and has three inferences: sequential input, delayed output and a differentiable memory. The model is able to learn simple algorithms including copying and sorting. The Stack RNN (Joulin & Mikolov, 2015) has an RNN controller and three interfaces: sequential input, a stack memory and sequential output. The learning of simple binary patterns and regular expressions is demonstrated. A closely related work to this is (Das et al., 1992), which was recently extended in the Neural DeQue (Grefenstette et al., 2015) to use a list instead. End-to-End Memory Networks (Sukhbaatar et al., 2015) use a feed-forward network as the controller and interfaces consisting of a soft-attention input, plus a delayed output (by a fixed number of "hops"). The model is applied to simple Q&A tasks, some of which involve logical reasoning. In contrast, our model automatically determines when to produce output and uses more general interfaces. Williams and Zipser (Williams & Zipser, 1989) proposed one of the first approaches that aimed to directly learn a Turing Machine, demonstrating that it could solve the parenthesis balancing problem. However, all these models are not directly comparable as they are tightly coupled with their interfaces. For instance, the Stack RNN is based on a stack, therefore it cannot be executed with a grid interface.

However, most of these approaches use continuous interfaces that permit training via back-propagation of gradients. Our approach differs in that it uses discrete interfaces thus is more challenging to train since as we must rely

on reinforcement learning instead. A notable exception is the Reinforcement Learning Neural Turing Machine (RL-NTM) (Zaremba & Sutskever, 2015) which is a version of the NTM with discrete interfaces. The Stack-RNN (Joulin & Mikolov, 2015) also uses a discrete search procedure for its interfaces but it is unclear how this would scale to larger problems.

The problem of learning algorithms has its origins in the field of program induction (Nordin, 1997; Liang et al., 2013; Wineberg & Oppacher, 1994; Solomonoff, 1964). In this domain, the model has to infer the source code of a program that solves a given problem. This is a similar goal to ours, but in quite a different setting. I.e. we do not produce a computer program, but rather a neural net that can operate with interfaces such as tapes and so implements the program without being human-readable. A more relevant work is (Schmidhuber, 2004) which learns an algorithms for the Hanoi tower problem, using a simple form of program induction and incremental learning components (this method has strong asymptotic guarantees). Genetic programming (Holland, 1992; Goldberg, 1989; Gomez et al., 2008) also can be considered a form of program induction, but the underlying algorithm relies on random mutations rather than gradient information.

Similar to (Mnih et al., 2013), we train the controller to approximate the Q-function. However, we introduce several modifications on top of the classical Q-learning. First, we use Watkins $Q(\lambda)$ (Watkins, 1989; Sutton & Barto, 1998). This helps to overcome a *non-stationary* environment (previously pointed out by (Loch & Singh, 1998)). Second, we reparametrized Q function, to become invariant to the sequence length. Finally, we penalize $||Q(s, \bullet)||$, which might help to remove positive bias (Hasselt, 2010).

5. Supervised Experiments

To understand the behavior of our model and to provide an upper bound on performance, we train our model in a supervised setting, i.e. where the ground truth actions are provided. Note that the controller must still learn which symbol to output. But this now can be done purely with backpropagation since the actions are known.

To facilitate comparisons of difficulty between tasks, we use a common measure of *complexity*, corresponding to the number of time steps required to solve each task (using the ground truth actions[*]). For instance, a reverse task involving a sequence of length 10 requires 20 time-steps (10 steps to move to the "r" and 10 steps to move back to the start). The conversion factors between sequence lengths and complexity are as follows: copy=1; reverse=2; walk=1; addition=2; 3 row addition=3 and single digit multiplication=1.

For each task, we train a separate model, starting with sequences of complexity 6 and incrementing by 4 once it achieves 100% accuracy on held-out examples of the current length. Training stops once the model successfully generalizes to examples of complexity 1000. Three different cores for the controllers are explored: (i) a 200 unit, 1-layer LSTM; (iii) a 200 unit, 1-layer GRU model and (iii) a 200 unit, 1-layer feed-forward network. An additional linear layer is placed on top of these model that maps the hidden state to either action for a given interface, or the target symbol.

In Fig. 3 we show the accuracy of the different controllers on the six tasks for test instances of increasing complexity, up to $20,000$ time-steps. The simple feed-forward controller generalizes perfectly on the copy, reverse and walk tasks but completely fails on the remaining ones, due to a lack of required memory[†]. The RNN-based controllers succeed to varying degrees, although some variability in performance is observed.

Further insight can be obtained by examining the internal state of the controller. To do this, we compute the autocorrelation matrix[‡] A of the network state over time when the model is processing a reverse task example of length 35, having been trained on sequences of length 10 or shorter. For this problem there should be two distinct states: move right until "r" is reached and then move left to the start. Fig. 2 plots A for models with three different controllers. The larger the controller capacity, the less similar the states are within the two phases of execution, showing how it has not captured the correct algorithm. The figure also shows the confidence in the two actions over time. In the case of the high capacity models, the initial confidence in the move left action is high, but this drops off after moving along the sequence. This is because the controller has learned during training that it should change direction after at most 10 steps. Consequently, the unexpectedly long test sequence makes it unsure of what the correct action is. By contrast, the simple feed-forward controller does not show this behavior since it is stateless, thus has no capacity to know where it is within a sequence. The equivalent automata is shown in Fig. 4(a), while Fig. 4(b) shows the incorrect

[*]In practice, multiple solutions can exist (see Section 6.5), thus the measure is approximate.

[†]Ammending the interfaces to allow both reading and writing on the same interface would provide a mechanism for long-term memory, even with a feed-forward controller. But then the same lack of generalization issues (encountered with more powerful controllers) would become an issue.

[‡]Let h_i be the controller state at time i, then the autocorrelation $A_{i,j}$ between time-steps i and j is given by $A_{i,j} = \frac{\langle h_i - E, h_j - E \rangle}{\sigma^2}, i, j = 1, \ldots, T$ where $E = \frac{\sum_{k=1}^{T} h_k}{T}, \sigma^2 = \frac{\sum_{k=1}^{T} \langle h_k - E, h_k - E \rangle}{T}$. T is the number of time steps (i.e. complexity).

time-dependent automata learned by the over-expressive RNN-based controllers. We note that this argument is empirically supported by our results in Table 3, as well as related work such as (Graves et al., 2014) and (Joulin & Mikolov, 2015) which found limited capacity controllers to be most effective. For example, in the latter case, the counting and memorization tasks used controllers with just 40 and 100 units respectively.

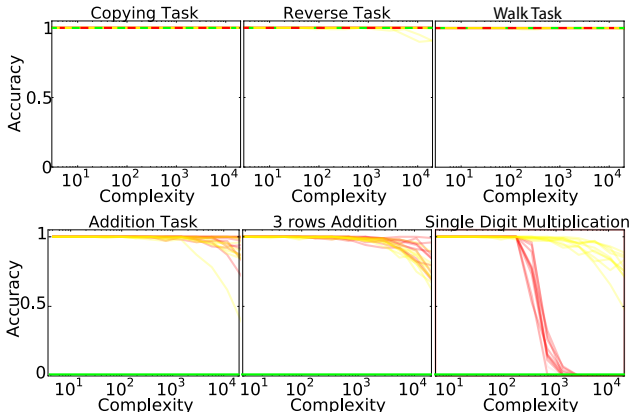

Figure 3. Test accuracy for all tasks with **supervised** actions over 10 runs for feed-forward (green), GRU (red) and LSTM (yellow) controllers. In this setting the optimal policy is provided. Complexity is the number of time steps required to compute the solution. Every task has slightly different conversion factor between complexity and the sequence length: a complexity of 10^4 for copy and walk would mean 10^4 input symbols; for reverse would correspond to $\frac{10^4}{2}$ input symbols; for addition would involve two $\frac{10^4}{2}$ long numbers; for 3 row addition would involve three $\frac{10^4}{3}$ long numbers and for single digit multiplication would involve a single 10^4 long number.

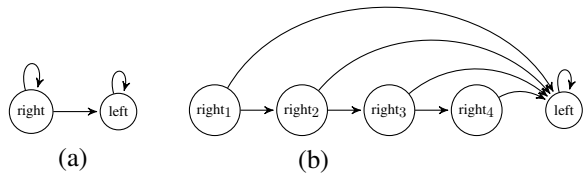

Figure 4. (a): The automata describing the correct solution to the reverse problem. The model first has to go to the right while suppressing prediction. Then, it has to go to the left and predict what it sees at the given moment (this figure illustrates only actions over the Input Tape). (b) Another automata that solves the reverse problem for short sequences, but does not generalize to arbitrary length sequences, unlike (a). Expressive models like LSTMs tend to learn such incorrect automata.

6. Q-Learning

In the previous section, we assumed that the optimal controller actions were given during training. This meant only the output symbols need to be predicted and these could be learned via backpropagation. We now consider the setting where the actions are also learned, to test the true capabilities of the models to learn simple algorithms from pairs of input and output sequences.

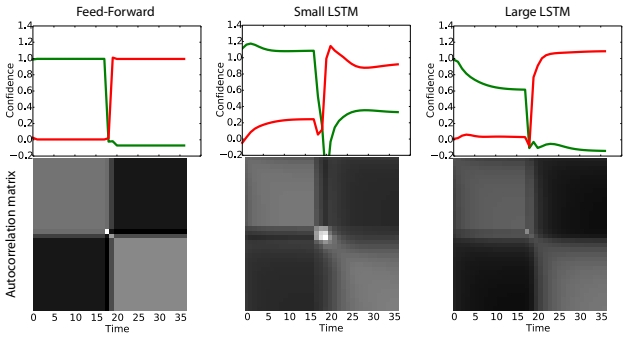

Table 2. Three models with different controllers (feed-forward, 200 unit LSTM and 400 unit LSTM) trained on the reverse task and applied to a 20 digit test example. The top row shows confidence values for the two actions on the input tape: move left (green) and move right (red) as a function of time. The correct model should be equivalent to a two-state automata (Fig. 4), thus we expect to see the controller hidden state occupy two distinct values. The autocorrelation matrices (whose axes are also time) show this to be the case for the feed-forward model – two distinct blocks of high correlation. However, for the LSTM controllers, this structure is only loosely present in the matrix, indicating that they have failed to learn the correct algorithm.

We use Q-learning, a standard reinforcement learning algorithm to learn a sequence of discrete actions that solves a problem. A function Q, the estimated sum of future rewards, is updated during training according to:

$$Q_{t+1}(s,a) = Q_t(s,a) \tag{1}$$
$$- \alpha\big[Q_t(s,a) - \big(R(s') + \gamma \max_a Q_n(s',a)\big)\big] \tag{2}$$

Taking the action a in state s causes a transition to state s', which in our case is deterministic. $R(s')$ is the reward experienced in the state s'. The discount factor is γ and α is the learning rate. The another commonly considered quantity is $V(s) = \max_a Q(s,a)$. V is called the value function, and $V(s)$ is the expected sum of future rewards starting from the state s. Moreover, Q^* and V^* are function values for the optimal policy.

Our controller receives a reward of 1 every time it correctly predicts a digit (and 0 otherwise). Since the overall solution to the task requires all digits to be correct, we terminate a training episode as soon as an incorrect prediction is made. This learning environment is *non-stationary*, since even if the model initially picks the right actions, the symbol prediction is unlikely to be correct, so the model receives no reward. But further on in training, when the symbol prediction is more reliable, the correct action will be re-

warded[§]. This is important because reinforcement learning algorithms assume stationarity of the environment, which is not true in our case. Learning in non-stationary environments is not well understood and there are no definitive methods to deal with it. However, empirically we find that this non-stationarity can be partially addressed by the use of Watkins $Q(\lambda)$ (Watkins, 1989), as detailed in Section 6.2.

6.1. Dynamic Discount

The purpose of the reinforcement learning is to learn a policy that yields the highest sum of the future rewards. Q-learning does it by learning a Q-function. The optimal policy can be extracted by taking argmax over $Q(s, \bullet)$. Note that shifting or scaling Q induces the same policy. We propose to dynamically rescale Q so (i) it is independent of the length of the episode and (ii) Q is within a small range, making it easier to predict.

We define $\hat{Q}$ to be our reparametrization. $\hat{Q}(s, a)$ should be roughly in range $[0, 1]$, and it should correspond to how close we are to $V^*(s)$. Q could be decomposed multiplicatively as $Q(s, a) = \hat{Q}(s, a)V^*(s)$. However, in practice, we do not have access to $V^*(s)$, thus instead we use an estimate of future rewards based on the total number of digits left in the sequence. Since every correct prediction yields a reward of 1, the optimal policy should achieve sum of future rewards equal to the number of remaining symbols to predict. The number of remaining symbols to predict is known and we denote it by $\hat{V}(s)$. Note that this is a form of supervision, albeit a weak one.

Therefore, we normalize the Q-function by the remaining sum of rewards left in the task:

$$\hat{Q}(s, a) := \frac{Q(s, a)}{\hat{V}(s)}$$

We assume that s transitions to s', and we re-write the Q-learning update equations:

$$\hat{Q}(s, a) = \frac{R(s')}{\hat{V}(s)} + \gamma \max_a \frac{\hat{V}(s')}{\hat{V}(s)} \hat{Q}(s', a)$$

$$\hat{Q}_{t+1}(s, a) = \hat{Q}_t(s, a) - \alpha \big[\hat{Q}_t(s, a) - \big(\frac{R(s')}{\hat{V}(s)} + \gamma \max_a \frac{\hat{V}(s')}{\hat{V}(s)} \hat{Q}_t(s', a) \big) \big]$$

Note that $\hat{V}(s) \geq \hat{V}(s')$, with equality if no digit was predicted at the current time-step. As the episode progresses, the discount factor $\frac{\hat{V}(s')}{\hat{V}(s)}$ decreases, making the model myopic. At the end of the sequence, the discount drops to $\frac{1}{2}$.

[§]If we were to use reinforcement to train the symbol output as well as the actions, then the environment would be stationary. However, this would mean ignoring the reliable signal available from direct backpropagation of the symbol output.

6.2. Watkins $Q(\lambda)$

The update to $Q(s, a)$ in Eqn. 2 comes from two parts: the observed reward $R(s')$ and the estimated future reward $Q(s', a)$. In our setting, there are two factors that make the former far more reliable than the latter: (i) rewards are deterministic and (ii) the non-stationarity (induced by the ongoing learning of the symbol output by backpropagation) means that estimates of $Q(s, a)$ are unreliable as environment evolves. Consequently, the single action recurrence used in Eqn. 2 can be improved upon when on-policy actions are chosen. More precisely, let $a_t, a_{t+1}, \ldots, a_{t+T}$ be consecutive actions induced by Q:

$$a_{t+i} = \operatorname*{argmax}_a Q(s_{t+i}, a)$$
$$s_{t+i} \xrightarrow{a_{t+i}} s_{t+i+1}$$

Then the optimal Q^* follows the following recursive equation:

$$Q^*(s_t, a_t) = \sum_{i=1}^{T} \gamma^{i-1} R(s_{t+i}) + \gamma^T \max_a Q^*(s_{t+n+1}, a)$$

and the update rule corresponding to Eqn. 2 becomes:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) - \alpha \big[Q_t(s_t, a_t) - \big(\sum_{i=1}^{T} \gamma^{i-1} R(s_{t+i}) + \gamma^T \max_a Q_t(s_{t+n+1}, a) \big) \big]$$

This is a special form of Watkins $Q(\lambda)$ (Watkins, 1989) where $\lambda = 1$. The classical applications of Watkins $Q(\lambda)$ suggest choosing a small λ, which trades-off estimates based on various numbers of future rewards. $\lambda = 0$ rolls back to the classical Q-learning. Due to reliability of our rewards, we found $\lambda = 1$ to be better than $\lambda < 1$, however this needs further study.

Note that this unrolling of rewards can only take place until a non-greedy action is taken. When using an ϵ-greedy policy, this means we would expect to be able to unroll ϵ^{-1} steps, on average. For the value of $\epsilon = 0.05$ used in our experiments, this corresponds to 20 steps on average.

6.3. Penalty on Q-function

After reparameterizing the Q-function to $\hat{Q}$ (Section 6.1), the optimal $\hat{Q}^*(s, a)$ should be 1 for the correct action and zero otherwise. To encourage our estimate $\hat{Q}(s, a)$ to converge to this, we introduce a penalty that "pushes down" on incorrect actions: $\kappa \| \sum_a \hat{Q}(s, a) - 1 \|^2$. This has the effect of introducing a margin between correct and incorrect actions, greatly improving generalization. We commence training with $\kappa = 0$ and make it non-zero once good accuracy is reached on short samples (introducing it from the outset hurts learning).

6.4. Reinforcement Learning Experiments

We apply our enhancements to the six tasks in a series of experiments designed to examine the contribution of each of them. Unless otherwise specified, the controller is a 1-layer GRU model with 200 units. This was selected on the basis of its mean performance across the six tasks in the supervised setting (see Section 5). As the performance of reinforcement learning methods tend to be highly stochastic, we repeat each experiment 10 times with a different random seed. Each model is trained using 3×10^7 characters which takes ~ 4 hrs. A model is considered to have successfully solved the task if it able to give a perfect answer to 50 test instances, each 100 digits in length. The GRU model is trained with a batch size of 20, a learning rate of $\alpha = 0.1$, using the same initialization as (Glorot & Bengio, 2010) but multiplied by 2. All tasks are trained with the same curriculum used in the supervised experiments (and in (Joulin & Mikolov, 2015)), whereby the sequences are initially of complexity 6 (corresponding to 2 or 3 digits, depending on the task) and once 100% accuracy is achieved, increased by 4 until the model is able to solve validation sequences of length 100. For 3-row addition, a more elaborate curriculum was needed which started with examples that did not involve a carry and contained many zero. The test distribution was unaffected.

We show results for various combinations of terms in Table 3. The experiments demonstrate that standard Q-learning fails on most of our tasks (first six columns). Each of our additions (dynamic discount, Watkins $Q(\lambda)$ and penalty term) give significant improvements. When all three are used our model is able to succeed at all tasks, providing the appropriate curriculum and controller are used. We also explored alternate reinforcement learning algorithms, in particular, REINFORCE (Williams, 1992) with the architectural modifications described in (Zaremba & Sutskever, 2015). This could solve the Copy and Reverse tasks, but was sensitive to the hyper-parameter settings.

For the reverse and walk tasks, the default GRU controller failed completely. However, using a feed-forward controller instead enabled the model to succeed, when dynamic discount and Watkins $Q(\lambda)$ was used. As noted above, the 3-row addition required a more careful curriculum before the model was able to learn successfully. Increasing the capacity of the controller (columns 2-4) hurts performance, echoing Fig. 2. The last two columns of Table 3 show results on test sequences of length 1000. Except for multiplication, the models still generalized successfully.

Fig. 5 shows accuracy as a function of test example complexity for standard Q-learning and our enhanced version. The difference in performance is clear. At very high complexity, corresponding to 1000's of digits, the accuracy starts to drop on the more complicated tasks. We note that these trends are essentially the same as those observed in the supervised setting (Fig. 3), suggesting that Q-learning is not to blame. Instead, the inability of the controller to learn an automata seems to be the cause. Potential solutions to this might include (i) noise injection, (ii) discretization of state, (iii) a state error correction mechanism or (iv) regularizing the learned automata using MDL principles. However, this issue, the inability of RNN to perfectly represent an automata can be examined separately from the setting where actions have to be learnt (i.e. in the supervised domain).

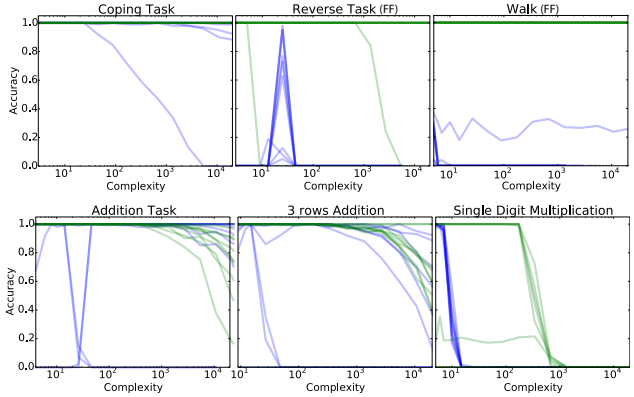

Figure 5. Test accuracy as a function of task complexity (10 runs) for standard Q-learning (blue) and our enhanced version (dynamic discount, Watkins $Q(\lambda)$ and penalty term). Accuracy corresponds to the fraction of correct test cases (all digits must be predicted correctly for the instance to be considered correct).

6.5. Multiple Solutions

On examination of the models learned on the addition task, we notice that three different solutions were discovered. While they all give the correct answer, they differ in their actions over the input grid, as shown in Fig. 6.

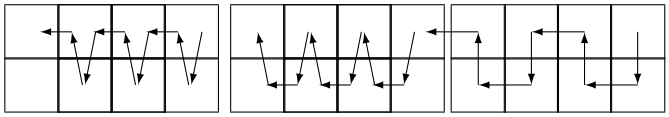

Figure 6. Our model found three different solutions to the addition task, all of which give the correct answer. The arrows show the trajectory of the read head over the input grid.

6.6. Reward Frequency vs Reward Reliability

We explore how learning time varies as the size of the target vocabulary is varied. This trades off reward frequency and reliability. For small vocabularies, the reward occurs more often but is less reliable since the chance of the wrong action sequence yielding the correct result is relatively high (and vice-versa for for larger vocabularies). For copying and reverse tasks, altering the vocabulary size just alters the variety of symbols on the tape. However, for the arithmetic operations this involves a change of base, which influences the task in a more complex way. For instance, addition in

	Test length	100	100	100	100	100	100	100	100	1000	1000
	#Units	600	400	200	200	200	200	200	200	200	200
	Discount γ	1	1	1	0.99	0.95	D	D	D	D	D
	Watkins $Q(\lambda)$	×	×	×	×	×	×	✓	✓	✓	✓
Task	Penalty	×	×	×	×	×	×	×	✓	×	✓
Copying		30%	60%	90%	50%	70%	90%	100%	100%	100%	100%
Reverse		0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Reverse (FF controller)		0%	0%	0%	0%	0%	0%	100%	90%	100%	90%
Walk		0%	0%	0%	0%	0%	0%	10%	90%	10%	80%
Walk (FF controller)		0%	0%	0%	0%	0%	0%	100%	100%	100%	100%
2-row Addition		10%	70%	70%	70%	80%	60%	60%	100%	40%	100%
3-row Addition		0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
3-row Addition (extra curriculum)		0%	50%	80%	40%	50%	50%	80%	80%	10%	60%
Single Digit Multiplication		0%	0%	0%	0%	0%	100%	100%	100%	0%	0%

Table 3. Success rates for classical Q-learning (columns 2-5) versus our enhanced Q-learning. A GRU-based controller is used on all tasks, except reverse and walk which use a feed-forward network. Curriculum learning was also used for the 3-row addition task (see text for details). When dynamic discount (D), Watkins $Q(\lambda)$ and the penalty term are all used the model consistently succeeds on all tasks. The model still performs well on test sequences of length 1000, apart from the multiplication task. Increasing the capacity of the controller results in worse performance (columns 2-4).

base 4 requires the memorization of digit-to-digit addition table of size 16 instead of 100 for the base 10. Table 4 shows the median training time as a function of vocabulary size. The results suggest that an infrequent but reliable reward is preferred to a frequent but noisy one.

	Vocabulary Size								
Task	2	3	4	5	6	7	8	9	10
Copying	1.4	0.6	0.6	0.6	0.6	0.5	0.6	0.6	0.6
Reverse (FF)	6.5	23.8	3.1	3.6	3.8	2.5	2.8	2.0	3.1
Walk (FF)	8.7	6.9	6.8	4.0	6.2	5.3	4.4	3.9	11.1
Addition	250.0	30.9	14.5	26.1	21.8	21.9	25.0	23.4	21.1
3-number +	250.0	61.5	250.0	250.0	112.2	178.2	93.8	79.1	81.9
Single Digit ×	-	6.2	17.8	20.9	21.4	21.5	22.3	23.3	24.7

Table 4. Median training time (mins) over 10 runs as we vary the base used (hence vocabulary size). Training stops when the model successfully generalizes to test sequences of length 100. The results show the relative importance of reward frequency versus reliability (the latter being more important).

6.7. Reward Structure

Reward in reinforcement learning systems drives the learning process. In our setting we control the rewards, deciding when, and how much to give. We now examine various kinds of rewards and their influence on the learning time of our system. Our vanilla setting gives a reward of 1 for every correct prediction, and reward 0 for every incorrect one. We refer to this setting as "0/1 reward". We consider two other settings in addition to this, both of which rely on the probabilities of the correct prediction. Let y be the target symbol and $p_i = p(y = i), i \in [0, 9]$ be the probability of predicting label i. In setting "Discretized reward", we sort p_i. That gives us an order on indices $a_1, a_2, \ldots, a_{10}$, i.e. $p_{a_1} \geq p_{a_2} \geq p_{a_3} \cdots \geq p_{a_{10}}$. "Discretized reward" yields reward 1 iff $a_1 \equiv y$, reward $\frac{1}{2}$ iff $a_2 \equiv y$, and reward $\frac{1}{3}$ iff $a_3 \equiv y$. Otherwise, environment gives a reward 0. In the "Continuous reward" setting, a reward of p_y is given for every prediction. One could also consider reward $\log(p_y)$, however this quantity is unbounded, and further processing might be necessary to make it work. Table 5 gives results for the three different reward structures, showing training time for the five tasks (training is stopped once the model generalizes to test sequences of length 100). One might expect that a continuous reward would convey more information than a discrete one, thus result in faster training. However, the results do not support this hypothesis, as training seems harder with continuous reward than a discrete one. We hypothesize, that the continuous reward makes environment less stationary, which might make Q-learning less efficient, although this needs further verification.

Reward Type / Task	0/1 reward	Discretized	Continuous
Copying	0.6	0.6	0.8
Reverse (FF controller)	3.1	3.1	59.7
Walk (FF controller)	11.1	9.5	250.0
Addition	21.1	21.6	24.2
3-number Addition (extra curriculum)	81.9	77.9	131.9
Single Digit Multiplication	24.7	26.5	26.6

Table 5. Median training time (minutes) for the five tasks for the three different reward structures. "0/1 reward": the model gets a reward of 1 for every correct prediction, and 0 otherwise. "Discretized" reward provides a few more values of reward prediction, if sufficiently close to the correct one. "Continuous" reward gives a probability of correct answer as the reward. See text for details.

7. Discussion

We have explored the ability of neural network models to learn algorithms for simple arithmetic operations. Through experiments with supervision and reinforcement learning, we have shown that they are able to do this successfully, albeit with caveats. Q-learning was shown to work as well as the supervised case. But, disappointingly, we were not able to find a single controller that could solve all tasks. We found that for some tasks, generalization ability was sensitive to the memory capacity of the controller: too little and it would be unable to solve more complex tasks that rely on carrying state across time; too much and the resulting model would overfit the length of the training sequences. Finding automatic methods to control model capacity would seem to be important in developing robust models for this type of learning problem.

References

Cho, Kyunghyun, van Merrienboer, Bart, Gulcehre, Caglar, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Das, Sreerupa, Giles, C Lee, and Sun, Guo-Zheng. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *In Proceedings of The Fourteenth Annual Conference of Cognitive Science Society*, 1992.

Gers, Felix A, Schraudolph, Nicol N, and Schmidhuber, Jürgen. Learning precise timing with lstm recurrent networks. *The Journal of Machine Learning Research*, 3: 115–143, 2003.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Goldberg, David E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675.

Gomez, Faustino, Schmidhuber, Jürgen, and Miikkulainen, Risto. Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965, 2008.

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

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. *arXiv preprint arXiv:1506.02516*, 2015.

Hasselt, Hado V. Double Q-learning. In *Advances in Neural Information Processing Systems*, pp. 2613–2621, 2010.

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

Holland, John H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. *arXiv preprint arXiv:1503.01007*, 2015.

Liang, Percy, Jordan, Michael I, and Klein, Dan. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446, 2013.

Loch, John and Singh, Satinder P. Using eligibility traces to find the best memoryless policy in partially observable markov decision processes. In *ICML*, pp. 323–331, 1998.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Nordin, Peter. *Evolutionary program induction of binary machine code and its applications*. Krehl Munster, 1997.

Schmidhuber, Jürgen. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

Solomonoff, Ray J. A formal theory of inductive inference. Part I. *Information and control*, 7(1):1–22, 1964.

Sukhbaatar, Sainbayar, Szlam, Arthur, Weston, Jason, and Fergus, Rob. Weakly supervised memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

Sutton, Richard S and Barto, Andrew G. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Watkins, Chris. *Learning from Delayed Rewards*. PhD thesis, Cambrdige University, 1989.

Williams, Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229256, 1992.

Williams, Ronald J and Zipser, David. Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1):87–111, 1989.

Wineberg, Mark and Oppacher, Franz. A representation scheme to perform program induction in a canonical genetic algorithm. In *Parallel Problem Solving from NaturePPSN III*, pp. 291–301. Springer, 1994.

Zaremba, Wojciech and Sutskever, Ilya. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015.