Learning to Count by Think Aloud Imitation

Laurent Orseau
INSA/IRISA, Rennes, France
lorseau@irisa.fr

Abstract

Although necessary, learning to discover new solutions is often long and difficult, even for supposedly simple tasks such as counting. On the other hand, learning by imitation provides a simple way to acquire knowledge by watching other agents do. In order to learn more complex tasks by imitation than mere sequences of actions, a Think Aloud protocol is introduced, with a new neuro-symbolic network. The latter uses time in the same way as in a Time Delay Neural Network, and is added basic first order logic capacities. Tested on a benchmark counting task, learning is very fast, generalization is accurate, whereas there is no initial bias toward counting.

1 Introduction

Learning to count is a very difficult task. It is well-known that children do not learn only by themselves: their parents and their teachers provide an important active help.

On the other hand, in the field of on-line machine learning where the teacher does not provide help, some Recurrent Neural Networks are interestingly successful, such as the Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997]. However, to learn some counters the network still requires to be trained through tens of thousands of sequences, and it already contains a counting feature, although it had not been specifically designed for this.

Between active teaching and automatic discovery stands learning by imitation [Schaal, 1999]: the learner watches the "teacher" solve the task, but the teacher may not be aware of the presence of the learner. This is helpful for sequences of actions, for example to make a robot acquire a complex motor behavior [Schaal, 1999; Calinon *et al.*, 2005]: the learner can see what intermediate actions should be done.

Fewer studies on imitation focus on learning sequences that are more complex than sequences of physical actions, i.e., algorithms. Programming by Demonstration [Cypher, 1993] deals with this issue, but is more interested in human-machine graphical interaction and thus makes many assumptions about how tasks are represented. Maybe the work that is the closest to the one presented in this paper is that of Furse [Furse, 2001]. He uses a paper-and-pencil approach to learn, by imitation, algorithms as complex as making a division. However, still many assumptions are made that clearly ease learning: the paper-and-pencil approach provides an external infinite memory and useful recall mechanisms; the system also already contains some basic arithmetic tools, such as the notion of number, of numerical superiority, etc. Furthermore, it is able to learn to make function calls (access to subroutines), but the teacher has to explicitly specify which function to use, with which inputs and outputs. Calling a function is thus a special action.

In this paper, there is no specific action: the sequences are composed of completely *a priori* meaningless symbols, i.e., all these symbols could be interchanged without changing the task. It is able to make basic automatic function calls, without explicitly specifying it. There is also no initial internal feature that biases the system toward counting.

In section 2, Think Aloud Imitation Learning is described, with a usual neural network. The latter is superseded in the next section by the Presence Network, a growing neuro-symbolic network [Orseau, 2005a]. In section 4, the latter is then augmented by special connections, and its development is explained on a simple task. Then experiments such as counting are given, showing that learning can be faster by several orders of magnitude than with a usual neural network, and that some important learning issues can be circumvented, so that learning complex tasks on-line in a few examples is possible.

2 Think Aloud Imitation Learning

In imitation, the environment, which gives the task to solve, produces events $env(T)$ and the teacher, who knows how to solve it, produces $tch(T)$, where T denotes the present time. During learning, the agent tries to predict the teacher's next action. When it succeeds $a(T) = tch(T)$.

The agent is given sequences of events such as /abcdEfgH/, where the events provided by the environment are written in lowercase letters and those provided by the teacher are in capitals, but should be seen as the same symbols. The 26 letters of the alphabet are used as events all along this paper. For better legibility only, spaces will sometimes be introduced in the sequences. The goal of the learner is to predict the teacher's events, so that once it is on its own, it not only predicts but in fact acts like the teacher would do.

In imitation, one problem with Recurrent Neural Networks (for example) is that they have internal states; the learner cannot "see" these states inside the teacher, which are thus difficult to learn.

Therefore, in the Think Aloud paradigm, the teacher is forced to "think aloud", this means that it does not have internal states but it can "hear" what it itself says: the recurrence is external, not internal. Then it can be easily imitated by the learner, who "listens" to the teacher when learning. Once on its own, the agent, not learning anymore, also thinks aloud and can hear itself. This allows the teacher to make intermediate computation steps that can be imitated by the learner. During learning, the agent thus receives the sequence of events of the teacher and of the environment $e(T) = env(T) + tch(T)$. After learning, during testing, the agent is on its own, so $tch(T) = \phi$ and the agent listens to itself, not to the teacher: $e(t) = env(T) + a(T)$. This can be seen as some teacher forcing technique [Williams and Zipser, 1989].

A first, simplified view of the architecture is now given. In order to take time into account without having internal states, we use a sort of Time Delay Neural Network (TDNN) [Sejnowski and Rosenberg, 1988], that computes its outputs given only the last τ time steps. At present time T, it computes: $Net(T) = f(e(T), e(T-1), \dots e(T-\tau+1))$.

In the sequel, the indices i, j, k are used respectively only for inputs, hidden neurons, output neurons. x_i is the activation of input i. As with other variables, writing y_i is a shortcut for $y_i(T)$, where T is the present time, with the only exception that x_i^d stands for $x_i(T-d)$. Writing $e(t) = i$ means that input i is activated at time t, where inputs are encoded in a 1-of-N way.

Usually, the TDNN architecture is explained with delayed copy sets of inputs; here we describe it as follows: each connection (weight) w_{ji}^d from input unit i to hidden unit j has a delay d, so that if $e(t) = i$, w_{ji}^d transmits this activation to unit j at time $t + d$. A hidden neuron j is then connected to input i through a number τ of connections w_{ji}^d, each one having a different delay $\tau < d \leq 0$. Such a network cannot produce recurrent, infinite behaviors.

It is the fact that the agent can hear what it says during testing that gives it recurrent capacities: since actions are reinjected on inputs, the resulting recurrent network is quite similar to a NARX neural network [Lin *et al.*, 1996], without learning considerations through the recurrence (no backpropagation through time). For example, if the agent is given sequences like /abABABAB. . . /, it then learns to say a after any /ab/, and b after any /ba/. Then, once it hears /ab/, it says a. It has therefore heard /aba/, and then says b, has heard /abab/, says a, and so on. It then produces an infinite behavior, but learning occurs only in the TDNN.

Instead of direct supervision of outputs, we use instant reinforcements [Kaelbling *et al.*, 1996] on one single output. This is more general for an autonomous agent like a robot and may be used not only for imitation. While imitating, acting like the teacher is innerly rewarding. Actions are selected among the inputs. Thus, at each time step, to predict the next action/input, each input is tested and the one that predicts the best reinforcement for the next time step is chosen.

3 The Presence Network

In a TDNN, the number of input connections of a single hidden neuron j is $|w_j| = \tau N_I$, where N_I is the number of inputs. It grows quickly when τ or N_I grows and makes learning difficult: the number of training examples is often said to be in $|w|^3$. In the experiments described in section 5, many time steps, neurons and inputs are needed, so that using a mere TDNN is unpractical. TDNN and backpropagation also have important limitations when learning several tasks [Robins and Frean, 1998]. They are prone to catastrophic forgetting: learning a concept B after a concept A often modifies too much the knowledge of concept A. Another issue arises with unmodified weights: the network will tend to provide answers even before learning. Thus, even if weights are frozen after concept A, adding new neurons may (sometimes not) temporarily mask other already acquired knowledge. Learning must then be adapted to prevent this.

Instead of a mere TDNN, we use a Presence Network [Orseau, 2005a], which is a neuro-symbolic network more tailored for learning sequences of events such as those dealt with here. It uses time in the same way as a TDNN and its main properties are very fast learning, local connectivity and automatic creation of neurons. A slight modification is made: "specificity" is introduced to circumvent an issue of the initial network. The Presence Network is described below but for its use in a less specific context, see [Orseau, 2005a].

The heuristic idea is that new hidden neurons are connected only to inputs that were active in the short past (that are in the short-term τ-memory), so that the number of connections does not fully depend on the number of inputs. This is particularly useful when one event type is coded on a single input. Since sequences are composed of mutually exclusive events, there is only one event per time step, so a neuron would have only one connection per time step in such a network. Initially there is no hidden neuron, and new ones are added on-line when errors occur.

First, the hidden neuron and its optimization rule is given, in order to show what the network can represent. The action selection scheme can then be given to show how to circumvent one issue of the representation capacities of the network. Then the learning protocol follows.

3.1 Hidden Neuron

If, on a sequence e, a new neuron must be created (when specified in section 3.4), the new neuron j is connected to the inputs that were active in the short past with a corresponding delay d. Its set w_j of input connections is then:

$$w_j = \{w_{ji}^d \mid \forall t, T-\tau < t \leq T,\ \exists! w_{ji}^d,\ i = e(t),\ d = T-t\}.$$

The sequence used to create j is called its sequence of reference, r_j.

Let $\theta_j = 1/|w_j|$, which is not modifiable, and then set the initial weight value $\forall w_{ji}^d \in w_j, w_{ji}^d = \theta_j$, which ensures that $\sum_{i,d} w_{ji}^d = 1$. The output weight w_{kj} toward the reinforcement output is set to 1.

Let σ_j be the weighted sum $\sigma_j = \sum w_{ji}^d x_i^d$. The activation

function y_j is:

$$y_j = \begin{cases} 0 & \text{if } \sigma_j \leq 1-\theta_j \quad \text{(inactive neuron)} \\ \frac{\sigma_j-(1-\theta_j)}{1-(1-\theta_j)} & \text{if } 1-\theta_j < \sigma_j < 1 \quad \text{(active neuron)} \\ 1 & \text{if } \sigma_j \geq 1 \quad \text{(active neuron)} \end{cases}$$

Note that $1-\theta_j$ is the activation threshold. Thus, if r_j is presented again to the agent, j recognizes it and $y_j = 1$. Before optimization, only r_j can activate j. If at least one event is substituted for another one, $y_j = 0$.

3.2 Optimization of a Hidden Neuron

To optimize the neuron j, each w can be seen as either informative (close to θ) or not (close to 0). There is no negative weight. If the neuron should be more activated, "quanta" of weights are moved from inactive connections (that transmitted no activation) to active ones. The converse is done if the neuron should not be active.

Algorithm 1: Update rule for input weights of a hidden neuron j. α is the learning rate. Er is the error received from Algorithm 3.

$\delta = \alpha.\theta_j.Er$ // the maximum modification of w
$\forall w_{ji}^d \in w_j$:
if ($x_i^d > 0$ *and* $Er > 0$) *or* ($x_i^d = 0$ *and* $Er < 0$) **then**
  $\Delta_i^d = \min(\delta, 1 - w_{ji}^d)$
else $\Delta_i^d = -\min(\delta, w_{ji}^d)$
$S^+ = \sum_{i \in \{i \mid x_i^d > 0\}} |\Delta_i^d|$
$S^- = \sum_{i \in \{i \mid x_i^d = 0\}} |\Delta_i^d|$
$S = min(S^+, S^-)$
$\forall w_{ji}^d \in w_j$: **if** $S \neq 0$ **then**
  if $x_i^d > 0$ **then** $S_i^d = S^+$ **else** $S_i^d = S^-$
  $w_{ji}^d \leftarrow w_{ji}^d + \Delta_i^d . \frac{S}{S_i^d}$

This learning algorithm ensures $\sum_{i,d} w_{ji}^d = 1$, i.e., the sequence or reference r_j can always be recognized by j. All weights also stay in [0, 1]: since corresponding events were present when the neuron was created, it would not be logical that the weights become inhibitory. A neuron is thus a filter that is active when it recognizes a conjunction of events appearing at different time steps, which describes a potentially general sequence of events, and where weights of non-informative time steps are pushed toward zero.

The output weight of a chosen neuron j is updated by a simple delta rule and is bounded in [0, 1]: $w_{kj} \leftarrow w_{kj} + \eta Er$, where η is a learning rate, but its value is not critical. In fact, the output weight has not much influence on the concept embodied by the hidden neuron and may be used only to "discard" a neuron/concept if statistically found wrong.

For example, a single neuron can represent the sequence /abcZ/, meaning that if the agent receives the sequence /abc/, this neuron proposes to answer z. After some examples such as /adcZ/, /aecZ/, this neuron can get generalized to /a*cZ/, where * means any possible symbol.

3.3 Action Selection

The action selection scheme is basically the same as in section 2. In *passive* mode, a neuron is a filter that recognizes a sequence and predicts a reinforcement value. In *active* mode, the agent predicts an action (so that at the next time step the sequence is recognized).

Since neurons only consider events that are present, i.e., active inputs, the main drawback of the Presence Network is that inactive inputs are not taken into account, even if this inactivity is important. Said in another way, the network cannot distinguish exception cases from general cases. For example, consider the sequences /abcZ/, /adcZ/, /aecZ/, etc. Suppose now that there is also the exception case /aycX/. If one neuron embodies this case, then after /ayc/, it naturally predicts X . But at the same time, the neuron representing the general case /a*cZ/ predicts Z. Then how to select the right action? The neuron /a*cZ/ should then have an inhibitory link with input y, but to preserve the interesting properties of the Presence Network, i.e., not to connect inactive inputs (in potentially great number) to a new neuron, precedence in action selection is automatically given to cases that are the most specific: more specific neurons take more events into account, and thus "explain" more accurately the given sequence.

First, the usefulness of a connection is defined: $U(w_{ji}^d) = \min(1, w_{ji}^d/\theta_j)$. Specificity ψ_j means how well the neuron j explains the current sequence: $\psi_j = \sum_d x_i^d . U(w_{ji}^d)$. Then it is preferred to select the most specific neuron, among the activated ones (see Algorithm 2).

The specificity of the neuron that recognizes /a*cZ/ is $\psi = 3$, because the middle weight has become 0; and the specificity of the neuron that recognizes /aycX/, which has not been optimized, is $\psi = 4$. This means that after the sequence /ayc/, the agent will always choose X preferentially. A neuron representing an exception case similar to a general case has more non-zero weights, which ensures it to be selected in priority.

A interesting property is that the general case needs no modification at all when learning one, or more specific cases.

Algorithm 2: Action selection.

// Find event that activates each j most at next time step:
$\forall j, e_j = argmax_{e(T+1)}\ y_j(T+1)$
Compute corresponding specificity $\psi_j(T+1)$
// Keep only active and most specialized neurons:
$A = \{j \mid y_j(T+1) > 0, \psi_j(T+1) = \max_h\ \psi_h(T+1)\}$
// Select p, the most specialized and active neuron:
$p = argmax_{j \in A}\ y_j(T+1)$
Select action $a(T+1) = e_p$ if p exists, ϕ otherwise

3.4 Learning Protocol

Catastrophic forgetting, e.g. with backpropagation, happens partly because knowledge is shared between many neurons, and many neurons can be modified at the same time. To avoid this, here at most 3 neurons can be modified at the same time. First, in order to compare two neurons on the present

sequence, when a neuron is not active ($y_j = 0$) preactivation is defined[1]: $\tilde{y}_j = \sigma_j/(1-\theta_j)$.

Let $\tilde{N}$ be the most preactivated neuron, which must have a connection with $d = 0$ pointing to $e(T)$, in order to be able to predict it:
$\tilde{N} = argmax_j \{\tilde{y}_j \mid y_j = 0, \tilde{y}_j > 0 \; and \; \exists w^d_{ji}, d = 0, i = e(T)\}$. If none exists, $\tilde{N} = \phi$.

With an accurate selection mechanism of the nearest neuron, a high learning rate can be used. Of course, the nearest neuron may not always be the correct one to optimize, but it then will probably be optimized again in the other way.

Algorithm 3 describes the learning protocol, to decide when neurons must be created or modified. A neuron can be added at the same time another one is modified, because the latter may be wrongly optimized. If a neuron is already active, it means it has recognized the sequence, can thus predict the right answer, and no neuron needs to be added.

Algorithm 3: Learning protocol.

Time T: Predict teacher action: $a(T+1)$
$T \leftarrow T+1$
if $a(T) = e(T)$ **then** // correct action selected
  Optimize selected neuron p with error $Er = (1 - w_{kp}.y_p)$
else
  Create new neuron on current specific case
  if $\exists h, y_h > 0$, $w_{kh}.y_h = max_j \; w_{kj}.y_j$ **then**
    // h should have been selected, not p
    Optimize p if exists, $Er = (0 - w_{kp}.y_p)$
    Optimize h, $Er = (1 - w_{kh}.y_h)$
  else
    Optimize $\tilde{N}$ if exists, $Er = 1$

4 Time Delayed Identity Connections

Time Delayed Identity Connections (TDIC) allows to know when there has been a repetition of an event in the short term. In [Orseau, 2005b], they are presented in the form of short term memory special inputs. Here, they are dynamic connections, but it is the same. They enhance the network generalization capacity, and allow both to compare two events in passive mode, and to repeat a past event in active mode.

When creating a neuron j at present time T with a sequence of events e, if two events $e(t_1)$ and $e(t_2)$ are identical such that $\tau > T - t_1 > T - t_2$, a connection w^{dD}_{ji} is added (before setting θ_j) to the neuron with the following properties. In addition to the normal delay $d = T - t_2$, it has another one $D = T - t_1$, pointing to the event to compare with. This connection is dynamic: at each time step T, the target input i of w^{dD}_{ji} is changed[2] such that $i = e(T - D)$. Then the connection behaves exactly as a normal one of delay d. Note that the special case $D = d$ is a normal connection.

[1] $(1-\theta_j) = 0$ never happens: the neuron has a single connection and then either $y_j = 1$ or $i \neq e(T)$.

[2] If $e(T - D)$ does not exists, i can be an idle input unit.

Now since a single event can be referred to by several connections, the definition of specificity must be modified so that at most one event by delay d is taken into account. For example, on /abc/, initially $\psi = 3$. On /aba/, there are 3 normal connections plus one TDIC, thus $\psi = 4$, but should logically be 3. Thus: $\psi_j = \sum_d max_i \; (x^d_i.U(w^d_{ji}))$.

The resulting system can represent almost Prolog logic programs [Muggleton, 1999] without lists, but in an on-line way, and with no explicit function name and parameter. The TDICs bind variables (through time) and allow comparison between variables (here variables are events, or time steps), a typical first order logic capacity.

4.1 A Simple Task: Equality

To give an example of how the network works, its development is shown when trained on a simple equality task, where the agent must answer True or False whether two given events are identical or not. It is a static task in essence. Examples of sequences: /ab F/, /dd T/, /ph F/, /cc T/, /vv T/, etc.

The network has 26 inputs and $\tau = 3$ (or higher). Initially there is no hidden neuron.

The first sequence /ab F/ is provided to the agent. During the 2 first time steps, nothing happens, except that the network puts these symbols in its τ-memory. On symbol F, the network should have predicted F, but did not, because there is no neuron. Therefore, a new neuron N_1 is created, in order to make it correlate F with /ab/. It is connected to input a with a delay $d = 2$, to b with a delay $d = 1$, and to f with $d = 0$. Input weights are set to $1/|w_{N_1}| = 1/3$. If the sequence /ab/ is presented again to the agent, the network predicts F, then the teacher says F, N_1 is activated, expecting a reward, effectively receives a reward, and no modification is needed.

The sequence /dd T/ is then provided to the agent. Once again, there is no active or preactive neuron that could predict T, so a new neuron N_2 is added. N_2 has 4 connections: 3 are similar to the previous case, and 1 is a TDIC. The latter is created because of the repetition of d, with delays $d = 1$ and $D = 2$. N_2 can now answer correctly for /dd T/. Note that the TDIC is not necessary to learn this case by heart; it is useful only if N_2 is optimized.

The sequence /ph F/ is presented. N_2's TDIC is now pointing to p, but does not transmit an activation. Again, no neuron predicted F. So a new neuron N_3 is added to answer correctly to this sequence. But now N_1 is preactivated, so it is chosen to be slightly modified to predict F. The weight of its active connections (those that transmitted an activation) are increased, whereas the other input weights are decreased. Thus, input connections from a and b of N_1 are considered as noise.

On the sequence /cc/, N_1 predicts F because it would slightly activate N_1, although there is a repetition. But the teacher says T instead. No neuron is activated, so a new neuron $N4$ is added, and N_1 is not modified. And N_2, being the most preactivated neuron, is modified: in particular, the weight of the TDIC is increased because it is pointing to c.

On the sequence /vv/, N_1 predicts F and N_2 predicts T. Since $\psi_{N_1} = 1$ (approximately) and $\psi_{N_2} = 2$, N_2 explains more specifically this sequence and thus the network predicts T. This is correct, so no neuron is added, and N_2 is optimized again to better predict T (with a higher value).

Now the network predicts correctly any sequence like /$\gamma\gamma$ T/, recognized by N_2, and /$\gamma\beta$ F/, recognized by N_1, where γ and β are any two letters.

5 Experiments

The present work is close to the continual learning framework [Ring, 1997] where the agent must re-use previously acquired knowledge in order to perform better on future tasks. Here, we are interested in tasks where the re-use of knowledge is not only useful, but *necessary* to generalize, and is called as a function. Thus, even for static tasks, using a temporal framework can give access to these generalization capacities.

In order to show the usefulness of context and how the agent can make automatic function calls, in section 5.1 we use the task described in [Orseau, 2005b], where it is tested with a TDNN with backpropagation and "TDIC" inputs.

The previous tasks can be seen as static tasks. In section 5.2 is given a typical dynamic one: in the domain of languages, a sequence of $x^n y^n$ is a sequence of a finite number of x followed by a sequence of the same amount of y.

Inputs are the 26 letters of the alphabet, $\alpha = 1$ and $\eta = 0.5$. τ is set to 20, but shorter sequences may create less connections with the Presence Network (τ should be set to a sufficiently high value so as to take enough events into account). Learning stops after a given number of correctly predicted sequences. This number is not taken into account in the table. Since learning in the Presence Network is deterministic, the order of the training sequences is randomized. During testing, the agent is on its own, so its predicted actions take the place of the teacher's events. 10 successive trials (learning and testing) are done for each task (see results in Table 1).

Table 1: Average results for 10 successive trials. Tr (Ts) is the number of training (testing) sequences; ETr (ETs) is the number of sequences on which the agent made errors during training (testing); #N is the number of neurons created only for the task being learned; #W is the total number of weights for the #N neurons; the * is for tasks trained with a normal TDNN.

Task	Tr	ETr	Ts	ETs	#N	#W
gp*	41 600	*N.A.*	100	0	8	968
gp	97.5	27.3	100	0	27.3	139.8
is gp*	9 300	*N.A.*	100	2	5	605
is gp	15.8	7.6	100	0	14.9	140.5
flw	219.3	81.9	675	0	92.7	887
$x^n y^n$	13.2	6.4	10	0	114.2	3082.5

The system could hardly be directly compared with other methods. The closest one is [Furse, 2001], which has yet many important differences (section 1). There are also some similarities with Inductive Logic Programming [Muggleton, 1999], but the aims are different. ILP is not on-line, has explicit function names and uses global search mechanisms.

5.1 Groups

The alphabet is decomposed into 6 groups of consecutive letters: a to e are in group a, f to j are in group b, ... and z is in group f. The agent must first learn by heart in which group is each letter. Then, given a letter and a group, it must learn and generalize to answer yes if the letter is in the group, and no otherwise.

This task nicely shows how intermediate computations to re-use background knowledge is necessary to generalize.

On the first task, the agent is given the 26 sequences of the type /gpaA/, /gpzF/, etc. Here, gp is the name of the task, the context. Learning stops when the whole training set is correctly processed. After learning, neurons are frozen to facilitate learning on the second task.

On the second task, the agent is given sequences like /is fb GPFB Y/ ("is f in the group b? The group of f is b, so yes"), /is bc GPBA N/, etc. The context gp is used to call the appropriate function. Also, without this context in the first task, there would be conflicts between the two tasks: neurons would be too general and would give answers most of the time when not needed. The context also allows to choose the nearest neuron more accurately. After a sequence /is bc/, the agent makes intermediate computations by saying GPB. Thus, since it has now heard GPB, it is auto-stimulated to answer A, given its knowledge of the first task. The TDICs allows to repeat the letter b given by the teacher to put it automatically after GP. The agent has learned to call the function gp. But there is no specific symbol to tell what is the function or the parameter. The function call is only a consequence of how knowledge is represented and used. Then the TDICs are used to compare the group given by the teacher and the one the agent knows.

During training, only the 10 first letters and the 4 first group letters are used. Learning stops after 20 successive correctly processed sequences. With the TDNN used in [Orseau, 2005b] it is necessary to have 5 more letters, generalization still induces some errors, and more importantly it is necessary to force the agent not to say anything during 4 time steps after the end of the sequences of the first task in order to prevent these neurons from disturbing learning on the second task. None of these problems arise with the Presence Network.

Furthermore, the results (see Table 1) show that learning is much faster, which is important for on-line learning, generalization is perfect to any letter and any group.

5.2 Counting

The agent has no special internal feature that can generalize a counter by itself, so it must acquire knowledge about numbers beforehand. First, the agent needs to learn to increment any 2-digit number, and then uses this incrementation as a function to solve the task $x^n y^n$.

Incrementation

In this task, given a 2-digit number the agent must answer its successor: which number follows aa? ab. Examples of sequences: /flw aa AB/, /flw ab AC/, .../flw az BA/, /flw ba BB/, .../flw zy ZZ/.

There are 25 general cases to learn (/flw γa γb/, /flw γb γc/, .../flw γy γz/), and 25 specific cases (/flw az ba/, /flw bz ca/, .../flw yz za/), for a total of 675 cases.

The training set contains all the 25 specific cases and the 150 first cases, in which sequences are then chosen randomly

in order not to have only small $a\gamma$ numbers. Training stops after 170 correctly classified sequences. After learning, the network has always generalized to the 675 cases. Finally, weights are frozen.

Here, a minimal TDNN would have 26 inputs, τ =7, 7 "TDIC" inputs, and probably at least 50 neurons would be necessary. This would make a total of more than 10 000 weights. For this task and moreover for the next one, training the TDNN does not seem reasonable.

Recurrence: 2-digit $x^n y^n$

The incrementation can now be used as a function inside a recurrent task. The outline is to initialize the counter to 0 at the beginning of the sequence, and then to call the incrementation function after each x and each y, and when the two numbers match, stop. One difficulty is that during the whole sequence of y, the number of x must be memorized. Since the teacher does not have internal states, it must use its τ-memory to keep track of it. One way to do that is to repeat it aloud at constant intervals.

Example of a sequence: /start stpx FLW AA AB stpx FLW AB AC ... FLW DF DG stpy DG FLW AA AB stpy DG FLW AB AC ... stpy DG FLW DF DG STOP/, where the number DG of x is repeated after each stpy. The context stp (for "step") is added to x and y in order not to be confused with anything else, since x and y appear also elsewhere.

After each stpx, the agent learns to say FLW, then to repeat (with TDICs) the last number, and this automatically stimulates itself to call the incrementation function and say the following number. TDICs are also used to compare the numbers and end the sequence if they match: the activated TDIC provides a higher specificity than the general case. Such sequences can have more than 14 000 events.

During training, the agent learns to use its TDICs for recurrence instead of learning only specific cases by heart. The training set contains the first 130 values of n. After 20 correctly classified sequences, learning stops. It is then tested on 10 randomly chosen sequences with $n \leq 675$.

Again, there is no single error (see Table 1), the network has generalized very quickly to unseen sequence lengths. This is far away from the tens of thousands of iterations needed by the LSTM [Hochreiter and Schmidhuber, 1997] to learn $a^n b^n$, although the learning paradigms are completely different.

Preliminary results also show that on $x^n y^n$, about 90 neurons can be automatically deleted afterwards, often with no loss in generalization accuracy.

6 Conclusion

Within Think Aloud Imitation Learning, an agent can learn on-line and accurately generalize complex sequences such as counters in very few examples shown by the teacher. The Presence Network can take long delays and many events into account and its size grows automatically. For such tasks, compared to a usual Time Delay Neural Network, learning is faster by several orders of magnitude and no problem occurs when re-using knowledge during learning. Time Delayed Identity Connections allow to make automatic function calls and provide some simple first-order logic capacities.

Current work focuses on the re-usability of knowledge so that called functions can be of unbounded size to hierarchically learn such complex sequences.

References

[Calinon *et al.*, 2005] S. Calinon, F. Guenter, and A. Billard. Goal-directed imitation in a humanoid robot. In *Proceedings of the IEEE Intl Conference on Robotics and Automation (ICRA)*, Barcelona, Spain, April 18-22 2005.

[Cypher, 1993] A. Cypher. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[Furse, 2001] E. Furse. A model of imitation learning of algorithms from worked examples. *Cybernetic Systems, Special issue on Imitation in Natural and Artificial Systems*, 32(1–2):121–154, 2001.

[Hochreiter and Schmidhuber, 1997] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[Kaelbling *et al.*, 1996] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Lin *et al.*, 1996] T. Lin, B. G. Horne, P. Tiño, and C. L. Giles. Learning long-term dependencies is not as difficult with NARX networks. In *Advances in Neural Information Processing Systems*, volume 8, pages 577–583. The MIT Press, 1996.

[Muggleton, 1999] S. Muggleton. Inductive Logic Programming. In *The MIT Encyclopedia of the Cognitive Sciences (MITECS)*. MIT Press, 1999.

[Orseau, 2005a] L. Orseau. The principle of presence: A heuristic for growing knowledge structured neural networks. In *Proceedings of the Neuro-Symbolic Workshop at IJCAI (NeSy'05)*, Edinburgh, August 2005.

[Orseau, 2005b] L. Orseau. Short-term memories and forcing the re-use of knowledge for generalization. In *ANN: Formal Models and Their Applications - ICANN 2005*, LNCS 3697, pages 39–44. Springer-Verlag, 2005.

[Ring, 1997] M. Ring. CHILD: A first step towards continual learning. *Machine Learning*, 28(1):77–104, 1997.

[Robins and Frean, 1998] A. V. Robins and M. R. Frean. Local learning algorithms for sequential learning tasks in neural networks. *Journal of Advanced Computational Intelligence*, 2(6):107–111, 1998.

[Schaal, 1999] S. Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242, 1999.

[Sejnowski and Rosenberg, 1988] T. J. Sejnowski and C. R. Rosenberg. NETtalk: a parallel network that learns to read aloud. In *Neurocomputing: foundations of research*, pages 661–672. MIT Press, 1988.

[Williams and Zipser, 1989] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.