Transforming Occupancy Grids Under Robot Motion

Joris W.M. van Dam and Ben J.A. Kröse and F.C.A. Groen

Faculty of Mathematics and Computer Science, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
dam@fwi.uva.nl

Abstract: This paper addresses the problem of how occupancy values from one occupancy grid can be used to calculate occupancy values in another grid, where the latter is rotated and/or translated with respect to the former. The mapping is described in terms of a neural network, of which the parameters are learned from examples. An activation function is derived taking into account that the input and output values represent probabilities. It is also determined how many points should be taken in a learning sample to optimise learning speed.

1 Introduction

To represent the working environment of an autonomous mobile robot, occupancy grids can be used [2]. An occupancy grid partitions the robot's environment in a number of subspaces, each of which is represented by a pixel. The pixel-*value* gives the *probability* of the subspace being occupied by some obstacle. These grids can be used in mobile robot navigation [3, 5, 1] and in sensor data fusion [6, 4].

Usually, occupancy grids are defined to be 'world fixed', i.e., they give a *global representation* of the working environment and the partitioning of the environment remains unchanged. We have, however, chosen to work with a *local* representation of the environment. Grids are now defined to be 'robot fixed' and the partitioning changes every time the robot moves. This paper addresses the problem of how, after a robot move, the information in the occupancy grid *before* the move can be used to calculate pixel-values in the grid *after* the move. This is called the *transform of occupancy grids*.

If the transform of grids is to be computed with conventional methods, not only the exact robot move (e.g., the translation and rotation of the robot) needs to be known, but also the exact shapes and sizes of the pixels in the grid are to be known and to be chosen appropriately. Here, neural networks are used to *learn* the transform of grids of unknown sizes, shapes and dimension as a function of a known robot move. The remainder of the text shows how such a transform can be learned for a single, a-priori fixed, arbitrary robot move.

2 The network model

This section defines a neural network architecture, derives an appropriate activation function, and suggests a learning procedure to learn the transform of occupancy grids for a single robot move.

Define a set of *input* neurons X^A with activations x_j^A giving the pixel-value (the probability of being occupied) of pixel j in the grid *before* the move. Let X^B be a set of *output* neurons with activations x_i^B representing the pixel-value of pixel i in the grid *after* the move. We have $j = 1, 2, \dots N^A$ and $i = 1, 2, \dots N^B$ with N^A and N^B the sizes of the grids. Each output neuron i is connected to all input neurons j with weights w_{ij}.

Next, activation functions $\mathcal{F}()$ are to be determined such that $x_i^B = \mathcal{F}(w_{ij}, x_j^A)$ give the occupancy values of the grid after the move. If a pixel j was to be split up further in a number of sub-pixels

j_k, it follows from the interpretation of pixel-values that

$$x_{j_k} = \frac{A(x_{j_k})}{A(x_j)} x_j \tag{2.1}$$

where $A()$ calculates the area of a subspace represented by a pixel. For this, we assume that the size of the obstacle occupying the pixel's subspace is minimal. Conversely, if a pixel i would be composed from the union of disjoint subspaces i_h, it follows that

$$x_i = 1 - P(\text{all } i_h \text{ are empty}) = 1 - \prod_h (1 - x_{i_h}) \tag{2.2}$$

For this, we assume that all occupancy probabilities x_{i_h} are independent and that $(1 - x_{i_h})$ gives the probability of subspace i_h being empty. Examining figure 2.1, which envisages the grid after a

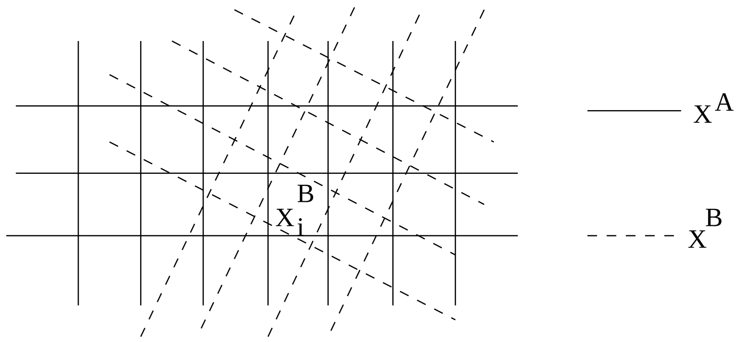

Figure 2.1: Pixel x_i^B overlaying several pixels x_j^A. X^B is the transform of X^A.

robot move overlaying the grid before the move, and using (2.1) and (2.2) gives:

$$\mathcal{F}(w_{ij}, x_j^A) = 1 - \prod_j (1 - w_{ij} x_j^A) \tag{2.3}$$

with

$$w_{ij} = \frac{A(x_i^B \cap x_j^A)}{A(x_j^A)} \tag{2.4}$$

This expression for the optimal weight values explains the statement made earlier, that conventional calculation of weight values can only be performed if the sizes and shapes of the pixels are known and chosen appropriately. A learning procedure is to be applied to estimate these optimal weight values.

If examples of grids and their transform $(X^A, \hat{X}^B)$ can be given, with $\hat{X}^B$ the *correct* grid after the move, the error in the learned transform $\mathcal{F}(w_{ij}, x_j^A)$ is given by

$$\sum_{\text{examples}} \sum_i (\hat{x}_i^B - x_i^B)^2 \tag{2.5}$$

We can now define a learning rule that follows the gradient of this error function with respect to the weight values, known as the δ-rule. With the activation function in (2.3) this gives

$$\Delta w_{ij} = -\gamma(\hat{x}_i^B - x_i^B) \cdot \frac{\partial}{\partial w_{ij}} \mathcal{F}(w_{ij}, x_j^A) = \gamma(\hat{x}_i^B - x_i^B) \cdot x_j^A \cdot \prod_{k \neq j} (1 - w_{ik} x_k^A) \tag{2.6}$$

This gives rise to the following learning algorithm:

1. create a sample environment: choose a set of m randomly distributed points in the input space;
2. create a learning sample: for each point, determine the unique subspaces j and i of the original and the transformed grid, respectively, that are occupied by these points. Set $x_j^A = 1$ and $\hat{x}_i^B = 1$;
3. calculate occupancy values: use (2.3) to calculate the current estimates of occupancy values x_i^B;
4. update parameters: use (2.6) to update the weight values w_{ij}.

3 Speeding up the algorithm

In this section it is determined how many randomly distributed points should be chosen in each sample environment (m in the algorithm of section 2). Taking one point per sample means that only a small number of weights is updated each time a sample is presented. However, if more points per sample are taken, the product term in (2.6), which *reduces* the update if it is likely that $x_j^A = 1$ and $x_i^B = 1$ are *not* caused by a *single* point in the learning sample, will slow down learning considerably. These two considerations are combined quantitatively to calculate an optimal value for m.

Equation (2.6) shows that the number of weights for which $\Delta w_{ij} \neq 0$ is proportional to the number of pixels j with $x_j^A = 1$, say s. If m points are chosen in the sample environment, it follows that the expected value of x_j^A is given by:

$$E[x_j^A] = 1 \cdot P(x_j^A = 1 \mid m) + 0 \cdot P(x_j^A = 0 \mid m) = 1 - \left(\frac{A(X^A) - A(x_j^A)}{A(X^A)}\right)^m = 1 - \left(\frac{N^A - 1}{N^A}\right)^m \quad (3.1)$$

with $A()$ once again the 'area'-function. Here, we take $\forall j \; A(x_j^A) = 1$ which gives $A(X^A) = N^A$, the number of pixels in the original grid. The expected value of s is now given by:

$$E[s] = N^A \cdot P(x_j^A = 1 \mid m) = N^A \left[1 - \left(\frac{N^A - 1}{N^A}\right)^m\right] \quad (3.2)$$

Another look at equation (2.6) shows that each Δw_{ij} contains a multiplication factor $g_{ij} = \prod_{k \neq j}(1 - w_{ik} x_k^A)$. With one point per sample environment ($m = 1$ and $s = 1$) we have $x_j^A = 1$ implies $g_{ij} = 1$. However, taking $m > 1$ and thus $s > 1$ gives $g_{ij} < 1$, a reduction of the update. The exact value of g_{ij} is also a function of r_i: the number of input neurons k for which $w_{ik} \neq 0$. The learning procedure starts with $r_i = N^A$, a fully connected, randomly initialised network. However, in practical situations it can be expected that r_i converges to a small value. This gives that, as learning proceeds, r_i decreases and consequently g_{ij} increases towards 1.

We have for the expected value of g_{ij}:

$$E[g_{ij}] = E[\prod_{k \neq j}(1 - w_{ik} x_k^A)]$$

We now assume independence of x_k^A and x_l^A, which is justified in our algorithm. We also take $x_j^A = 1$, since $x_j^A = 0$ implies $\Delta w_{ij} = 0$, we use an average weight value $w_{ik} = e$ and use (3.1) and r_i as defined earlier. This gives:

$$E[g_{ij}] = \prod_{k \neq j}(1 - E[w_{ik}]E[x_k^A]) = \left(1 - e\left[1 - \left(\frac{N^A - 1}{N^A}\right)^{m-1}\right]\right)^{r_i} \quad (3.3)$$

We now have that taking $m > 1$ points per sample environment, the number of updates is proportional to (3.2), while each update is reduced with a factor given by (3.3). We take m^*, the optimal number of points per sample to be the maximum of the resulting net speed-up:

$$m^* = \max_m \left\{ N^A \left[1 - \left(\frac{N^A - 1}{N^A}\right)^m\right] \cdot \left[1 - e\left(1 - \left(\frac{N^A - 1}{N^A}\right)^{m-1}\right)\right]^r \right\} \quad (3.4)$$

4 Implementation and results

The algorithm given in section 2 was implemented for a fixed robot move. Also, (3.4) was used to compute m^* every 1000 learning samples, using $e = 0.5$. The samples were processed in batches of 100. Every 100 samples we took

$$w_{ij} = w_{ij} + \gamma \frac{\sum_{100 \text{ samples}} \Delta w_{ij}}{E[m_j]}$$

with $E[m_j]$ the expected number of updates for weights w_{ij}, calculated using (3.2). Learning parameter γ was taken to decrease exponentially from 0.4 to 0.01.

Using knowledge on the robot move and the sizes and shapes of pixels, optimal weight values were calculated directly from (2.4). These values were compared with the learned values. Results show the weights are estimated too high with an error of approximately 20%. This is due to the fact that as learning proceeds, we take m *fixed* at $m^* \gg 1$ to speed up the algorithm. This gives an unfair bias in learning the weight values. The algorithm introduced in section 2 is derived for *varying* values of m. Current research aims to adapt the algorithm for m fixed at larger values.

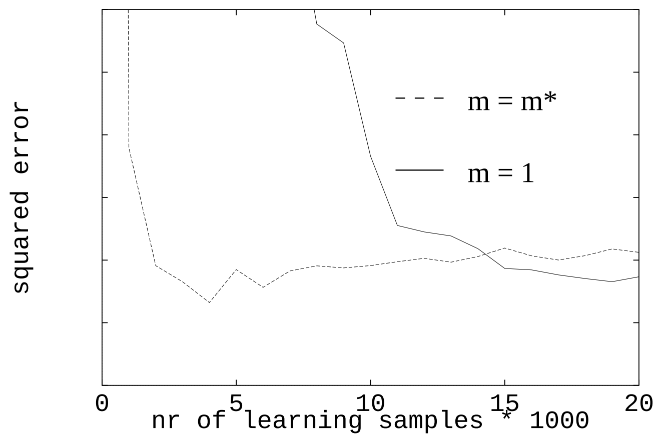

Figure 4.1: The total squared error in the learned transform calculated every 500 samples for $m = 1$ and $m = m^*$.

In figure 4.1 some results are given. The squared errors are calculated over 10,000 test sample grids every 500 learning samples. The numbers of points in test samples were drawn at random from $\{1, 2, \ldots N^A\}$ for each sample, explaining fluctuations in the squared error. The graphs show that indeed taking $m = m^*$ results in a significant speed-up of the algorithm. However, as learning proceeds, m^* converges to a value $m^* \gg 1$ (as r_i converges to a small value) and weights are estimated too high. This explains that the algorithm converges to an error value which is higher than can be obtained with $m = 1$.

References

[1] A. Elfes, *Sonar-Based Real-World Mapping and Navigation.* IEEE Journal of Robotics and Automation, June 1987.

[2] Alberto Elfes. *Using Occupancy Grids for Mobile Robot Perception and Navigation.* Proceedings of the 1989 IEEE Int. Conference on Robotics and Automation.

[3] G. Fahner, R. Eckmiller, *Learning Spatio Temporal Planning from a Dynamic Programming Teacher: A Feed Forward Net for the Moving Obstacle Avoidance Problem.* Proc. of ICANN92, Brighton.

[4] L. Matthies, A. Elfes, *Integration of Sonar and Stereo Range Data Using a Grid-Based Representation.* IEEE Conf on Robotics and Automation, 1988.

[5] J. del R. Millan, *Building Reactive Path-Finders through Reinforcement Connectionist Learning: Three Issues and An Architecture.* Proc. 10th European Conf on AI, Vienna Austria, 1992.

[6] H.P. Moravec, A. Elfes, *High Resolution Maps from Wide Angle Sonar.* IEEE Conf on Robotics and Automation, 1985.

[Rabinowitz] P.J. Davis and P. Rabinowitz, *Methods of Numerical Integration.* Chap 5.9, Multiple Integration by sampling.