Connectionist Architectures: Optimization

Advanced article

Marcus Frean, Victoria University of Wellington, Wellington, New Zealand

CONTENTS

A key issue in using connectionist methods is the choice of which network architecture to use. There are a number of ways this choice can be made automatically, driven by the problem at hand.

INTRODUCTION

If one takes a training set in the form of input–output pairs and trains a large connectionist network on it, the result is generally 'overfitting'. There are many functions which exactly fit the existing data and the act of learning arrives at just one of them, somewhat arbitrarily. The problem is compounded where the data is noisy, in which case the network uses its extra degrees of freedom to fit the noise rather than the underlying function generating the data. Conversely, if the network is too small it 'underfits', which is equally unsatisfactory. The real aim is usually not to get the training set correct, but to generalize successfully to new data. The model selection problem is to arrive at the network that gives the best possible predictions on new inputs, using only the available training data and prior knowledge about the task. (*See* **Machine Learning**)

There are several ways of controlling the complexity of mappings learned by neural networks. These include varying the number of weights or hidden units by building up or paring down an existing network, and direct penalties (otherwise known as regularization) on model complexity, such as weight decay. Other ideas include partitioning the input space into regions which are locally linear as in 'mixtures of experts', or using genetic algorithms to choose between different architectures.

CRITERIA FOR NETWORK OPTIMALITY

The optimality or otherwise of a network is, in many cases, determined by its ability to generalize. Almost by definition this ability is not directly observable, so in practice we have to make an educated guess at it and use that to choose between networks.

The simplest method takes part of the available data and sets it aside. Once the network has been trained on the remaining data it can be 'validated' by seeing how well it performs on the withheld data, thus giving an estimate of how well it will generalize. This estimate won't be very good unless the hold-out set is large, which wastes a lot of the data that could otherwise be used for training. To minimize this effect, 'cross-validation' applies the same idea repeatedly with different subsets of the data, retraining the network each time. In k-fold cross-validation, for example, the data is divided into k subsets. One at a time, these serve as hold-out sets, and the validation performance is then averaged across them to give an estimate of how well the network generalizes. 'Leave-one-out' cross-validation uses $k = N$, the number of samples, but $k = 10$ is typically used.

Another general approach, from conventional statistics, is to attempt to quantify the generalization performance of trained networks without a validation set at all. One prefers networks with a low 'prediction error'

$$C = C_{data} + C_{net} \quad (1)$$

Here C_{data} is the usual training error, such as the sum of squared errors, and C_{net} is taken to be a measure of the complexity of the network,

proportional to the effective number of free parameters it has. Assuming a nonlinear network is locally linear in the region of the minimum, an approximation to C_{net} can be calculated (Moody, 1992; Murata *et al*., 1994) given the Hessian matrix of second derivatives $H_{ij} = \partial^2 C_{data}/\partial w_i \partial w_i$, which can be found using a number of methods (Buntine and Weigend, 1994). For large training sets, leave-one-out cross-validation and the above are essentially equivalent, with the latter giving the same effect for much less computational effort. These approaches assume a single minimum however, so the estimate can be strongly affected by local minima. On the other hand, leave-one-out cross-validation also gets trapped in local minima, in which case 10-fold cross-validation is preferable.

A third approach is to use Bayesian model comparison to choose between networks, as well as to set other parameters such as the amount of weight decay. Bayesians represent uncertainty of any kind by an initial or 'prior' probability distribution, and use the laws of probability to update this to a 'posterior' distribution in the light of the training set. In this view we should choose between models based on their posterior probabilities given the available data – again this does not require that any data be set aside for validation (MacKay, 1995). In the fully Bayesian approach, ideally we should use not one set of weights and one structure but many sets and many architectures, weighting the prediction of each by their posterior probability. To the extent this averaging can be done, deciding on an 'optimal' model (and indeed all learning as it is usually thought of) becomes unnecessary. (*See* **Reasoning under Uncertainty**; **Pattern Recognition, Statistical**)

Generalization performance is not the only measure of usefulness of a given network architecture. Other potentially important measures are its fault tolerance, training time on the problem at hand, robustness to 'catastrophic forgetting', ease of silicon implementation, speed of processing once trained, and the extent to which hidden representations can be interpreted. (*See* **Catastrophic Forgetting in Connectionist Networks**)

PRUNING OF UNIMPORTANT CONNECTIONS OR UNITS

Pruning algorithms start by training an overly complex network before trimming it back to size. In other words, we knowingly overfit the data and then reduce the number of free parameters, attempting to stop at just the right point. Clearly the general model selection schemes described above (cross-validation, estimated prediction error, and Bayesian model comparison) can be applied to prune overly large networks; however a number of ideas have been formulated that are specific to pruning. Pruning algorithms can remove weights or whole units, and one can think of the choice of which element to remove as being driven by a measure of 'saliency' for that element. Each algorithm uses a different form for this saliency.

A simple measure of saliency to use for weights is their absolute value. However, while it may be true that removing the smallest weight affects the network the least, it doesn't follow that this is the best weight to remove to improve generalization. Indeed this seems completely opposite to weight decay (see below), which in effect 'removes' large weights by decaying them the most, and it performs poorly in practice.

A more principled idea, known as 'optimal brain damage' (Le Cun *et al*., 1990), approximates the change to the error function that would be caused by removal of a given connection or unit, and uses this measure to decide which to remove. To make this approximation, one trains the network until it is at a minimum of the usual error function, and then calculates the Hessian H. Ignoring the off-diagonal elements of this matrix, the saliency of the weight is given by $H_{ii}w_i^2$.

This idea has been further developed in 'optimal brain surgeon' (Hassibi and Stork, 1993), which avoids the assumption that the Hessian is diagonal. Interestingly this gives a rule for changing all the weights, with the constraint that one of these involves the setting of a weight to zero. One must first calculate the full inverse Hessian matrix however, which can make the algorithm slow and memory intensive for large problems.

Statistical tests can also be applied to detect non-contributing units that could be made redundant. For example if two units are in the same layer and are perfectly correlated (or anti-correlated) in their activity, we know the network can perform the same mapping with one of them removed. A particularly simple case is when a unit has the same output all the time, making it functionally no different from the bias unit.

WEIGHT DECAY

In networks whose output varies smoothly with their input, small weights give rise to outputs which change slowly with the input to the net, while large weights can give rise to more abrupt changes of the kind seen in overfitting. For this reason one response to overfitting is to penalize

the network for having large weights. The most obvious way to do this is by adding a new term

$$C_{net} = \frac{\beta}{2}\sum_i w_i^2 \quad 0 < \beta < 1 \tag{2}$$

to the objective function being minimized during learning. The total cost $C = C_{data} + C_{net}$ can then be minimized by gradient descent. For a particular weight we have

$$\Delta w \propto -\frac{\partial C}{\partial w} = -\frac{\partial C_{data}}{\partial w} - \beta w \tag{3}$$

The first term leads to a learning rule such as back propagation (depending on the form of C_{data}), while the second removes a fixed proportion of the weight's current value. Hence each weight has a tendency to decay toward zero during training, unless pulled away from zero by the training data. The 'decay rate' β determines how strong this tendency is. Clearly a major question is how to set β, for which the general methods described earlier are applicable. (*See* **Backpropagation**)

Weight decay helps learning in other ways as well as its effect on generalization – it reduces the number of local minima, and makes the objective function more nearly quadratic so quasi-Newton and conjugate gradient methods work better.

From a Bayesian perspective, weight decay amounts to finding a *maximum a posteriori* (MAP) estimate given a Gaussian prior over the weights, reflecting our belief that the weights should not be too large. Weight decay is not usually applied to bias weights, reflecting the intuition that we have no *a priori* reason to suppose the bias offset should be small. Depending on the nature of the problem, this may not be a particularly sensible prior – for instance we may actually believe that most weights should be zero but that some should be substantially nonzero. One expression of this to use a different weight cost such as

$$C_{net} = \beta \sum_i \frac{w_i^2}{c^2 + w_i^2} \tag{4}$$

This has been called weight elimination, because it is more likely to drive weights towards zero than simple weight decay. Very small weights can then be eliminated. c is a second parameter which needs to be set by hand. An interesting alternative is 'soft weight sharing' (Nowlan and Hinton, 1992) which implements MAP with a prior that is a mixture of Gaussians. The means (which need not be zero) and variances of these Gaussians can be adapted by the learning algorithm as training proceeds.

GENERATIVE ARCHITECTURES

Generative architectures, also called constructive algorithms, build networks from scratch to suit the problem at hand. Once each unit is trained, its weights are 'frozen' before building the next unit. An important advantage of this is that only single layers of weights are being trained at any one time. Accordingly the learning rules involved need only be local to the unit in question (unlike back propagation), which tends to make learning particularly fast and straightforward.

For simplicity each algorithm is described here as it applies to a single output unit – multiple outputs are trivial extensions to this, as described in the cited papers.

Upstarts

The upstart algorithm (Frean, 1990) is a method for constructing a network of threshold units. Imagine a single linear threshold unit (perceptron) that is trained to minimize the number of errors it makes on the training set, and then frozen. This unit, which we will call u, makes two kinds of error: it is either wrongly on, or wrongly off. In the upstart algorithm these errors are dealt with separately by recruiting two new units, which we could call u_- and u_+, one for each type of error. These new units receive the same inputs, but their outputs go directly to u (see Figure 1(a)). The role of u_- is to correct the 'wrongly on' errors by the parent unit u so it has a large negative output weight, while the output weight of u_+ is large and positive since its function is to correct the wrongly off errors. (*See* **Perceptron**)

It is easy to derive appropriate targets for u_-, given the original targets and u's responses: u_- is to be given the target 1 whenever u is wrongly on, while in all other cases its target should be 0. Similarly the target for u_+ is 1 whenever u is wrongly off, and 0 otherwise. Notice that the output of u_- (u_+) does not matter if u was already correctly off (on), so these can be omitted from the child node's training set. Should the child units be free of errors on their respective training sets, u will itself be error-free. If, however, either u_- or u_+ still make mistakes of their own, these errors are likewise of two types and we can apply the same idea, recursively. The result is a binary tree of units, grown 'backwards' from the original output unit. Child nodes spend their time loudly correcting their parent's mistakes, hence the algorithm's name.

Suppose u_- has the output 1 for just one of the wrongly on patterns of u, and is zero in all other

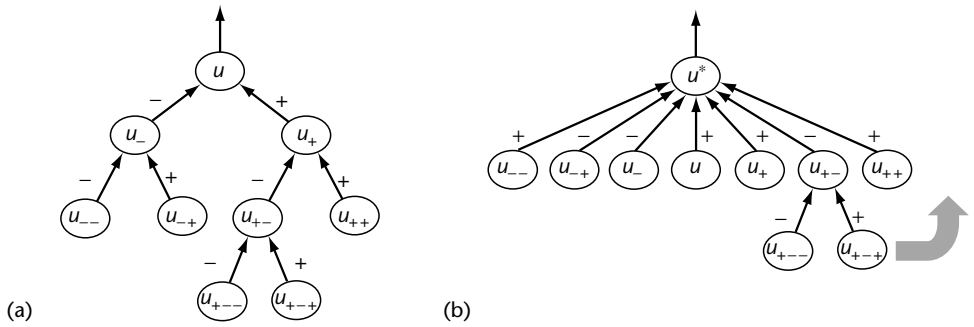

Figure 1. (a) A binary tree constructed by the upstart algorithm. All units have direct inputs, omitted here for clarity. The 'leaves' of the tree make no errors, so neither does the root node. (b) The same network being rearranged into a single hidden layer.

cases. For convex training sets (e.g. binary patterns) it is always possible to 'slice off' one pattern from the others with a hyperplane, so in this case it is easy for u_- to improve u by at least one pattern. Of course we hope that u_- and u_+ will confer much more advantage than this in the course of training. In practice a quick check is made that the number of errors by u_- is in fact lower than the number of wrongly on errors by u, to ensure convergence to zero errors.

Networks constructed using this method can be reorganized into a single hidden layer, if desired. That is, a new output unit can get zero errors by being connected to this layer with weights which are easily found, as shown in Figure 1(b).

For noise-free data this procedure usually produces networks that are close to the smallest that can fit the data, with attendant gains in generalization ability compared to larger networks. Notice however that the training set is learned without errors, so this is just as prone as any other algorithm to overfitting of noisy data (the idea has not been generalized to handle such noise, although there seems no reason why this couldn't be done). One can also use the same procedure to add hidden units to a binary attractor (Hopfield) network, thereby increasing its memory capacity from $\sim N$ to 2^N patterns.

The Pyramid Algorithm

Another algorithm for binary outputs is the pyramid algorithm (Gallant, 1993). One begins as before with a single binary unit, connected to the inputs and trained to minimize the number of errors. This unit is then 'frozen' and (assuming errors are still being made) a new unit is designated the output: this new unit sees both the regular inputs and any (frozen) predecessors as its input, as shown in Figure 2. (*See* **Perceptron**)

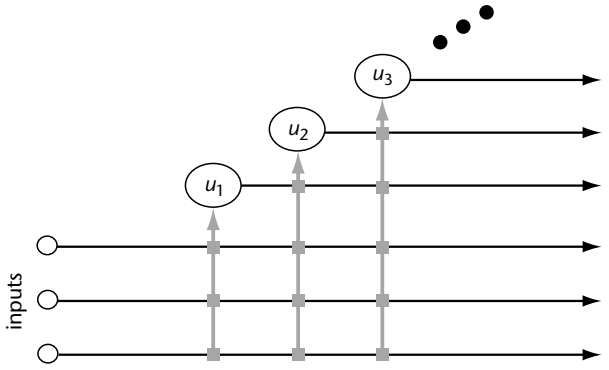

Figure 2. The architecture constructed by the pyramid algorithm. Vertical lines represent multiple connections (shown as squares) to the unit above. Each new unit assumes the role of output.

It is not hard to show that this new unit can achieve fewer errors than its predecessor, provided the input patterns are convex. If it sets its weights from the network inputs to zero and has a positive weight from the previous frozen unit, these two obviously make the same number of errors. As with upstarts, given convex inputs it could then easily reset its input weights so that this behavior was altered for just one input pattern where it was previously in error. This is the 'worst case' behavior, and appropriate weights can easily be predefined, to be improved by training (any method for arriving at good weights for a single unit is applicable). Despite its apparently 'greedy' approach to optimization and its extreme simplicity, the method can build concise networks. For example, given the N-bit parity problem (where the task is to output the parity of a binary input) the upstart algorithm generates a network with N hidden units, while the pyramid algorithm builds the apparently minimal network having only $(N+1)/2$

hidden units. Like the upstart algorithm, the method as it stands is prone to overfitting noisy data.

Cascade Correlation

Cascade correlation (Fahlman and Lebiere, 1990) can be applied to networks with real-valued outputs, and uses sigmoidal hidden units. We begin with a network having only direct connections between inputs and outputs, with no hidden units. These weights are trained using gradient ascent (the delta rule), or whatever learning procedure you like. We then introduce a hidden unit, with connections to the input layer. This unit sends its output via new weighted connections to the original output layer. In upstarts, the hidden unit is binary and is preassigned one of two roles, which determines how it is trained and the sign of its output weight – this is because being binary it can only correct errors of one type by the output unit, given its output weight. In this case, however, the hidden unit is real-valued and this means it can play a role in correcting errors of either sign by the output unit. Fahlman and Lebiere's idea is to train the hidden unit to maximize the covariance between its output and the existing errors by the output units. We can then use this gradient to learn input weights for the hidden unit in the usual way. When this phase of learning is deemed to have finished, all the output unit's connections are retrained, including the new ones. The process can now be repeated with a new hidden unit, with each such unit receiving inputs from the original inputs as well as all previous hidden units. Figure 3 shows the resulting cascade architecture.

A potential problem is that the output can make a lot of errors yet, after averaging, the correlation with a hidden unit can be very small. Despite this the method seems to work well in practice, and can be extended to recurrent networks.

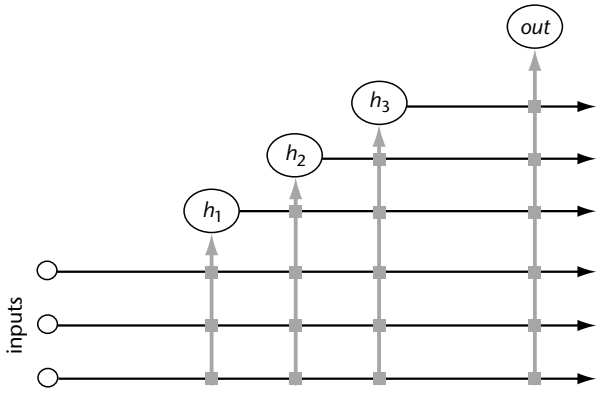

Figure 3. The architecture constructed by cascade correlation.

ADAPTIVE MIXTURES OF EXPERTS

In conventional back propagation networks, each sigmoid unit potentially plays a part in the network's output over its entire range of inputs. One way of restricting the power of the network is to partition the input space into distinct regions, and restrict the influence of a given unit to a particular region. Ideally we would like to learn this partition rather than assume it from the beginning.

A particularly appealing way to do this is known as the 'mixture of experts' architecture (Jacobs *et al.*, 1991). Each 'expert' consists of a standard feedforward neural network. A separate 'gating' network, with as many outputs as there are experts, is used to choose between them. The output of this network is chosen stochastically using the softmax activation function at its output layer,

$$Pr(i = 1) = \frac{e^{\phi_i}}{\sum_j e^{\phi_j}} \tag{5}$$

where ϕ_i is the weighted sum into the ith output unit. This reflects the fact that only one of the outputs is active at any given time. All of the nets (including the switch) are connected to the same inputs as shown in Figure 4, but only the expert that happens to be chosen by the switch is allowed to produce the output. A learning procedure can be found by maximizing the log likelihood of the network generating the training outputs given the inputs, in the same way as the cost function for back propagation is derived. Indeed the learning rule for the 'experts' turns out to be simply a weighted version of back propagation. During learning each expert network gets better at generating correct outputs for the input patterns that are

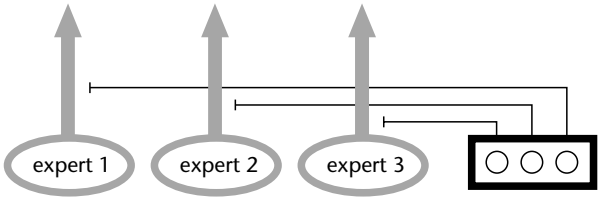

Figure 4. The mixtures of experts architecture. Each expert consists of a separate network, and may have multiple outputs. A separate gating network acts as a switch, allowing just one of the experts to generate the output. All of the experts, together with the gating network, have access to the input pattern.

assigned to it by the switch. At the same time, the switch itself learns to apportion inputs to the best experts. One can think of the switch as performing a 'soft' partition of the input space into sections which are learnable by individual experts. (*See* **Backpropagation**)

A further possibility is to treat each expert as a mixture of experts system itself (Jordan and Jacobs, 1994). A form of hierarchical decomposition of the task can thus be repeated for as many levels as desired. If simple linear units are used for the leaf nodes of the resulting tree-structured network, training can be achieved using a version of the expectation-maximization (EM) algorithm rather than gradient descent.

USING GENETIC ALGORITHMS TO EVOLVE CONNECTIONIST ARCHITECTURES

One criticism of both pruning and constructive algorithms is that they alter networks in only very limited ways, and as such they are prone to getting stuck in local optima in the space of possible architectures. Genetic algorithms offer a richer variety of change operators in the form of mutation and crossover between encodings (called 'chromosomes') of parent individuals in a population. The hope is that networks which are more nearly optimal may be found by evolving such a population of candidate structures, compared to making limited incremental changes to a single architecture. (*See* **Evolutionary Algorithms**)

In generating new candidate architectures, genetic algorithms choose parents based on their performance ('fitness'), which may be evaluated using the techniques described previously for determining network optimality. The main contribution of genetic algorithms then is their more general change operators, principally that of crossover, which operate on the chromosome rather than the network directly. Accordingly, the way in which architectures are mapped to chromosomes and vice versa is of central importance (Yao, 1999).

One approach is to assume an upper limit N to the total number of units and consider an $N \times N$ connectivity matrix, whose binary entries specify the presence or absence of a connection. Any units without outputs are effectively discarded, as are those lacking inputs. A population of such matrices can then be evolved, by training each such network using a learning algorithm initialized with random weights. To apply genetic operators, each matrix is simply converted to a vector by concatenating its rows. Restriction to feedforward networks is straightforward: matrix elements on and below the diagonal are set to zero, and are left out of the concatenation.

A drawback of this approach (though by no means unique to it) is that the evaluation of a given network is very noisy, essentially because the architecture is not evaluated on its own but in conjunction with its random initial weights. Averaging over many such initializations is computationally expensive, and one solution is to evolve both the connections and their values together. In this case an individual consists of a fully specified architecture together with the values of weights. On the other hand cross-over makes little sense for combining such specifications (unless the neural network uses localist units such as radial basis functions) because it destroys distributed representations.

Less direct encodings can be used, such as rules for generating networks, rather than the networks themselves. Evolutionary algorithms have also been used to change the transfer functions used by units (such as choosing between sigmoid and Gaussian for each unit), and even to adapt the learning rules used to set the weights.

References

Buntine WL and Weigend AS (1994) Computing second derivatives in feedforward networks: a review. *IEEE Transactions on Neural Networks* **5**(3): 480–488.

Fahlman SE and Lebiere C (1990) The cascade correlation learning architecture. In: Touretzky DS (ed.) *Advances in Neural Information Processing Systems*, vol. II, pp. 524–532. San Mateo, CA: Morgan Kaufmann.

Frean M (1990) The upstart algorithm: a method for constructing and training feedforward neural networks. *Neural Computation* **2**(2): 198–209.

Gallant SI (1993) Neural network learning and expert systems. Cambridge, MA: MIT Press.

Hassibi B and Stork DG (1993) Second-order derivatives for network pruning: optimal brain surgeon. In: Hanson SJ, Cowan JD and Giles CL (eds) *Advances in Neural Information Processing Systems*, vol. V, pp. 164–171. San Mateo, CA: Morgan Kaufmann.

Jacobs RA, Jordan MI, Nowlan SJ and Hinton GE (1991) Adaptive mixtures of local experts. *Neural Computation* **3**(1): 79–87.

Jordan MI and Jacobs RA (1994) Hierarchical mixtures of experts and the EM algorithm. *Neural Computation* **6**(2): 181–214.

Le Cun Y, Denker JS and Solla SA (1990) Optimal brain damage. In: Touretzky DS (ed.) *Advances in Neural Information Processing Systems*, vol. II, pp. 598–605. San Mateo, CA: Morgan Kaufmann.

Moody JE (1992) The effective number of parameters: an analysis of generalization and regularization in nonlinear learning systems. In: Moody JE, Hanson SJ and Lippmann RP (eds) *Advances in Neural Information Processing Systems*, vol. IV, pp. 847–854. San Mateo, CA: Morgan Kaufmann.
Murata N, Yoshizawa S and Amari S (1994) Network information criterion – determining the number of hidden units for artificial neural network models. *IEEE Transactions on Neural Networks* **5**: 865–872.
Nowlan SJ and Hinton GE (1992) Simplifying neural networks by soft weight sharing. *Neural Computation* **4**(4): 473–493.
Yao X (1999) Evolving artificial neural networks. *Proceedings of the IEEE* **87**(9): 1423–1447.

Further Reading

Bishop C (1995) *Neural Networks for Pattern Recognition*. Oxford: Clarendon Press.
Neal R (1996) *Bayesian Learning for Neural Networks*. New York: Springer-Verlag.
Read RD and Marks RJ (1999) *Neural Smithing – Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA: MIT Press.

Connectionist Implementationalism and Hybrid Systems

Intermediate article

Ron Sun, University of Missouri, Columbia, Missouri, USA

CONTENTS

We may incorporate symbolic processing capabilities in connectionist models, including implementing such capabilities in conventional connectionist models and/or adding additional mechanisms to connectionist models.

INTRODUCTION

Many cognitive models have incorporated both symbolic and connectionist processing in one architecture, apparently going against the conventional wisdom of seeking uniformity and parsimony of mechanisms. It has been argued by many that hybrid connectionist–symbolic systems constitute a promising approach to developing more robust and powerful systems for modeling cognitive processes and for building practical intelligent systems. Interest in hybrid models has been slowly but steadily growing. Some important techniques have been proposed and developed. Several important events have brought to light ideas, issues, trends, controversies, and syntheses in this area. In this article, we will undertake a brief examination of this area, including rationales for such models and different ways of constructing them.

MODELING DIFFERENT COGNITIVE PROCESSES WITH DIFFERENT FORMALISMS

The basic rationale for research on hybrid systems can be succinctly summarized as 'using the right tool for the right job'. More specifically, we observe that cognitive processes are not homogeneous: a wide variety of representations and processes seem to be employed, playing different roles and serving different purposes. Some cognitive processes and representations are best captured by symbolic models, others by connectionist