A First Approach to Tackling Planning Problems with Neural Networks

S. Fernández, I.M. Galván, R. Aler
Universidad Carlos III de Madrid
Avenida de la Universidad, 30
28911 Leganés (Madrid), España
sfarregu@inf.uc3m.es, igalvan@inf.uc3m.es, aler@inf.uc3m.es

Abstract

Many different machine learning techniques have been used to learn control knowledge for planning. However, subsymbolic techniques have not been widely used, in particular artificial neural networks. In this paper, a feedforward neural network (FNN) will be used to learn a heuristic function for improving planning efficiency. The main aim of this paper is to present a preliminary study about the issues and the ability to use FNN in this context.

Introduction

It is well known that domain independent planning, which is most usually based on search, is not efficient. In order to make planning efficient, knowledge about the domain must be injected into the system. A popular approach in the machine learning field is to supplement domain independent planners with control knowledge learned automatically. A large amount of work has been done in this area. Please, see (Zimmerman & Kambhampati 2001) for a very good summary of most of the relevant work. However, not all machine learning techniques are equally well represented.

The main aim of this paper is to explore new learning mechanisms and control knowledge representations which have not been widely used. In particular, we study the performance of FNN (Rumelhart, Hinton, & Williams) to solve this kind of problems.

In this paper we train a FNN to be used as a heuristic function to improve forward search performance for planning. The network will learn what operator to use next, from examples that represent planning problems (i.e. initial and final planning states).

However, control knowledge must be applied to problems of varying sizes (for instance, in the blocks world, different problems can have an incrasing number of blocks). When symbolic representations (like control rules (Minton *et al.* 1989, Aler, Borrajo, & Isasi 2001) or PRS's (Khardon 1999b, Khardon 1999a)) are used, this is not a problem. But it presents a challenge if FNN are to be used, because their inputs are of fixed size.

In this paper, we have faced this problem by decomposing a planning problem in different instantiations. Every instantiation has constant size and can be fed into the neural network. Then, the different outputs of the network must be combined somehow to produce the final answer. This is similar, in symbolic terms, to allowing PRS rules to have only a limited number of variables (Khardon 1999a).

Actually, this problem can be seen in a more general way as follows: machine learning techniques which use the propositional ("attribute and value") representation assume problems of fixed size. In the case of planning (in the blocks world, for instance), this means that only problems with up to a fixed number of blocks can be solved. If it is desired to solve problems with any number of blocks but learning with only simple problems, it is neccesary to use relational representations, of the ILP kind. In this paper, we are trying to apply a propositional representation to solve a relational problem.

So far, we have done only some preliminary testing of our approach in the blocks world. Here, we are interested in how the performance of the FNN changes as problem complexity increases.

The structure of this paper is as follows. In Section we describe our approach. Section presents somre preliminary experiments.

Description of the Approach

Representing planning problems for neural networks

In this paper we want to train a FNN to be used as a heuristic function to improve search performance for planning. In order to do so, it is necessary to define the inputs and the outputs of a FNN and how the training patterns will be codified. In our case, the inputs to the FNN will represent the current state in the search process and the goal state to be reached, and the output will be the instantiated planning operator that should be applied to the current state, to get to another state which is, hopefully, closer to the goal state. This process will be iterated until the goal state is reached.

In the STRIPS formalism, states are represented us-

ing predicates and objects. For instance, in a version of the blocks world, states are represented using five predicates (`arm-empty`, `holding(X)`, `clear(X)`, `on-table(X)`, and `on(X,Y)`) and names for the blocks (`A`, `B`, ...). States in the blocks world can be represented by instantiating the predicates with all the blocks. However, the size of a state increases with the number of blocks. But our goal is that the FNN can be applied to any world containing any number of blocks. Since FNN can accept only a fixed number of inputs, we have introduced the concept of 'variable': at any time the FNN will be only aware of a fixed number of variables which later will be instantiated in all possible ways with the blocks of the state. For instance, in order to represent a state containing three blocks (A, B, C) with only two variables (X, Y), we would have to consider 6 instantiations: (X=A,Y=B), (X=A,Y=C), (X=B,Y=A), (X=B,Y=C), (X=C,Y=A), and (X=C,Y=B).

Now, all the possible combinations of the predicates and the variables are computed. With 2 variables and 5 predicates, the combinations are: {`on(X,Y)`, `on(Y,X)`, `on-table(X)`, `on-table(Y)`, `clear(X)`, `clear(Y)`, `arm-empty`, `holding(X)`,. `holding(Y)` }. Each one of these predicates is represented with 1 if they are true in a particular instantiation and as 0 if they are false. Hence, one instantiation requires 9 bits to be represented. Therefore, in this case, the input to the FNN is made of 18 bits, 9 for the initial state and 9 for the goal state. Given an initial state and a final state, there is a binary 18-bit input pattern for every instantiation of the variables.

For instance, let us consider two variables X and Y, and three blocks. If the initial state is {on(A,B), clear(A), on-table(B), on-table(C), clear(C), arm-empty} and the final state is {on(A,B), on(C,A), clear(C), on-table(B), arm-empty}, the instantiation (X=A,Y=B) would be represented as:

100110100,100000100

In order to represent an initial and final state pair, there will be as many 18-bit binary patterns as possible instantiations depending on the number of variables. In this case there will be 6 input patterns.

As mentioned previously, the output of the FNN represents the operator that has to be applied in the current state to reach another state wich is closer to the final state. The output is made of as many bits as operators there are and another extra `no-op` bit. In the blocks world, there are four operators ({`stack(X,Y)`, `unstack(X,Y)`, `pick-up(X)`, `put-down(X)` }), so the output consists of 5 bits. The variables of the operators correspond to the variables of the instantiations. For every instantiation (X=x,Y=y), if the operator to apply to the current state is `op(X=x,Y=y)` then the bit corresponding to `op` is set to 1, and the rest of the bits are set to 0. Otherwise, the `no-op` bit is set to 1 and the rest to 0. Table 1 displays the five bits for the output of the FNN for each one of the possible instantiations, in case the operator to apply is `stack(A,B)`. In all but the first of the instantiations, the `no-op` bit is set to 1 because the variables of the instantiation are different to the variables of the operator.

Table 1: Outputs of the network for every instantiation of stack(X=A,Y=B), the 5 bit approach. S=stack, U=Unstack, PU=Pick-up, PD=Put-down.

`stack(X=A,Y=B)`					
X,Y	No-op (1)	S. (2)	U. (3)	PU. (4)	PD. (5)
A,B	0	1	0	0	0
A,C	1	0	0	0	0
B,A	1	0	0	0	0
B,C	1	0	0	0	0
C,A	1	0	0	0	0
C,B	1	0	0	0	0

Table 2: Outputs of the network for every instantiation of stack(X=A,Y=B), the 8 bit approach. S=stack, U=Unstack, PU=Pick-up, PD=Put-down.

`stack(X=A,Y=B)`				
	Operator			
X,Y	S. (1)	U. (2)	PU. (3)	PD. (4)
A,B	1	0	0	0
A,C	0	0	0	0
B,A	0	0	0	0
B,C	0	0	0	0
C,A	0	0	0	0
C,B	0	0	0	0
	No-operator			
X,Y	S. (5)	U. (6)	PU. (7)	PD. (8)
A,B	0	0	0	0
A,C	1	0	0	0
B,A	1	0	0	0
B,C	1	0	0	0
C,A	1	0	0	0
C,B	1	0	0	0

Another way we have used to represent the output is by using 8 bits. The meaning of the first four bits is the same than in the later paragraph. The other four bits expand the `no-op` bit. They represent the operator that should be used, in case the variables of the instantiation were the right ones as shown in Table 2.

Obtaining the training patterns

Given an initial situation, a final situation, and the first instantiated operator that must be applied to reach that final situation (let us call them a planning triplet), several training patterns will be obtained. The number of training patterns obtained depends on the number of possible instantiations which in turn depends on the number of variables. For instance, let us suppose that we consider two variables and three blocks. Also, we have the following planning triplet:

- Initial state: all the three blocks on the table
- Final state: a tower made of the three blocks (A on

Table 3: Obtaining training patterns.

	Input		Output
Instantiation	Initial state	Final state	
(X=A,Y=B)	001111100	100010100	10000
(X=A,Y=C)	001111100	000110100	10000
(X=B,Y=A)	001111100	010001100	00010
(X=B,Y=C)	001111100	100100100	00010
(X=C,Y=A)	001111100	001001100	10000
(X=C,Y=B)	001111100	011000100	10000

B, and B on C)

- The first operator to apply is `pick-up`(B)

From this planning triplet, six training patterns can be obtained (with a 5 bit output), as it is shown in Table 3.

There will be many cases where several training patterns with the same input have a different output associated. That is, they are contradictory patterns. To deal with this problem we have tried two approaches:

1. **OR approach:** a single training pattern is obtained from all the training contradictory patterns by using the OR function on the output. In that case, a pattern can have several bits set to 1 at the output.
2. **AVERAGE approach:** a single training pattern is obtained from all the training contradictory patterns by computing the probability of every bit in the output being 1. In this case, the bits at the output are real numbers between 0 and 1.

Using the network to solve planning problems

First of all, given an initial and final planning situation, the set of instantiated operators that could be applied in the initial situation is obtained. Now, the problem is to choose one of them. The steps to determine it are the following:

1. All possible instantiations of the initial and final situation are obtained. Therefore, a set of input patterns to the FNN are obtained.
2. For each one of the input patterns, the output of the FNN is calculated. Therefore, now there is a set of outputs that need to be combined.
3. Also, for each instantiation and for each operator that could be applied in the initial situation, the set of outputs that the network should give are determined. Then, they are compared with the actual outputs of the FNN. This is actually done by substracting the actual output of the FNN and the desired output. The instantiated operator that gets the minimum value (that is, which is closest to the desired output) is applied and an new initial situation is obtained.
4. These steps are applied until the final situation is reached.

Table 4: How the FNN is used to select the next operator to apply.

	Actual output	`PU.`(B) output	`PU.`(A) output	`PU.`(C) output
X,Y				
A,B	0.99 0.00 0.01 0.00 0.00	10000	00010	10000
A,C	0.98 0.10 0.02 0.01 0.10	10000	00010	10000
B,A	0.01 0.12 0.01 0.87 0.00	00010	10000	10000
B,C	0.01 0.12 0.01 0.99 0.00	00010	10000	10000
C,A	0.99 0.00 0.02 0.01 0.00	10000	10000	00010
C,B	0.90 0.00 0.09 0.10 0.00	10000	10000	00010

For example, let us assume that the planning problem to solve is:

- Initial state: all the three blocks on the table
- Final state: a tower made of the three blocks (A on B, and B on C)

In this case, the solution is to apply `pick-up`(B). The set of operators that could be applied in the initial situation is: {`pick-up`(B), `pick-up`(A), `pick-up`(C)}. The way to decide the operator that should be applied is illustrated in Table 4.

From this planning triplet, the following six training patterns can be obtained (when the output is made of 5 bits):

As it can be seen in Table 4 the instatiated operator which is closest to the actual output is `pick-up`(B).

Experimental Results

The experimentation was divided in two parts. First, we compared different experimental configurations of the system. And second, we selected the best configuration and then it was tested with some simplified problems.

Comparing different configurations

We have tested the following configurations (so far, we have only used the 5 bit approach):

- To compact equal patterns using the OR function
- To compact equal patterns using the AVERAGE fuction

We generated all the possible planning problems in the blocks world with 3 blocks and with 4 blocks and we have trained the FNN according to the following configurations:

- 3 blocks world using the OR function to compact
- 3 blocks world using the AVERAGE function to compact
- 4 blocks world using the OR function to compact
- 4 blocks world using the AVERAGE function to compact

Table 5: Empirical results for the different configurations.

Blocks	Var.	Outputs	Compact	Training error	Testing error
3	2	5	OR	0.009	0.052
4	2	5	OR	0.004	0.005
4	2	5	AVE.	0.001	0.001
4/3	2	5	OR	0.005	0.043
4/3	2	5	AVE.	0.004	0.033
4	3	5	OR	0.012	0.105
4	3	5	AVE.	0.033	0.112

- 3 and 4 blocks world using the OR function to compact
- 3 and 4 blocks world using the AVERAGE function to compact

Configuration were tested using 10-fold cross-validation. The results are shown in Table .

According with the results it seems slightly better to compact equal patterns using the AVERAGE approach insteed of the OR approach.

Testing the FNN

In this subsection the FNN trained with examples from the blocks world with 3 and 4 blocks and the AVERAGE approach has been chosen for testing.

We have considered two kinds of problems. The goal of the first ones is to stack a block C1 on C2. The initial situation contains several blocks on both C1 and C2, that must be unstacked before being able to stack C1 on C2. For instance, in the following there are 2 blocks on C1 and 0 blocks on C2:

- Final state: ((ON C1 C2) (ON-TABLE C2) (ON-TABLE C3) (ON-TABLE C4) (CLEAR C1) (CLEAR C3) (CLEAR C4) (ARM-EMPTY))
- Initial state: ((ARM-EMPTY) (ON-TABLE C1) (ON C3 C1) (ON C4 C3) (CLEAR C4) (ON-TABLE C2)(CLEAR C2))

The second kind of problems consist in bulding towers from an initial situation in which all the blocks are on the table. For instance, the goal of the following problem is to build a tower of two blocks:

- Final state: ((ARM-EMPTY) (ON-TABLE C2) (ON C1 C2) (CLEAR C1))
- Initial state: ((ARM-EMPTY) (ON-TABLE C1) (ON-TABLE C2) (CLEAR C1) (CLEAR C2))

We have decided to have two different kinds of problems, which can be easily scalated as a methodological help for testing the system. This way, it will be possible to know in which cases the FNN has learned something useful, and how well it scalates for more difficult problems. This is not possible if the system is tested only in randomly generated problems.

So far, we have tested the FNN with the two kind of problems, in the blocks world with 3, 4, 5, and 6 blocks. It has been observed that the FNN found the optimum solution for the first kind of problems. However, it is also observed that for the second kind of problems, the system is not very efficient.

Conclusions

In this paper we have used a FNN to be used as a heuristic function to improve forward search performance for planning. The network will learn what operator to use next from examples that represent planning problems (i.e. initial and final planning states).

The preliminary results show that the FNN could be an appropriate way to approach this problem. However, the experiments show that more work is required. We believe that to improve results it is necessary to find better ways to combine the different outputs of the network to obtain which operator should be applied. Also, it could be interesting to try different representations for the network.

Finally, in order to evaluate the system more accurately, it would be necessary to measure how much time is saved by using the FNN in comparison with a random search and also other domain independent planners.

References

Aler, R.; Borrajo, D.; and Isasi, P. 2001. Learning to solve planning problems efficiently by means of genetic programming. *Evolutionary Computation* 9(4):387–420.

Khardon, R. 1999a. Learning action strategies for planning domains. *Artificial Intelligence* 113:125–148.

Khardon, R. 1999b. Learning to take actions. *Machine Learning* 35(1):57–90.

Minton, S.; Carbonell, J.; Knoblock, C.; Etzioni, O.; and Gil, Y. 1989. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40.

Rumelhart, D.; Hinton, G.; and Williams, R. *Learning Internal Representations by Error Propagation in Parallel Distributed Processing.*

Zimmerman, T., and Kambhampati, S. 2001. What next for learning in ai planning. In *Proceedings IC-AI.*