Partition Nets: An Efficient On-Line Learning Algorithm

Karl F. MacDorman
Department of Systems and Human Science
Graduate School of Engineering Science
Osaka University
Toyonaka, Osaka 560-8531 JAPAN
http://robotics.me.es.osaka-u.ac.jp/~kfm

Abstract

Partition nets provide a fast method for learning sensorimotor mappings. They combine the generalizing power of neural networks with the "one shot" learning of instance-based search algorithms. Partition nets adjust receptive fields and allocate network weights "on the fly," in response to the *local* sample density and complexity of the mapping. They can help a robot adapt quickly to bodily and environmental changes. Thus, they satisfy several important criteria for a cognitive theory of sensorimotor learning.

1 Introduction and motivation

This paper proposes a new method of learning a sensorimotor mappings for successfully navigating a mobile robot in a dynamic environment. It can also be applied to learning many other kinds of nonlinear mappings (e.g., hand-eye mappings in a humanoid robot). The method incorporates important aspects of animal learning to ensure responsiveness to unexpected bodily and environmental change.

1.1 The importance of learning in sensorimotor activity

The survival of even the simplest organisms depends on their ability to orient their movements relative to their environment. The standard approach to modeling this relationship in robotics relies on analytically derived equations [17]. It demands the accurate measurement and representation of the robot's dynamics (which is often impossible) and the use of reliable, high precision equipment with known performance characteristics. A humanoid robot designed along these lines, for example, cannot cope with an elbow joint that sticks or a camera eye that is knocked out of alignment. Any unexpected change in the robot's dynamics, or the dynamics of the objects it manipulates, is potentially catastrophic.

This rigidity contrasts with the robustness and adaptability of biological systems. While responsiveness to change does not always require learning, learning is essential to the speed, grace, and coordination of vertebrate activity. These features cannot be explained by servo-mechanical feedback loops given the relatively slow speed at which neural impulses travel. Human movement is highly resilient to environmental perturbations, with trajectory correction generally occurring in less than 100 ms. This suggests the presence of a learned sensorimotor mapping that compares visual, proprioceptive and motor signals to make open-loop adjustments [18, 8, 9], see [4] for a computer model. Moreover, only learning can explain the accuracy of ballistic movements (e.g., hitting a hole-in-one at golf), since external feedback concerning the relative positions of projectile and goal comes only after control of the object has ceased.

1.2 Key features of learning in animals

Vertebrates are capable of learning from even a single example. Sometimes this kind of learning goes amiss, as with the dog that bites any man in uniform. Generally though, as experience accumulates, the learner is able to move from reasoning explicitly from recollections of past situations to a kind of "automatic" response that does not refer to any situation in particular. It is commonly assumed that learned generalizations underlie this response.

The process of remembering instances and generalizing from them operates in the context of other cognitive processes that support it and contribute to its effectiveness. Orienting responses, for example, direct attention toward unexpected events, thus bringing conscious processing to bear on them. Conscious events take on a global influence and may be reasoned about abstractly. However, the power and flexibility of conscious processing come at a high computational cost. This may explain why conscious processing is essentially a limited resource, one that the brain cannot squander. Processing limitations may explain why we usually perform conscious tasks sequentially, such as reasoning from specific past events. Thus, learning is vital to off-loading conscious processing by helping the brain automate its responses to routine events. These events may then be handled simultaneously by massively parallel nonconscious processes.

Based on the above observations, we can flesh out a possible theory of sensorimotor learning:

(1) Developing a sensorimotor model is a matter of learning predictions concerning the consequences

of actions based on past experience. These predictions are derived from spatiotemporal correlations in sensory projections and motor signals.

(2) The predictions are contingent on the agent's motivation, internal condition, and its perceived relation to the environment. Therefore, only a subset of its predictions are active at any moment.

(3) Active predictions constitute the agent's conceptualization. They elicit anticipatory responses.

(4) Predictions are initially derived from recollections concerning action outcomes in similar situations. As learning progresses, generalizations replace predictions based on specific situations.

(5) Learning is proportional to the degree of unexpectedness. Learning does not occur when the the difference between the predicted and actual outcome is negligible. (This avoids wastage of memory and processing time and helps prevent the learning of noise.)

(6) Failed predictions result in orienting responses that direct attention toward the source of error. This brings the full repertoire of conscious processing to bear on the unexpected event, and the event may be integrated into a new perspective on the situation.

(7) Failed predictions may lead to the memorization of new instances, the forgetting of instances that no longer apply, and the relearning of generalizations from the new instances.

While this theory can be applied to the recognition of objects and events, in this paper we limit its scope to learning a sensorimotor mapping.

1.3 Nearest neighbors and neural networks

The contrast between recalling specific instances and responding from a learned generalization finds a natural analog among computational learning techniques. Algorithms and data structures for nearest neighbor search enable the efficient retrieval of past instances (prototypes, or categories) according to their similarity to some new instance. Both nearest neighbor [5] and neural network approaches [12, 13] have proven useful for learning sensorimotor predictions from past experience. However, they differ vastly in their performance characteristics.

Nearest neighbor learning converges immediately; it will converge to a final approximation once it has been exposed to all data points. Learning in feedforward neural networks by back propagation of error may require orders of magnitude more time [14, 15, 6], and the network may never reach a globally optimal solution. Nearest neighbor algorithms require on the order of $\log_2 N$ to N calculations to predict an output given an input, where N is the number of memorized data points; they require memory on the order of KN to store N K-dimensional points. Neural networks require on the order of W calculations, where W is the number of weights; they require memory on the order of $KN + W$ during training but only W thereafter. While nearest neighbor prediction accuracy often requires a large number of noise free data points, interpolating nearest neighbors (in noise free environments) or blending them (in noisy environments) can overcome this limitation.

	nearest neigbhor	*back prop.*
convergence	immediate	slow
access time	$O(\log_2 N), O(N)$	$O(W)$
memory	$O(N)$	$O(N) \rightsquigarrow O(W)$
resists noise	poor $\sim$ good	very good

The trade-offs between nearest neighbor methods and neural networks render the two approaches complementary — both as cognitive models and as practical engineering solutions.

2 Partition nets

Partition nets are a new method of learning sensorimotor (or other nonlinear) mappings that combine many of the advantages of nearest neighbor learning and neural network learning by the backpropagation of error. The name derives from the fact that the input space is partitioned into subspaces, each with its own neural network and bucket of points.

Key features of partition nets include the ability to

- learn in "one shot" from an instance;
- interpolate and blend across instances with sensitivity to the local sample density;
- converge quickly to a generalization (see Fig. 1);
- mix instance based and generalization based prediction to improve accuracy;
- resist noise;
- learn a mapping piecewise, thus saving memory;
- learn only when the CPU is idle;
- adapt neuronal receptive fields "on the fly" to the local complexity and sample density of the mapping;
- partition receptive fields without corrupting previously learned weights;
- allocate weights according to the local complexity and sample density;
- provide an "everywhere continuous" mapping;
- satisfy the sensorimotor prediction aspect of at least five out of the seven requirements (1-5) of the proposed theory in section 1.2.

2.1 How partition nets work

To illustrate their use, let's consider a mobile robot that, at time t, perceives an object at position (x_t, y_t) on its camera image planes. It needs to predict the location of the object (x_{t+1}, y_{t+1}) at time $t + 1$ after it has turned its left and right wheel by (l, r): $(x_{t+1}, y_{t+1}) = \boldsymbol{f}(x_t, y_t, l_t, r_t)$. More generally,

$$\vec{o}_{t+1} = \boldsymbol{f}(\vec{\imath}_t)$$

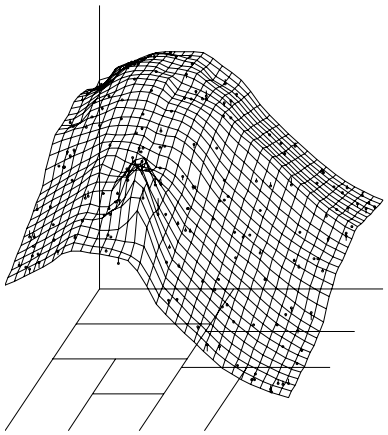

Figure 1: A 2D-to-1D mapping with partition nets after only a few second of learning. The coordinates vary from 0 to 1. Circles represent data points. All points lie on f. The length of the line above or below a circle gives the prediction error $f - g$. To use a partition nets Java tutorial, visit: http://robotics.me.es.osaka-u.ac.jp/~kfm

A memory $\vec{m}_t \equiv (\vec{\imath}_t, \vec{o}_{t+1})$ of an experience has input dimensions $\vec{\imath}_t$ constituted by the perceived state and action and output dimensions $\vec{o}_{t+1}$ constituted by the perceived outcome state. Since $f(\vec{\imath})$ is unknown, the robot must learn a function $g(\vec{\imath})$ that approximates $f(\vec{\imath})$.

Backprop. Partition nets use the back propagation learning algorithm [19] to minimize network error, but they can easily be generalized to other learning algorithms. Back propagation is a means of training nonlinear feed-forward neural networks. Using the standard *pattern by pattern* method, training entails repeatedly presenting the memorized data points in a randomly shuffled order at the input layer. The net input to each unit$_j$ in layer l is the sum of the weighted activations a_i from the layer below $(l-1)$ and its activation a_j is the sigmoid of that sum:

$$S_j = \sum_i a_i w_{j,i} \qquad a_j = \frac{1}{1+e^{-S_j}}$$

Activations for the hidden layers are calculated layer by layer in ascending order and then for the output layer $\vec{o} \equiv (a_1, ...a_O)$, where O is the output dimensionality. For the standard backprop algorithm, input activations range from 0 to 1 inclusively. The sigmoid function ensures that activations in the output and hidden layers also range between 0 and 1.

The synaptic weights for the output layer are at first adjusted to minimize prediction error, then (in descending order) for the hidden layers: $\Delta w_{j,i} = \eta \delta_j a_i$, where η is the learning rate. The deltas δ_j for the output layer are the second derivative of the sigmoid of the unit's net input multiplied by the difference between the target value and the output unit's activation:

$$\delta_j = \frac{e^{-S_j}}{(1+e^{-S_j})^2}(t_j - a_j)$$

The deltas for the hidden layers are calculated as follows:

$$\delta_j = \frac{e^{-S_j}}{(1+e^{-S_j})^2} \sum_k \delta_k w_{k,j}$$

From instance to generalization. One advantage of back propagation is that unlike traditional smooth functions it appears not to suffer from the curse of dimensionality [1, 2]. Nevertheless, back propagation may be unsatisfactory for on-line learning because of its slow convergence. If the environment changes, a robot needs to be able to respond immediately; it needs a provisional sensorimotor model until a better model has been learned. Partition nets base their predictions on a blending of M closest points in a network's subspace $\Re$ until the mean squared error (at the output layer) is low enough for the network to take over:

$$\mathcal{E} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{k=1}^{O} ((\vec{o}_{n+1})_k - g(\vec{\imath}_n)_k)^2 < \epsilon_1$$

where N is the number of points in $\Re$ and K is the output dimensionality. To focus learning and conserve memory, partition nets only memorize a new data point $\vec{p}_t$ if they failed to predict its output dimensions within a certain error $\|\vec{o}_{t+1} - g(\vec{\imath}_t)\| \geq \epsilon_2$. They forget all data points in a subspace once a function has been learned to the desired degree of approximation $\mathcal{E} < \epsilon_3$. The network is given a low priority thread so it can spend spare cycles learning without typing up the CPU.

Blending nearby points. Up to M currently closest points are kept in a heap with the least close among them at the top. (A heap is a fundamental data structure in computer science that is used here as a priority queue [20].) Conceptually, we may think of a heap as tree structure in which a nontransitive relation holds between every parent and its pair of children (or child). In our case, the heap is implemented as a pointer array, and every parent must point at (a structure containing) a data point that is at least as far from the input as its children's data points:

$$\|\vec{h}_i - \vec{\imath}_t\| \geq \|\vec{h}_{2i} - \vec{\imath}_t\|, \quad \|\vec{h}_i - \vec{\imath}_t\| \geq \|\vec{h}_{2i+1} - \vec{\imath}_t\|$$

We insert data points into the heap until it is full and then replace the least close point at the top of the heap h_1 with the next data point $\vec{m}_t$ if $\vec{m}_t$ is closer to the input $\vec{\imath}_t$.[1] Each insertion or replacement requires

[1] The distance squared is used in all distance comparisons. There is no need to take the square root to get the actual distance since relative distances are enough to determine which point is closer to the input.

To find the M closest points in an input space of dimensionality K, this method requires N distance squared calculations (each involving K multiplications, K subtractions, and $K-1$

$\mathcal{O}(H)$ operations, where H is the size of the heap, since it may result in the promotion of other points in the heap that are further from the input than $\vec{p}_t$.[2]

While $\mathcal{E} \geq \epsilon_1$, the M points $(\vec{h_1}, \ldots \vec{h_M})$ that are nearest to the input $\vec{i}_t$ (in the input space) are collected in the heap and $\vec{o}_{t+1}$ is predicted by blending their output: $\forall k \in \{1, \ldots K\}$,

$$o_k \leftarrow \sum_{i=1}^{M} \omega_i \, (\vec{h}_i)_k$$

where $M \geq 3$. The weights in this affine combination determine the relative influence of each of the nearby data points and must sum to 1. Raw weights are first calculated using a Gaussian function: $\forall i \in \{1, \ldots M\}$,

$$r_i \leftarrow e^{-4 \|\vec{h}_i - \vec{i}_t\|^2 / \|\vec{h}_1 - \vec{i}_t\|^2} - e^{-4}$$

Each weight r_i is a Gaussian function of the distance from $\vec{h_i}$ to the input $\vec{i}_t$ as a proportion of the distance to the furthest point $\vec{h_1}$ among the M closest points. The function ranges from almost 1 at the input point $\vec{i}_t$ to 0 at $\vec{h_1}$. Thus, $\vec{h_1}$ has no influence but ensures that the function is continuous. Each raw weight is then divided by the sum of the raw weights so that the final weights sum to 1: $\forall i \in \{1 \ldots M\}$,

$$\omega_i \leftarrow \frac{r_i}{\sum_{i=1}^{M} r_i}$$

One benefit of this approach over other Gaussian blending approaches (e.g., [16]) is that the width of the Gaussian is scaled to the *local* sample density at the input point.[3]

Partitioning neuronal receptive fields. Often it is hard to determine the number of hidden units needed for a neural network to generalize adequately without overgeneralizing. This is especially true for on-line learning. It may not be possible to estimate the complexity of the mapping in advance. The number of available data points changes as memories of new experiences accumulate and as old memories become dated and are forgotten. To avoid smoothing out relevant detail or learning noise, the complexity of the model should match the complexity of the underlying function f. This places bounds on the optimum ratio of data points to synaptic weights. Partition nets set bounds this ratio during learning by a fixing the number of hidden units in a network to a small constant and setting a limit B (the *bucket size*) on the number of data points associated with that network. Since partition nets only memorize data points when they cannot be predicted accurately, a full bucket may indicate that more synaptic weights are needed or that learning has not yet converged. When the number of points exceeds the threshold B, the current input subspace and data points contained therein are partitioned into two halves. The original network is assigned to one half and a clone of it to the other.

There are many ways to partition the points in a vector space $\Re$ into roughly equal halves. Partition nets use one of the simplest methods. The partition plane must be perpendicular to one of the coordinate axes d. That dimension becomes the discriminator dimension. Only a single comparison is needed to determine whether an input point lies on the low side $\Re_L$ or high side $\Re_H$ of the partition plane p: $(\vec{i}_t)_d < p$. If, for example, the input point lies in the low subspace $\Re_L$ and $\Re_L$ has also been partitioned, the process may be repeated recursively.

Partition nets set the partition at the mean value of the dimension with highest mean absolute deviation:

$$mean_k = \frac{\sum_{i=1}^{B} (\vec{i}_i)_k}{B}$$

$$\max_k \sum_{i=1}^{B} |(\vec{i}_i)_k - mean_k|$$

The mean absolute deviation is usually more robust than the variance and, unlike the variance, requires no multiplications. This method requires on the order of B operations, where B is the size of the bucket.

Scaling. Since input values vary between 0 and 1, if the input space has been partitioned many times, a neural network will only receive a small range of input values. This will greatly slow learning. The input to the network should be scaled to match its shrunken receptive field. To do this, the network records the end points $[(e_L)_k, (e_H)_k]$ for each dimension k that delimit its subspace. For the original network, all end point values range from 0 to 1. When the input space is partitioned at p in discriminator dimension d, $(e_L)_d \leftarrow p$ for the high subspace network and $(e_H)_d \leftarrow p$ for its low subspace clone, so that $[(e_L)_d, (e_H)_d]_L$ is $[0, p]$ for the clone and $[(e_L)_d, (e_H)_d]_H$ is $[p, 1]$ for the original network. The input to each network is then scaled according to the end point values for its corresponding subspace: $\forall k \in \{1 \ldots K\}$,

$$s_k \leftarrow \frac{(\vec{i}_t)_k - (e_L)_k}{(e_H)_k - (e_L)_k}$$

---

additions), N comparisons to the furthest point $\vec{h_1}$ among the M current closest points, and $\mathcal{O}(R \log_2 M)$ operations for R insertions or replacements on a heap of size M.

The excessive sum optimization is used to eliminate many multiplications in calculating the distance squared. The the sum of the square of the differences is accumulated dimension by dimension. If the value ever exceeds the distance squared to $\vec{h_1}$, the point is excluded without calculating the squared differences for the remaining dimensions. For certain computers faster performance is obtained by avoiding the excessive sum optimization or modifying it to suit the hardware. The Pentium III instruction set, for example, allows four simultaneous floating point multiplications.

[2] See [20] for an explanation of big $\mathcal{O}$ notation.

[3] It should be noted that a $1D$ Gaussian function is used irrespective of the input dimensionality. Higher accuracy may be obtained by using the eigenvectors and eigenvalues of the covariance matrix for the M closest points to rotate and scale either the points themselves or a KD Gaussian used to blend them.

This ensures that all input values will again fall between 0 and 1.

Scaling, however, creates another problem. The weights of the low and high subspace network are no longer applicable. A correction is made by a linear scaling of the weights from the input unit for the discriminator dimension d to all the units of the first hidden layer. For the low subspace network, the adjustment is: $\forall j$,

$$w_{j,d} \leftarrow w_{j,d} \frac{p - (e_L)_k}{(e_H)_k - (e_L)_k}$$

Here and in the next two equations, $(e_L)_k$ and $(e_H)_k$ are the end points of the entire space *before* partitioning $[(e_L)_k, (e_H)_k]$. For the high subspace network, the adjustment is: $\forall j$,

$$w_{j,d} \leftarrow w_{j,d} \frac{(e_H)_k - p}{(e_H)_k - (e_L)_k}$$

It is also necessary to adjust the weights from the bias unit of the input layer to all the units of the hidden layer because the receptive field of the high subspace has been translated as well as scaled: $\forall j$,

$$w_{j,0} \leftarrow w_{j,0} + \frac{p - (e_L)_k}{(e_H)_k - (e_L)_k} w_{j,d}$$

Partition nets show how neural networks can dovetail with a closest point approach. So, for example, if network learning has sufficiently converged ($\mathcal{E}_A < \epsilon_1$) in one subspace $\Re_A$ but not in another $\Re_B$, network$_A$ is used to predict $\vec{o}_{t+1}$ when $\vec{i}_t \in \Re_A$ but Gaussian blending of M closest points in $\Re_B$ is used to predict $\vec{o}_{t+1}$ when $\vec{i}_t \in \Re_B$. Once learning convergence reaches its final target ($\mathcal{E}_A < \epsilon_3$), the points in $\Re_A$ are forgotten and learning cycles are spent in other subspaces where they are needed more.

The prediction algorithm. Multidimensional search trees [3, 21] provide an ideal data structure for hierarchically partitioning subspaces. Each node in the tree corresponds to a subspace. Internal nodes have a discriminator dimension d, partition p (which indicates where dimension d is partitioned), and a pair of pointers, one that point at the node's low child $\Re_L$ and the other that points at its high child $\Re_H$ (see [21]). Partition nets extend the KD tree structure so that, in addition to a pointer that points a subspace's bucket of points, each leaf node also a pointer that points at its associated neural network.

If for any input point $\vec{i}$ only the network or closest points in one leaf node's subspace are used for approximating the output, the function $g(\vec{i}_t, \Re)$ will be discontinuous at subspace boundaries. Regardless of whether the subspace is approximated by a neural network or Gaussian blending of M closest points, values at these boundaries will generally be less accurate than in the central area of the subspace because these values are extrapolated. Therefore, the output $\vec{o}$ near boundaries is calculated from a linear blending of the outputs of the two neighboring subspaces.

```
function g(i, ℜ) {
  if ( ℜ is a leaf node ) {
    if ( E < ε1 ) return network(i);
    else return gblend(i, (m1, ... mN));
  } else { /* ℜ is an internal node */
    dp ← i_d − p; /* distance to partition plane */
    blendzone_L ← κ width(ℜ_L, i, d);
    if ( dp ≤ −blendzone_L ) return g(i, ℜ_L);
    blendzone_H ← κ width(ℜ_H, i, d);
    if ( dp ≥ blendzone_H ) return g(i, ℜ_H);
    else { /* i is in the blend zone, so blend */
      o_L ← g(i, ℜ_L);
      o_H ← g(i, ℜ_H);
      ω ← (dp + blendzone_L) / (blendzone_H + blendzone_L);
      ∀k ∈ {1 ... K}, (o)_k ← (1 − ω)(o_L)_k + ω(o_H)_k;
      return o;
    }
  }
}
```

Listing 1: Pseudocode for function $g(\vec{i})$.

If $\Re$ is the subspace of a leaf node, $g(\vec{i}, \Re)$ returns an estimate of the perceived outcome state $\vec{o}_{t+1}$ (see Listing 1). The estimate of $\Re$'s neural network is returned if it is accurate enough ($\mathcal{E} < \epsilon_1$). Otherwise, the estimate is determined by gblend$(\vec{i}, (\vec{m}_1, \ldots \vec{m}_N))$, which collects in a heap the M closest points in $\Re$'s bucket and blends them using the Gaussian function, as explained above. If $\Re$ is the subspace of a leaf node, its distance to the partition plane dp is calculated. The function width, which may recursively call itself, will find $\vec{i}$'s leaf node subspace and return its width along the dimension d. So a blend zone is calculated which begins in the low subspace $\Re_L$ some fraction κ (e.g., 0.2) of $\Re$'s width below the partition plane and ends in the high subspace $\Re_H$ some fraction κ of $\Re$'s width above the partition plane (κ must fall in the range $[0, 1]$). If $\vec{i}$ is on the low side of the blend zone, no blending at this partition is required, and $g(\vec{i}, \Re_L)$ is called recursively for $\Re_L$. Likewise, if $\vec{i}$ is on the high side of the blend zone, $g(\vec{i}, \Re_H)$ is called recursively for $\Re_H$. Otherwise, it is necessary to obtain the results of both $g(\vec{i}, \Re_L)$ and $g(\vec{i}, \Re_H)$ and blend them. The blending is linear across the width of the blend zone (i.e., an affine combination with linear influence weights).

2.2 A note about complexity

Let us assume that neural network learning has converged sufficiently in each subspace so that only neural networks are used for prediction ($\mathcal{E}_i < (\epsilon_1)_i, \forall i \in \{1, \ldots S\}$, where S is the number of subspaces $\Re_1 \ldots \Re_N$). Let us further assume that the ratio of the total number of data points to the total number of weights is constant ($S = kW$). So if there are S subspaces, each subspace will have a network with W/S weights. If the KD tree is balanced, the depth will be $\log_2 S$. For best case complexity, there is no blend

zone ($\kappa = 0$). So there will be $\mathcal{O}(\log_2 S)$ comparisons to find the appropriate subspace, $\mathcal{O}(\log_2 S)$ calls to the $\mathcal{O}(\log_2 S)$ function `width`,[4] and $\mathcal{O}(W/S)$ synaptic calculations to predict $\vec{o}_{t+1}$. So the algorithm is $\mathcal{O}(W/S + \log_2^2 S)$, which compares favorably to using a single neural network $\mathcal{O}(W)$, since $W \geq S$.

However, accuracy demands a blend zone to avoid extrapolation. In the worse case the blend zone is always unavoidable (i.e., $\kappa = 1$, not a recommended value). Then calculations are performed on all synaptic weights. A constant number of comparisons and call to `width` must be made at every partition in the KD tree, so the complexity is $\mathcal{O}(W + S \log_2 S)$.

Let us consider intermediate values of κ. For a uniform distribution with $\kappa = 0.2$ (a more reasonable value), at each partition plane comparison, there are still a constant number of comparisons and calls to `width`. Additionally, there is a 80% chance of eliminating half the weights, comparisons, and calls to `width` of the levels below and a 20% chance of eliminating no weights, one comparison, and one call to `width`.

In the general case, the number of synaptic calculations is

$$\mathcal{O}\left(W\left(\frac{1+k}{2}\right)^{\log_2 S}\right) \qquad (1)$$

If S is the number of leaf nodes, $n = 2S - 1$ is the total number of nodes. The complexity for the number of comparisons and calls to `width` is described by the following recurrence relation:

$$T(n) = (1-\kappa)T(\lfloor\frac{n}{2}\rfloor) + \kappa\, T(n-1) + \mathcal{O}(\log_2 n)$$

So, regardless of input dimensionality, the number of synaptic weight calculations is given by equation (1), and for small values of κ, the average case performance for other operations is $\mathcal{O}(\log_2^2 S)$ but rising to $\mathcal{O}(S \log_2 S)$ for bounded S as $\kappa \to 1$. Average case complexity results generalize to the unbalanced KD trees that typically result from on-line learning, though the constants will be somewhat larger. Given the influence of constants, empirical comparisons are probably more useful for small to medium-sized problems than the above complexity analysis.

3 Applications

Partition nets can be used to learn many kinds of sensorimotor mappings, such as stereoscopic hand-eye mapping for humanoid robots [5]. One of our students, Nobuyuki Satoh, has applied this kind of mapping to a robot that plays ping-pong. The obvious advantage of a sensorimotor mapping is that the robot can intercept or grasp objects rapidly. Furthermore, if the robot's kinematics or camera geometry change, it can easily relearn the mapping.

Our current application is developing a sensorimotor mapping for a mobile robot Ψro [11]. The purpose of the robot is to recognize and respond to affordances. Affordances are the opportunities for interaction that objects afford. Ψro receives internal and external feedback when its exploits an affordance, and this feedback helps the robot learn to recognize affordances by developing predictions concerning how its motor signals transform sensor information from internal variables and external objects. The robots goal is to survive in a dynamic environment populated by edible and dangerous robots.

It is crucial for a robot or any creature to recognize *what* is *where*. But rather than impose our concept of object or location on the robot, Ψro learns to recognize its own affordances and to locate them in terms of its own internal sensorimotor mapping.[5] So partition nets are primarily meant to address where an object is in terms of the movements it takes to intercept or manipulate it. In Ψro's case, this was the 4D-to-2D mapping mentioned at the start of section 2.1.

Once Ψro has learned a sensorimotor mapping, it uses the mapping to plan paths to potential affordances. The method involves the use of heuristics for spatiotemporally projecting the learned mapping in a quantized phase space. (Interested readers may refer to the tutorial at http://robotics.me.es.osaka-u.ac.jp/~kfm.) A method related to dynamic programming is used to avoid recomputing subpaths between cells in the quantized space. Beginning from Ψro's starting (or finishing) position, the robot attempts actions "mentally." Ψro repeats this process among those hypercubes it has examined according to their priority.

A good heuristic gives highest priority to hypercubes that are closest to the start in terms of the time it take to reach them and closest to the goal in terms of distance. A path is calculated faster by giving a higher weighting to the distance term in the priority function. However, the path will begin to resemble one calculated by local hill climbing. As the priority sifts back toward favoring time and distance equally, the path becomes smoother and approaches an optimal solution (for a given coarseness of quantization), even though its calculation takes only a fraction of the time as when using a priority function based solely on time.

4 Conclusion

Partition networks are able to simulate the sensorimotor aspect of five key features of vertebrate learning (points 1–5 identified in section 1.2). They can also be integrated into a higher level attentional and planning system (point 6). They learn when necessary to correct errors or fill in gaps in a sensorimotor

[4] If $\kappa = 0$, of course, there is no point in calling `width`.

[5] In brief, the *what* system uses wavelet analysis to highlight variations in image sequences of tracked objects [10]. Feedback concerning how interactions transform internal variables (in the presence of segmented encodings of objects) elicits the formation of categorical representations. Categorical representations [7] are derived from sensorimotor invariance that is indicative of the same kind of affordance.

mapping. Once learning has converged, they are able to adapt quickly to bodily or environmental change (point 7). This is because *(a)* new memories of unexpected events can be used right away, before the relevant neural networks converge; *(b)* a relevant network's empty subspace will fill with new memories of the unexpected events, and it will begin learning them immediately, and *(c)* if the old mapping is not radically different from the new mapping, the network's old synaptic weights will help to speed convergence. Other advantages of partition nets are summarized at the top of section 2.

Acknowledgments

I gratefully acknowledge the kind assistance of my students, Yoji Miyazaki and Koji Tatani, and support from a research grant from CREST of JST.

References

[1] Barron, A.R. (1992). Neural net approximation. In *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems*, pp. 69-72. New Haven, CT: Yale University.

[2] Barron, A.R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions of Information Theory, 39,* pp. 930-945.

[3] Bentley, J.L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the* ACM, *18*(9), 509-517.

[4] Bullock, D. & Grossberg, S. (1988). Neural dynamics of planned arm movements. Emergent invariants and speed-accuracy properties during trajectory formation. *Psychological Review, 95*, 49-90.

[5] Clocksin, W.F. & Moore, A.W. (1989). Experiments in adaptive state-space robotics. In *Proceedings of the Seventh Conference of the Society for Artificial Intelligence and Simulation of Behaviour,* pp. 115-125.

[6] Gross, E.M. & Wagner, D. (1996). *K*D trees and Delaunay-based linear interpolation for function learning: A comparison to neural networks with error backpropagation. *IEEE Transactions on Control Systems Technology, 4*(6), 649-653.

[7] Harnad, S. (1987). Category induction and representation. In S. Harnad (Ed.), *Categorical perception: The groundwork of cognition.* Cambridge: Cambridge University Press.

[8] Jeannerod, M. (1990). The representation of the goal of an action and its role in the control of goal-directed movements. In E.L. Schwartz (Ed.), *Computational neuroscience.* Cambridge, MA: MIT Press, pp. 352-368.

[9] Jeannerod, M. (1994). The represented brain: Neural correlates of motor intention and imagery. *Behavioral and Brain Sciences, 17*(2), pp. 187-202.

[10] MacDorman, K.F. (1997). *Symbol grounding: Learning categorical and sensorimotor predictions for coordination in autonomous robots.* Technical Report No. 423. Computer Laboratory, Cambridge (e-mail librarian@cl.cam.ac.uk for a copy).

[11] MacDorman, K.F. (1999). Grounding symbols through sensorimotor integration. *Journal of the Robotics Society of Japan, 17*(1), 20-24.

[12] Mel, B.W. (1988). Building and using mental models in a sensory-motor domain: A connectionist approach. In *Proceedings of the Fifth International Conference on Machine Learning*, 207-213.

[13] Mel, B.W. (1990). *Connectionist robot motion planning: A neurally-inspired approach to visually-guided reaching.* Boston: Academic Press.

[14] Omohundro, S.M. (1987). *Efficient algorithms with neural network behavior.* Technical Report No. UIUCDCS-R-1331. Department of Computer Science, University of Illinois at Urbana Champaign.

[15] Omohundro, S.M. (1990). Geometric learning algorithms. *Physica D, 42*(1-3), 307-321.

[16] Omohundro, S.M. (1991). Bumptrees for efficient function, constraint, and classification learning. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, *Advances in Neural Information Processing Systems 3.* San Mateo, CA: Morgan Kaufmann.

[17] Paul, R.P. (1981). *Robot manipulators, mathematics, programming and control.* Cambridge, MA: *MIT* Press.

[18] Prablanc, C., Echallier, J.F., Komilis, E. & Jeannerod, M. (1979). Optimal response of eye and hand motor systems in pointing at a visual target. I. Spatio-temporal characteristics of eye and hand movements and their relationships when varying the amount of visual information. *Biological Cybernetics, 35*, 113-124.

[19] Rummelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning internal representation by error propagation. In D.E. Rummelhart and J.L. McCleland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition*, Vol. 1, Ch. 8. Cambridge, MA: MIT Press.

[20] Sidgwick, R. (1998). *Algorithms in C.* Reading, MA: Addison-Wesley.

[21] Sproull, R.F. (1991). Refinements to nearest-neighbor searching in *K*D trees. *Algorithmica, 6*(4), 579-589.