

LEARNING TO GENERATE SUBGOALS FOR ACTION SEQUENCES

(To appear in O. Simula, editor, Proceedings of the International Conference on Artificial Neural Networks ICANN'91. Elsevier Science Publishers B. V., 1991. Submitted in January 1991.)

Jürgen Schmidhuber

Institut für Informatik
Technische Universität München
München, Germany

Abstract: None of the existing learning algorithms for neural networks in time-varying environments addresses the problem of learning ‘to divide and conquer’. It is argued that algorithms based on pure gradient descent or on adaptive critic methods are not suitable for dynamic control problems with *long* time lags between actions and consequences, and that there is a need for algorithms that perform ‘compositional learning’. This paper first describes a feed-forward system which solves at least one problem associated with compositional learning. The system *learns* to generate sub-goals. This is done with the help of ‘time-bridging’ adaptive models that predict the effects of the system’s sub-programs. In addition, a recurrent adaptive sub-goal generator for generating sequences of sub-goals is described. An experiment (obstacle avoidance in a two-dimensional environment) illustrates the approach.

1 Motivation

Most algorithms for reinforcement learning and adaptive control in non-stationary environments can be classified into two major categories.

First, there is the approach of ‘back-propagation through time and through a frozen model network’. With different degrees of generality it has been pursued by [3], [5], [4], [7], and [13] in the case where there is external feedback through a reactive environment.

Second, there is the ‘adaptive critic’ approach, which, again with different degrees of generality, has been pursued by [6], [14], [2], [1], [9], and [7]. The ‘Neural Bucket Brigade Algorithm’ [8] also bears relationships to adaptive critics.

Both the algorithms based on pure gradient descent as well as the ‘adaptive critic’ algorithms have at least one thing in common: They show significant

drawbacks when the credit assignment process has to bridge *long* time gaps between past actions and later consequences.

Both approaches show awkward performance in the case where the learning system already has learned a lot of action sequences in the past. With both approaches, credit assignment proceeds ‘from time slice to time slice’ instead of allowing ‘*jumps through time*’ on a higher, more abstract level. One could say that both approaches tend to modify ‘sub-programs’ instead of modifying the *trigger conditions* for sub-programs. They do not have an explicit concept of something like a sub-program. Pure gradient descent methods *always* consider *all* past states for credit assignment. Adaptive critics based on ‘Temporal Differences’ (reinforcement comparison methods) or on ‘Heuristic Dynamic Programming’ consider only the most recent states for ‘handing expectations back into time’. Both methods in general tend to consider the wrong states, they do not selectively focus on *relevant* points in time. This is a major reason for slow performance in large scale applications where there can be long delays between actions and consequences.

If there is no prior knowledge about typical consequences of certain action sequences, then a learning system cannot be expected to sensibly reduce the number of past states that are ‘critical’ for credit assignment. But if it is assumed that the learning system has already learned to perform well on certain sub-tasks, then an intelligent credit assignment process should make use of available sub-programs to ease the learning of new tasks.

In fact, the learning system incrementally should use information about the starting conditions and the effects of sub-programs to compose more complicated sub-programs in a hierarchical fashion.

Compositional learning [10] means finding solutions for new tasks by sequentially combining solutions to older tasks. It means learning to ‘divide and conquer’. It means dividing the task of finding a sequence of actions leading from a current state to a goal state by decomposing the problem into sub-tasks, such that already existing sub-programs for the sub-tasks can be combined. It means to ignore irrelevant details of sub-programs. It means to focus on *the interfaces between sub-programs*. Compositions of sub-programs may serve as sub-programs for even more complicated tasks.

In this paper it is assumed that the ‘divide and conquer problem’ can be divided and partly conquered by decomposing it into two problems, namely, the ‘*dividing-problem*’, and the ‘*conquering-problem*’.

The ‘dividing-problem’ is the problem of structuring all kinds of sequences of events and/or actions into ‘parts that belong together’. It is the problem of deciding *what* a good sub-program is, which its initial conditions are, and when it ends. The ‘dividing-problem’ has to do with unsupervised learning. It has been addressed in [10], [11], and [12].

The ‘conquering-problem’ is to select from many available sub-programs and to combine them in a way that allows to reach a given goal state.

In the sequel the ‘conquering-problem’ will be isolated and studied under the

assumption that the ‘dividing-problem’ is already solved. In contrast to previous approaches for credit assignment ‘from time slice to time slice’ the approach described in the next section allows extensive ‘credit assignment jumps through time’.

2 Learning to Generate Sub-Goals

2.1 The basic approach

Figure 1 shows the basic components of a system consisting of a number of interacting neural networks. The heart of the system is a neural network with internal and external feedback, called the control network C . C serves as a program executer. While producing a sequence of outputs $o_p(t)$ during execution of program p , it receives as input a stationary vector s_p representing the start state, a stationary vector g_p representing a desired goal state, and time-varying input vectors $i_p(t)$ from the environment. The concatenation of s_p and g_p serves as a ‘program name’. We assume that C already has learned to solve a number of tasks. This means that there already are various working programs that actually lead from the start states to the goal states by which the programs are indexed. These programs may have been learned by an algorithm for dynamic networks (as described by the authors mentioned above), *or by a recursive application of the principle outlined below.*

Figure 1 shows a second important module: an adaptive evaluator network E which receives as input a start state s and a goal state g , and is trained to produce an output $e(s, g) \in [0...1]$ that indicates whether C knows a program that leads from the start state to or ‘close’ to the goal state. An output of 0 means that there *is* an appropriate sub-program, an output of 1 means that there is no such sub-program. An output between 0 and 1 means that there is a sub-program that leads from the start state to a state that comes close to the goal in a certain sense. This measure of closeness has to be given by some evaluative process that may be adaptive or not (it might be an adaptive critic based on TD-methods, for instance). E represents the system’s current model of its own capabilities. We assume that E has learned to correctly predict that each of the already existing sub-programs works. We also assume that E is able to predict the closeness of an end state of a sub-program to a goal state, given a start state. In the simplest case, E can be trained in an exploratory phase during which various combinations of start and goal states are given to the program executer.

Finally, the system contains an adaptive network S which serves as a sub-goal generator. With a given task p specified by a start/goal pair $(s_p^0, g_p = s_p^{n+1})$, the sub-goal generator receives as input the concatenation of s_p^0 and s_p^{n+1} .

The output of the sub-goal generator is a list of n sub-goals $s_p^1, s_p^2, \dots, s_p^n$. Like the goal, a sub-goal s_p^i is an activation patterns describing the desired

external input at the end of some sub-program, which also is the start input for the next sub-program. After training the subgoal generator, the list of sub-goals should satisfy the following condition:

$$e(s_p^0, s_p^1) = e(s_p^1, s_p^2) = \dots = e(s_p^n, s_p^{n+1}) = 0.$$

This means that there exists a sub-program leading from the start state to the first sub-goal, and another subprogram program leading from the first sub-goal to the second sub-goal etc. until the final goal is reached.

How does the sub-goal generator, which initially is a *tabula rasa*, learn to generate appropriate sub-goals? We take $n + 1$ copies of E , and connect them to the sub-goal generator as shown in figure 2. The desired output of each of the copies is 0. For all positive outputs of some copies, error gradients are propagated through E 's copies down into the sub-goal generator. E (as well as its copies, of course) *remains unchanged* during this procedure. Only the weights of the sub-goal generator change according to

$$\Delta W_S^T = \eta_S \frac{\partial \sum_{i=0}^n \frac{1}{2} e(s_p^i, s_p^{i+1})^2}{\partial W_S} = \eta_S \sum_{i=0}^n \frac{\partial s_p^i}{\partial W_S} \frac{\partial e(s_p^i, s_p^{i+1})^T}{\partial s_p^i} e(s_p^i, s_p^{i+1})$$

where η_S is the learning rate of the sub-goal generator, W_S is its weight vector, and ΔW_S its increment. For a given problem the procedure is iterated until the complete error is zero (corresponding to a solution obtained by combining the two sub-programs), or until a local minimum is reached (no solution found). The gradient descent procedure is used for a search in sub-goal space. This works because *the effects of programs are made differentiable with respect to program names* (= start/goal combinations). This contrasts the approach of 'back-propagation through time' which makes the effects of programs differentiable with respect to whole action sequences (instead of selectively focussing on more abstract *short representations* of action sequences).

2.2 Cascaded sub-goaling with a recurrent sub-goal generator

Instead of using different sub-goal generators for different numbers of sub-goals, we can change our basic architecture such that only one sub-goal generator is necessary for generating arbitrary numbers of sub-goals.

The idea is to take a recurrent sub-goal generator which at a given time step produces only one sub-goal. At the next time step this sub-goal is fed back to the start input of the same sub-goal generator (while the goal input remains constant). To adjust the weights of the sub-goal generator, we can use an algorithm inspired by the 'back-propagation through time'-method: Successive sub-goals have to be fed into copies of E as shown in figure 3 (figure 3 shows the special case of three sub-goals). Gradient descent requires to change W_S

according to the sum of all gradients computed for the various copies of S . (Of course, E 's weight vector has to remain constant during S ' credit assignment phase.)

While unfolding the S/E system in time, it is not necessary to build real copies of E and S . It suffices if during activation spreading each unit in E and C stores its time-varying activations on a stack, from which they are popped during back-propagation phase.

2.2.1 How Many Sub-Goals for which Task?

Perhaps the simplest answer to this question is: With a given task, first make a try without any sub-goal. If this does not work, try one sub-goal, then two, etc.

An extension of the trial-and-error approach would be to train a *fourth* network to map start/goal combinations to numbers of sub-goals. The training signals could be obtained by the trial-and-error-method: For a given problem, the desired output of the fourth network should be a representation of the minimal number of necessary sub-goals.

2.3 An Experiment: Learning to Generate Sub-Goals for Simple Obstacle Avoidance

A simple experiment was conducted in order to demonstrate sub-goal learning. The programming was done by Rudolf Huber, a student of computer science at TUM.

A two-dimensional artificial 'world' covering the unit square was constructed. An artificial 'animal' controlled by the program executor was able to move around in the world. The program executor's output was four-dimensional (one output unit for each of the directions 'north', 'south', 'east', 'west'). At a given time step, the activation of each output unit (ranging from 0 to 1) was divided by 20, the animal's move was calculated by adding the four corresponding vectors (thus the maximal stepsize in each direction was 0.05).

In the center of the world there was an obstacle (indicated by the black square in figure 4). If the animal hit the obstacle, it had to stop.

Conventional back-propagation (3-layer feedforward nets with logistic activation functions were employed, no recurrent connections were necessary) was used to train the program executor to move the animal in a straight line from randomly chosen points (starts) to other randomly chosen points (goals), until it hit the obstacle or until the number of time steps exceeded 20. The training procedure was as follows: At each time step of an action sequence the straight line leading from the current position to the goal was computed, and the corresponding desired output of the executor served as a training signal. Both start and goal states were indicated by pairs of coordinates of corresponding points.

The executer had 20 hidden units and experienced 100000 action sequences during the training phase.

The evaluator was trained by randomly selecting start/goal combinations, executing them with the program executer, and watching the result. The output of the evaluator (20 hidden units) was trained to be 1.0 in cases with the final position of the animal being more than 0.1 away from the goal. It was trained to be 0.0 in cases with the final position matching the goal. In between linear interpolation was used. The evaluator experienced 1000000 examples during the training phase. Both the executer and the evaluator were trained with a learning rate of 0.05.

The reason for choosing a comparatively simple environment was to isolate the sub-goal generation process from effects that could be introduced by an *adaptive on-line* evaluation function (an adaptive critic, say). For our simple environment it was easy to define a *prewired* evaluation function. (Future research will focus on parallel on-line learning of all components of the system, but, as always, it is preferable to proceed incrementally from small problems to bigger ones.)

Figure 4 shows the ‘world’ and traces of the animal for some of the many programs successfully executed by the program executer. (Due to the imperfect executer, not all of these traces correspond to perfect straight lines.)

In the final phase the sub-goal generator (20 hidden units) was trained: Combinations of start and goals states that did not have a working program associated with them were given to the sub-goal generation process described in the last section. *With $\eta_S = 1.0$, within about 10 iterations the sub-goal generator actually found appropriate sub-goals for given start/goal combinations.* See figure 5 for one out of many examples.

By using more examples and smaller learning rates for training the subgoal generator, it soon learned to generate appropriate subgoals for a whole variety of situations.

3 Conclusion

By giving a constructive example it has been demonstrated that so-called ‘higher-level symbolic processes’ (in our case sub-goal generation) and so-called ‘sub-symbolic’ neural networks can be made compatible. This may be viewed as one first step towards neural network architectures that bridge the gap between ‘sub-symbolic’ and ‘symbolic’ computation. A quite different approach to adaptive sub-goaling has been described recently [12].

References

- [1] C. W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts, Dept. of Comp.

and Inf. Sci., 1986.

- [2] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846, 1983.
- [3] M. I. Jordan. Supervised learning and systems with excess degrees of freedom. Technical Report COINS TR 88-27, Massachusetts Institute of Technology, 1988.
- [4] Nguyen and B. Widrow. The truck backer-upper: An example of self learning in neural networks. In *IEEE/INNS International Joint Conference on Neural Networks, Washington, D.C.*, volume 1, pages 357–364, 1989.
- [5] T. Robinson and F. Fallside. Dynamic reinforcement driven error propagation networks with application to game playing. In *Proceedings of the 11th Conference of the Cognitive Science Society, Ann Arbor*, pages 836–843, 1989.
- [6] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229, 1959.
- [7] J. H. Schmidhuber. Learning algorithms for networks with internal and external feedback. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, *Proc. of the 1990 Connectionist Models Summer School*, pages 52–61. San Mateo, CA: Morgan Kaufmann, 1990.
- [8] J. H. Schmidhuber. A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, 1(4):403–412, 1990.
- [9] J. H. Schmidhuber. Recurrent networks adjusted by adaptive critics. In *Proc. IEEE/INNS International Joint Conference on Neural Networks, Washington, D. C.*, volume 1, pages 719–722, 1990.
- [10] J. H. Schmidhuber. Towards compositional learning with dynamic neural networks. Technical Report FKI-129-90, Institut für Informatik, Technische Universität München, 1990.
- [11] J. H. Schmidhuber. Adaptive decomposition of time. In O. Simula, editor, *Proceedings of the International Conference on Artificial Neural Networks ICANN 91*. Elsevier Science Publishers B.V., 1991.
- [12] J. H. Schmidhuber. A neural sequence chunker. Technical Report FKI, Institut für Informatik, Technische Universität München, 1991.
- [13] J. H. Schmidhuber. Reinforcement learning in markovian and non-markovian environments. In D. Touretzky and D. S. Lippman, editors, *Advances in Neural Information Processing Systems 3, in press*. San Mateo, CA: Morgan Kaufmann, 1991.

- [14] P. J. Werbos. Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 2:179–189, 1990.