Sequentially Learning Multiple Meaningful Representations in Static Neural Networks

Avoiding Catastrophic Interference in Multi-layer Perceptrons

Bachelor thesis
of
Jordi Bieger
0416371
jbieger@gmail.com

Supervisors:
Ida Sprinkhuizen-Kuyper
Iris van Rooij

April 2, 2009

Contents

Sequentially Learning Multiple Meaningful Representations in Static Neural Networks

Avoiding Catastrophic Interference in Multi-layer Perceptrons

Jordi Bieger

April 2, 2009

Abstract

Artificial neural networks (ANNs) attempt to mimic human neural networks in order to solve problems and carry out tasks. However, in contrast to their human counterparts ANNs cannot generally learn to perform new tasks without forgetting everything they already know due to a phenomenon called *catastrophic interference*. This paper discusses this phenomenon, shows that it occurs in multi-layer perceptrons with arbitrary task representations and proposes and discusses the *static meaningful representation learning* method that uses meaningful task representations to circumvent this problem when learning to perform multiple tasks. The technique is powerful enough to enable the learning of several simple tasks without changing the weights of the network. It remains to be seen whether the technique scales to more interesting task domains. The real potential of using meaningful task representations lies in their combination with other techniques.

1 Introduction

For decades researchers have been trying to recreate intelligence in computers. One important method of doing this is to imitate what we know about human and animal intelligence. Our brains are networks of interconnected neurons that apparently allow us to think, learn, act and perceive as we do. Inspired by this knowledge, *artificial neural networks (ANNs)* were created that are able to learn to match patterns, perform tasks and solve problems [58, 60]. This paper focusses on a subclass of ANNs called *multi-layer perceptrons (MLPs)*. Using *supervised learning* MLPs can be taught how to perform a task by merely showing them examples of inputs associated with desired outputs called *targets*. After successfully training on these examples, MLPs are not only able to recall the correct output for any of the example inputs, but can usually also generalize what was learned fairly well to previously unseen input patterns. These are abilities that we attribute to humans and other intelligent animals as well.

However, while humans can learn to perform new tasks without forgetting old ones, this is not usually the case for artificial neural networks. When a network capable of performing one task is trained on another, it will in general forget everything it ever knew about the first task very quickly. This phenomenon is called "catastrophic interference" [23] and causes problems for both the efficiency of ANNs as well as their psychological plausibility as models of the mind.

When asked to perform a task, people require a description of that task in order to know what to do exactly. This is done using so-called *action words*, which are represented in the brain in

a way that is relevant to the task that they describe [29]. This paper presents a method called *static meaningful representation learning (SMRL)* for determining task representations that are meaningful in the context of existing knowledge without changing that knowledge. This enables ANNs to sequentially learn new tasks without suffering from catastrophic interference.

The remainder of this paper is organized as follows: in Section 2 I will give a description of the kind of ANNs used in this paper, describe how real world things may be represented in a manner that ANNs understand, shortly talk about how the goal of this paper is describe what catastrophic interference is exactly and explain why it occurs in these networks. in Section 2 I will provide the essential background information that is needed in order to understand the ideas presented in the rest of the paper. Section 2.1 will give a description of the kind of ANNs used in this paper and Section 2.2 talks about how real world things may be represented in a manner that ANNs understand. Technical details of the used ANNs are described in Appendix A. Section 2.3 gives a short introduction to the issues related to learning multiple tasks in an ANN and is followed by Section 2.4 and Section 2.5 that talk about the related areas of *catastrophic interference* and *transfer learning*. The SMRL method for learning multiple tasks without suffering from catastrophic interference is defined in Section 3. Section 4 describes the experiments carried out to show that catastrophic interference is indeed a big problem in regular MLPs and investigates how SMRL can best be used to prevent that problem. The results will be presented and analyzed in the subsections after each experiment and Appendix B contains a complete overview of the results obtained for the experiments with SMRL. Section 5 will provide a final analysis and evaluation of the results and is followed by Section 6 which contains ideas for future research. Section 7 summarizes the paper and concludes that using SMRL has a lot of potential for avoiding catastrophic interference, making ANNs more psychologically plausible and that the technique should be researched further, especially in combination with other techniques.

2 Background

This section will first describe how the artificial neural networks used in this paper work. The algoritms in this subsection are defined in more detail in A. Next, it is explained how ANNs might be used to perform real world tasks by talking about how things in the real world can be represented in ways that ANNs understand. This subsection is followed by a short introduction to multiple task learning with ANNs. After that, it is explained exactly what catastrophic interference is, why it occurs in ANNs and why it is a problem. Finally the related area of transfer learning is reviewed.

2.1 Artificial neural networks

Artificial neural networks are networks of *neurons* or *nodes*. Each of these nodes has a certain activation value that can either be set, or "clamped", by the user (input nodes) or determined by the nodes in the rest of the network (output and hidden nodes). When an ANN is used to perform some task, the user clamps the activation values of the input units. This activation then spreads throughout the network, eventually also affecting the output nodes' activations. The activation values of these output nodes encode the ANN's solution to the presented task. The activation of non-input nodes is determined by the activation of other nodes that they are connected to.

Connections Nodes are connected to each other with directed and weighted connections or "synapses". Both these connections and their strengths are often referred to as *weights*. These weights determine how a network reacts to inputs from the environment and it is therefore said that they encode the network's knowledge [39]. This paper uses a subclass of ANNs called *multi-layer perceptrons (MLPs)* which are organized into different layers of nodes (L) (see Figure 1). Activation flows from an input layer (L_1), further downstream through any number of hidden layers ($L_2 \dots L_{|L|-1}$) until it finally reaches the output layer ($L_{|L|}$), where $|L|$ is the total number of layers. The nodes in the hidden layers are usually not directly used by the user of the ANN, but they allow the network to transform the input into something it 'understands' better. Most tasks are fundamentally impossible to learn for a neural network without hidden nodes [42] (e.g. `XOR` and `IFF` in Section 4). All of the used networks are non-recurrent, which means that a connection may only go to a node in a layer that is further downstream. In a transfer of activation between two nodes, the activation providing, upstream node is called the "presynaptic neuron" and the receiving, downstream node is called the "postsynaptic neuron". Usually there are connections from every node in one layer L_i to every node in the next layer L_{i+1}.

Activation flow When the input nodes are clamped with values, their activation spreads throughout the network to the output nodes. Each non-input node receives the sum of the activations of the nodes in previous layers scaled by the strength of the connection from that node into this one as its *net input*. To this sum is added the node's *bias*. For all intents and purposes, the bias can be thought of as the strength of the incoming connection from some imaginary node

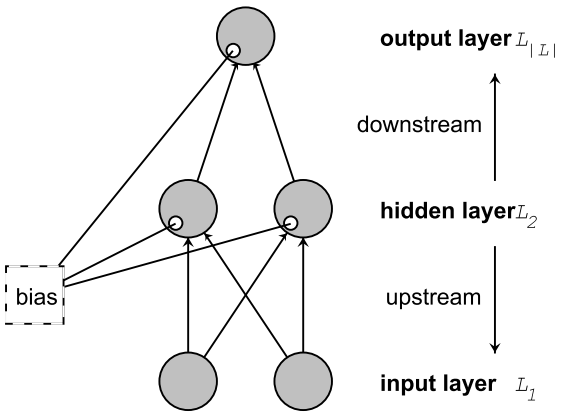


Figure 1: This multi-layer perceptron has two input nodes, two hidden nodes and one output node. All non-input nodes have a bias value, which can be thought of as the weight between the node and an imaginary bias node that always has activation 1. In the future the bias will be shown by a small white circle. Weights in these figures are always shown as lines from the bottom, presynaptic neuron to the top, postsynaptic neurons. Arrows are shown here to emphasize this, but are omitted later.

that always has the same value (i.e. is always fully activated).

$$\texttt{net}_j = \sum_{i \in \boldsymbol{Upstream}(j) \cup \{\texttt{bias}\}} \texttt{weight}_{ji} \times \texttt{activation}_i \tag{1}$$

where $\boldsymbol{Upstream}(j)$ is the set of all nodes that are upstream from $\texttt{neuron}_j$ and $\texttt{weight}_{ji}$ is the strength of the connection from node $\texttt{neuron}_i$ into node $\texttt{neuron}_j$. If no such connection exists, the weight is simply 0.

After accumulating the net input, an *activation function* is applied to it to calculate the node's activation. The most commonly used activation function is the *log sigmoid*. Equation 2 and Figure 9a) show the scaled version of the log sigmoid used in this paper. Section 4.3.2 also introduces the *arctangent* and *Gaussian* activation functions as alternatives to the log sigmoid.

$$\texttt{activation}_j = \boldsymbol{A}(\texttt{net}_j) = \frac{2}{1 + e^{-\texttt{net}_j}} - 1 \tag{2}$$

Training In order for a network to learn a new task, it has to be trained. Training algorithms usually adapt the network's weights, because that is where most the network's capability for performing tasks is encoded. Algorithms that dynamically change the network's structure [36] or activation functions [18] are beyond the scope of this paper.

The *supervised learning* paradigm uses a set of input-target pairs to teach the network by giving it examples. This set usually doesn't contain every possible input, so it is important that the network does not just learn the examples that it sees, but is also able to generalize what it learned to unseen cases. To measure the network's generalization ability, the example set is usually divided into a train set and a test set. During training, the network is shown input examples from the train set and its output is compared to the target output using an *error function*. After training, the network's performance is measured by using the test set.

This paper will employ the *squared error function* (Equation 3) to train networks using the *back propagation* learning algorithm with the *generalized delta rule* [61] seen in Equations 4, 5 and 6.

$$E = \frac{1}{2} \sum_{i \in L_{|L|}} (\texttt{target}_i - \texttt{activation}_i)^2 \tag{3}$$

The idea behind this method of learning is that the weights in the network are adjusted in the direction of the steepest descent in the error function. How much a weight will change is determined by the derivative of the error function with respect to that weight, the activation of the presynaptic neuron and a parameter called the *learn rate*. To speed up the learning process, this paper makes use of the *momentum* [47, 52] and *variable learn rate* [27, 70] techniques. More technical details can be found in Appendix A.

$$\delta_j = \frac{\partial \boldsymbol{E}}{\partial \texttt{net}_j} \tag{4}$$

$$\Delta \texttt{weight}_{ji}{}^t = -\texttt{learn rate} \times \delta_j \times \texttt{activation}_j + \texttt{momentum} \times \Delta \texttt{weight}_{ji}{}^{t-1} \tag{5}$$

$$\texttt{weight}_{ji}{}^t = \begin{cases} \text{random number} \in [-1, 1] & \text{if } t = 0 \\ \texttt{weight}_{ji}{}^{t-1} + \Delta \texttt{weight}_{ji}{}^t & \text{otherwise} \end{cases} \tag{6}$$

Here $\texttt{weight}_{ji}{}^{t}$ represents the weight from $\texttt{neuron}_i$ into node $\texttt{neuron}_j$ at a certain time $\texttt{t}$. Equation 7 shows how the δ of a node is calculated when the squared error function is used.

$$\delta_j = \begin{cases} \boldsymbol{A}'(\texttt{net}_j) \times -(\texttt{target}_j - \texttt{activation}_j) & \text{if } j \in L_{|L|} \\ \boldsymbol{A}'(\texttt{net}_j) \sum_{k \in \textbf{Downstream}(j)} \texttt{weight}_{kj} \times \delta_k & \text{otherwise} \end{cases} \tag{7}$$

where $\boldsymbol{A}'$ is the derivative of the activation function (e.g. $-\boldsymbol{A}(\texttt{net}_j)(1 - \boldsymbol{A}(\texttt{net}_j))$ for the log sigmoid).

For each training *epoch* the network iterates over every example in the train set and adjusts the weights to get closer to optimal performance. Using the variable learn rate technique requires training in batches, which means that the weights are updated after every epoch and not after every example within an epoch (see Appendix A for details). The network is trained until certain criteria are met. Useful moments to stop are after a predetermined amount of time has passed or number of epochs have been executed. However, it is also possible to train until the network's performance is satisfactory. The networks in this paper will either be trained until their squared error is smaller than 0.001 or until they are *correct*. The tasks that are used all have target outputs of either +1 and -1 (see Section 4) and a network is called "correct" when it always produces an output with the correct sign (+ or −).

After successsful training the network is capable of classifying the example inputs from the train set correctly. The true power of ANNs lies in their ability to generalize what they have learned. If the training set was representative for the entire task, unseen input vectors can usually be classified correctly as well, but obtaining such a representative train set can often be difficult. Their generalizing power enables ANNs to deal with new situations without having to be explicitly programmed for them, which is what makes them useful in fields like signal classification, image recognition and speech synthesis.

2.2 Representation

ANNs are used to perform tasks, solve problems, find patterns, etc. There are neural networks that learn aspects of languages [7, 62], play the saxophone [54] or play card games [11]. The inputs for the last task could for instance be the cards that the ANN can see as well as information about the actions of the other players. The output might be that the network lays a card or makes some sort of bid. These are not things that neural networks, or software systems in general, are capable of. The inputs for the real-life task need to be translated to something that the ANN can understand and the output needs to be translated to something that makes sense in the real world.

Thinking of a good representation for these inputs and outputs is in general a hard problem, so in most cases fairly arbitrary representations are used [7, 11, 54, 62]. In general, neural networks work fine with these arbitrary representations, presumably because they do not have the prior knowledge to make use of more meaningful ones. It has been shown though, that representing similar real situations with vectors that are close to each other and dissimilar situations with orthogonal vectors can increase performance [20]. One might say that the representations in such an approach are more meaningful.

Creating useful representations is not only a problem for task status inputs however. Even though most people are capable of performing more than one task, they do not always know what

to do in each situation. This could happen for instance if one is dealt a hand of cards without knowing the game, or when sitting down at a chess board without knowing which variant is played [50]. In these cases you need to be told what to do; extra inputs are necessary. This is done using so-called *action words*, which are represented in the brain in a way that is relevant to the task that they describe [29, 32, 37, 51]. The question then is: how to represent which task to perform in neural networks?

To the best of my knowledge, this question has hardly been researched. [74] gives an account of how to recognize and manage what he calls *contextual features*, but does not talk about how they might be specified. [66] use extra input nodes to represent the tasks it knows (and will learn). One *representation node* is used for every task that the network knows. Every node is turned off, except for the one associated with the task that should currently be performed. This approach is called *local*, because a task is represented locally in one node. This approach requires that N representation nodes are used, where N is the number of tasks that the network should be able to learn.

Another approach, which is more *distributed* might only require $\log(N)$ representation nodes (rounded up). So if the network needs to perform 7 different tasks, only 3 input nodes are necessary. Each task representation is given by the binary number signifying how many tasks were learned before it. For the first task (binary `000`) all the nodes are turned off, for the third (binary `010`) the second node is on and for the fourth (binary `011`) only the first node is off, etc.

Both of these methods assign fairly arbitrary representation vectors to each task. They do not even contain any information about the tasks themselves, but rather about the order in which the tasks were learned. In fact, these *representation vectors* are not so much *representing* tasks as they are *identifying* them.

According to [8] "representations are the fruits of perception" and perception cannot be separated from from learning and cognition. They also suggest that learning happens mostly by making analogies between what we already know and the thing we are trying to learn. The way things are perceived depends on the knowledge we have of it and vice-versa. It may very well be the case then, that perceptions and by extension representations, are learned in tandem with task content.

Parametric bias One way to obtain such meaningful task representations, is to treat the representation nodes as regular input nodes and train their activation values using the back propagation training algorithm. Tani et al. [71] have proposed a model called RNNPB (Recurrent Neural Network with Parametric Bias) that can learn to predict multiple time series. It accomplishes this by adding some *parametric bias (PB)* nodes to the input of the network. In the training phase, the network learns all the required time series in an interleaved fashion while determining the PB values for each of them. When the network is required to reproduce a certain time series, it should be fed the corresponding PB values in addition to the regular input. Also, when a time series is shown to the network without clamping the PB nodes, they will automatically converge towards their trained values. In other words they 'recognize' the time series. Because of this, Tani et al. compare these nodes to mirror neurons in human brains, which in turn have been linked to action representation [32].

The activation of these PB nodes can be learned with back propagation in a similar fashion to the weights in the network (see Equations 8 and 9). This is exactly what Tani et al.'s PB nodes do.

$$\Delta \texttt{prv}_j{}^t = -\texttt{learn rate} \times \left(\sum_{k \in \textbf{Downstream}(j)} \delta_k \times \texttt{weight}_{kj}{}^{t-1} \right) + \texttt{momentum} \times \Delta \texttt{prv}_j{}^{t-1} \quad (8)$$

$$\texttt{prv}_j{}^t = \begin{cases} \text{random number} \in [-1, 1] & \text{if } t = 0 \\ \texttt{prv}_j{}^{t-1} + \Delta \texttt{prv}_j{}^t & \text{otherwise} \end{cases} \quad (9)$$

While Tani et al. use these nodes to mimic mirror neurons, this paper will show that they can also be used for learning meaningful task representations.

The name "parametric bias", chosen by Tani et al. is suitable, because PB nodes do in some sense alter the biases of the nodes in the first non-input layer L_2. As mentioned earlier, a node's bias can be thought of as the weight between that node and an imaginary node that is always on, regardless the input. One might say that the bias represents the part of the neuron that is always there, that does not change with the task input. That is why using PB nodes can be thought of as adjusting the biases of the connected nodes: once a task is chosen, their values do not change. In other words: the network is parameterized by the values of the PB nodes so that it can perform different tasks by adjusting these biases.

2.3 Multiple tasks

Most research on ANNs has focused on performing single tasks [69]. Multiple tasks are usually encoded in multiple networks. In fact, sometimes multiple networks are used to perform one task (see e.g. [14, 15, 75]). One reason for the use of multiple smaller networks is that training large networks is very time consuming. Another reason is that *catastrophic interference* may occur when multiple tasks are learned within one network.

Our brains do not seem to suffer from these problems however. Apparently there is a mechanism that allows our natural neural networks to quickly learn new tasks without catastrophically forgetting about the tasks that are already known. Being able to replicate the brain's behavior in this regard would greatly increase both the efficiency and the psychological validity of ANNs as models of the mind and might provide interesting insights into this mechanism.

Earlier research in the field of learning multiple tasks has focused on two distinct topics: avoiding or mitigating catastrophic interference and transfer learning. Transfer learning is about taking advantage of previous knowledge when learning a new task that is related to that knowledge, to learn faster, use less training examples or eventually generalize better. Catastrophic interference is usually avoided by keeping backup copies of the networks that contain the previous knowledge.

Research into the avoidance of catastrophic interference is more often motivated by the idea of creating more plausible models of the mind. Most of the time the focus is not on learning multiple tasks, but on processing the training examples sequentially.

The next two subsections will give a more detailed overview of the respective fields of catastrophic interference and transfer learning research.

2.4 Catastrophic interference

Most people learned the skill of reading aloud in primary school. *NETtalk* is an MLP that can do this as well [62]. A lot of people additionally learn a couple of foreign languages during their education. When they do this, they do not suddenly forget everything about the language(s) that

they already know. This is however, exactly what MLPs normally do, as Section 4.1 will show for a simple task domain.

When an ANN is training, its weights are adjusted so that it can solve the problem or complete the task (read a text aloud). Then, when it tries to learn something new, it will again start adjusting those weights. It will not, however, take into account that it might want to leave those weights intact in case it is ever required to perform the first task again. *Catastrophic interference* or "catastrophic forgetting" is the disruptive effect that learning something new has on existing knowledge. Catastrophic interference is related to the plasticity-stability problem in models of memory, which states that they should be "simultaneously sensitive to, but not radically disrupted by, new input" [23]. Both intuition and research [23] suggest that while some interference may occur between different tasks, humans do not suffer from *catastrophic* interference. It would appear that artificial neural networks, which are abstracted models of the brain, have failed to model the characteristic of human neural networks that allows them to avoid catastrophic interference. Building an ANN that does not suffer from catastrophic interference might therefore provide insight into how this mechanism could work in humans.

The most common way to avoid catastrophic interference in ANNs is to interleave training on the new information with training on the existing knowledge. There are two problems with this: it is not how human learning works and it is terribly inefficient. The human equivalent of this learning technique would be to also keep studying all of the already known languages when learning a new one. This means, amongst others, that the more languages someone would learn, the slower learning a new language would become. In fact, if *Albert* knows no languages at all and *Bart* knows five languages, it would take Bart just as long to learn a new language as it would take Albert to learn all of the six languages that Bart would know. [73] confirms the intuition that having more relevant prior knowledge should make learning new tasks easier, not harder. Actually, interleaving is not even really a solution to the problem. Catastrophic interference is still happening, but the existing knowledge is simply overwritten with something that contains that knowledge as well.

Nevertheless, it has been argued that the rehearsal of so-called *pseudo-patterns* may in fact be more biologically plausible than previously thought [2, 56, 57]. [38] have discovered that learning in the brain may happen in two *Complementary Learning Systems*. The neocortex contains long term memories and is changed only very gradually. The hippocampus is used to quickly learn new tasks with the aid of the knowledge from the neocortex. Eventually the newly learned knowledge is transfered from the hippocampus to the neocortex. A number of dual models are based on these findings [5, 6, 21, 22, 31, 46, 63, 64, 66]. In these models, the existing knowledge is retained by letting the 'neocortical' network generate a train set that basically trains to continue performing the way that it does. The examples in this set are called pseudo-patterns and are generated by associating some input (either random or from the training set of the new task) with the output generated by the network before the new knowledge is incorporated. Next, the network is trained in an interleaved fashion on these pseudo-patterns as well as training examples for the new task. This retention process has been linked back to human REM sleep. Problems with this approach include that it does not account for the fact that we can learn multiple tasks on one day (i.e. before going to sleep) and that it requires a (temporary) copy of the network to generate pseudo-patterns. Furthermore, even though rehearsal of knowledge may happen during REM sleep, it is hard to believe that we rehearse **everything we know** every night.

This last issue is more or less adressed by [20]. He proposed a method called "activation

sharpening" that basically allowed the selection of a subnetwork that should be used to learn each pattern, which precludes catastrophic interference from happening in the rest of the network. Perhaps such a mechanism can be used to generate relevant pseudo-patterns in the earlier mentioned approach. Although activation sharpening reduces catastrophic interference, it also reduces the network's performance and its ability to generalize. Also, the method stops working when two or more patterns elect to use the same subnetwork. This happens when those patterns are too similar to each other or when the number of hidden nodes to choose from is too small. Node sharpening is actually a member of a family of techniques that attempt to cluster the hidden layers of feed forward networks in such a way that input vectors that should be classified differently should show orthogonal activation in the hidden layers [19, 20, 34, 44, 48, 49]. Another slightly different example comes from [40], who propose to pre-train networks with relevant knowledge. This supposedly constrains the number of hidden nodes that are activated by each input pattern as well as greatly speeds up learning [9, 25, 45], but requires that information to pre-train on is available.

Another category of techniques for preventing catastrophic interference is that of constructive neural networks [3, 30, 36, 44, 53, 76]. These networks add and remove nodes as new tasks are presented for learning, which is not biologically plausible since there is already a definite organization of the human brain at birth [4]. One notable and often used example is Grossberg and Carpenter's *Adaptive Resonance Theory (ART)* which was specifically developed to deal with the plasticity-stability issue [30]. ART networks deal well with catastrophic interference, but they are very complex, especially when they have to be adapted in order to support supervised learning (i.e. learning from input-target examples).

[59] take a novel two stage approach to the problem: interference prevention and retroactive interference minimization. In the first stage, the network is trained with initial knowledge in a way as that minimizes future disruption by new knowledge. This is achieved by incorporating a resistance to weight changes in the error function by using noise. In the second stage, the new task is trained using an error function that is the combination of the errors for the old and new tasks. This method helps to mitigate catastrophic interference of new with old tasks, but it is unclear how it would prevent interference of two new tasks.

2.5 Transfer Learning

Another field related to learning multiple tasks is the field of *transfer learning*. Our brains enable us to efficiently learn new things during our entire lives. People can often correctly generalize from only one example [1]. It is believed that this ability is facilitated by the fact that our brains already contain so much relevant knowledge about (most) new tasks. The idea behind transfer learning and the *Machine Life-Long Learning (ML3)* framework is that a learning system should take advantage of the knowledge it already possesses and use it as an *inductive bias* [43] when learning new tasks [72]. It should also be able to continue learning for the rest of its 'life'.

There are basically two distinct approaches to knowledge transfer: *functional transfer* and *representational transfer* [67]. With representational transfer a new task is not trained in a network with randomly initialized weights. With representational transfer the initialization of the weights in the network for learning a new task is biased by the existing knowledge of the system rather than random (see for instance [48, 49]. The main advantage of using this paradigm is that storing representational knowledge requires little memory. A disadvantage can be that accuracy can

decline over time, because the neural network representations are often not perfect.

In the functional transfer paradigm, existing knowledge is used to pressure the new network to share a similar encoding [5, 6, 63–65, 68]. The easiest way to do this is to just remember all training examples of previous tasks and use them in addition to the train set for a new task when learning something new. Storing all training examples takes up a lot of space however. An alternative might be to store neural networks for all of the previously learned tasks and use them to (re-)generate training examples when required. This is similar to the pseudo-pattern based approaches that were mentioned in the previous subsection. This can save storage space, but becomes inaccurate if an input vector from the train set for the new task is not valid for one of the already known tasks.

In Caruana's *multitask learning (MTL)* approach a network learns to perform multiple related tasks at once, even though the actual goal is to learn just one of those tasks [5]. The network accomplishes this by connecting the output nodes of each task to the same hidden layer. This causes them to build a shared encoding of the input space in that hidden layer which may help with generalization, training speed and the number of resources required for learning (if few training examples are available for one task, the network can still train the others). The fact that these networks must always perform all tasks at once and have no way of knowing what task they are supposed to carry out at any one time is obviously not very plausible from a psychological standpoint. Tasks are usually stored in their own networks or consolidated in a neocortical network from McClelland et al.'s Complementary Learning Systems framework [38]. When a new task is trained, it will be done in a network that also attempts to re-learn all previously known knowledge. Despite these issues, the idea has a lot of merit, because it seems very intuitive that we get better at multiple tasks at a time and that learning one might help in learning another. The approach also results in networks that generalize really well.

3 Learning multiple tasks

Section 2.3 already mentioned that most research about ANNs focused on learning single tasks. People are however capable of easily learning multiple tasks. A lot of research has also been done to mimic this behavior (see Section 2.4 and Section 2.5).

Sequential learning of multiple tasks is desirable both because it can potentially be more efficient and because it more closely resembles how people supposedly learn than rehearsed, interleaved training. In one of the first articles about catastrophic interference [53] tried several different strategies for learning a list of words. He argued that there are cases where people in fact do rehearse multiple elements in their minds when learning lists like these. He simulated this by letting a network learn buffers of four words until he knew those words (by interleaved reheasal training). At this point, the network would drop the oldest word from the buffer and add a new one. An approach that performed even better was proposed by [55]. He suggests that new items should be learned in addition to randomly chosen known items (instead of the three previously learned ones). [5] has also suggested that some tasks are indeed learned in parallel (e.g. tennis, running, hand-eye coordination and ball trajectory estimation). However, it is rarely the case that two such tasks are completely new. These accounts suggest that our brain does not learn everything in a sequential manner. The learning method proposed next therefore uses a hybrid philisophy on this topic: tasks are learned in a sequential manner, but the learning of a

task happens by interleaved rehearsal of the training set.

Static Meaningful Representation Learning *Static Meaningful Representation Learning (SRML)* lets networks learn new tasks in the context of existing knowledge without changing the weights in the network. The network only needs to learn a meaningful representation vector for these tasks. One might say that an analogy is learned between the new task and the existing knowledge encoded in the network [24, 33]. The learning of these representations is enabled by the use of parametric bias nodes in the input layer (see Section 2.2). Because the knowledge within the network is unaffected, catastrophic interference is completely avoided. The challenge is to give the representation nodes enough influence over the behavior of the network to actually make it perform a new task.

Like the framework of [59] SMRL consists of two phases. In the *initial knowledge acquisition (IKA)* phase the network's weights are determined. After that, the network becomes static and its weights are fixed. The IKA phase is followed by the *novelty learning (NL)* phase in which the network learns novel tasks by constructing representation vectors. Since these vectors are very small, they can be stored very efficiently for future reference. Although it could be said that this approach uses representational transfer of the initial knowledge, it should not be expected that new tasks are learned faster than with training algorithms like back propagation that are allowed to change all the weights.

SMRL may be viewed as a very extreme form of pre-training the network to avoid catastrophic interference, as is done in [9, 25, 36, 45], or of prohibiting the change of some weights as in [53]. Also, in an analogy with the complementary learning systems of [38] the network's weights may be considered as the neocortical structure and the PB nodes as the hippocampal structure. Like in most MTL approaches, knowledge of previous tasks (the representation vectors) somehow needs to be stored. No satisfactory method is available as of yet, but at least storage of small representation vectors is a lot more efficient than storage of entire neural networks.

The main advantages of the SMRL technique are that it is extremely simple and that it completely avoids catastrophic interference while sequentially learning multiple tasks. It is computationally much more efficient than approaches that have to interleave training on a new task with training on already known tasks and storage of acquired task knowledge is also extremely efficient. The representations that SMRL learns have real meaning in the context of the network and can be viewed as analogies with the existing knowledge. A disadvantage of SMRL is that it is not very biologically plausible that new things are learned without changes in the network's weights. What is learned, is essentially an *analogy* for the existing knowledge [24, 33]. This suggests that the relation between the initial knowledge and new task is important for the success of learning. Choosing the wrong initial knowledge might prohibit the accurate representation of some tasks.

The goal of this paper is to present a a proof of concept for this very simple novel method for learning multiple tasks without suffering from catastrophic interference. The experiments in Section 4.2 aim to test whether the idea has merit and if so, what the relation between existing and new knowledge is and this information might be used to mitigate the disadvantage of being at the mercy of the chosen initial knowledge.

4 Experiments and results

The experiments in this paper will use one of the simplest sets of tasks: all tasks with two Boolean inputs and one Boolean output that are fully specified, which means that the training set contains every possible input combination. These 'tasks' include logical operations such as AND, OR and XOR and are described in Table 1. It is important to note that since every legal combination of inputs is in the training set, there is no way to test the generalizability of the networks.

Inputs		NONE	AND	NIF	1ST	JUST2	2ND	XOR	OR	NOR	IFF	¬2ND	¬JUST2	¬1ST	IF	NAND	ALL
−	−	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+
−	+	−	−	−	−	+	+	+	+	−	−	−	−	+	+	+	+
+	−	−	−	+	+	−	−	+	+	−	−	+	+	−	−	+	+
+	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+
binary		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
#		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 1: Here the 16 tasks with two Boolean inputs and one Boolean output are listed. The tasks are given names for easy reference, some of which are fairly well known (e.g. AND, IF and XOR). For each of the four possible combinations of inputs, the target output for each task is listed. Inputs and target outputs are all either $+1$ or -1. The tasks can also be numbered from 0 to 15 when their outputs are first translated to binary by changing $-$'s into 0's and $+$'s into 1's and concatenating the four target outputs of the task.

The first experiment (Section 4.1) will show that catastrophic interference is indeed a problem when learning two tasks sequentially using arbitrary task representation vectors that merely identify the tasks. In the next experiment (Section 4.2), it will be tested if SMRL can prevent this problem. Section 4.3 describes some attempts to improve the performance of the networks in that experiment.

4.1 Catastrophic interference in MLPs using arbitrary task representations

The first experiment presented here will look at how catastrophic interference affects multi-layer perceptrons learning multiple tasks with arbitrary task representations. Two measures of catastrophic interference are frequently used in the literature: *exact error* and *relearn rate* [23], which are explained below.

MLPs in this experiment are first initialized with random weights after which they are trained on one task until the squared error is under 0.001. Next, the network is trained on a second task. The exact error is obtained by measuring the squared error that the network now makes on the first task. The relearn rate is obtained by counting the number of epochs necessary to train the network until it performs *correctly* on the initial task again (see Section 2.1).

These measures tell us how bad the network now performs on the initial task and how hard it is to regain that performance, but this does not necessarily say a lot about what the network has remembered. In order to test if the MLP benefitted at all from having known the initial task before training on the second, two new measures are introduced: *error benefit* and *relearn benefit*. To obtain these numbers, a second network is trained on the second task. This network however, does not start with knowledge of the first task, but starts with random weights. The error benefit

is obtained by subtracting the exact error from the error that this second network makes on the initial task and the relearn benefit is the relearn rate minus the number of epochs this network needs to learn that task.

The experiment will be conducted for MLPs with different architectures. Both of the methods for determining (arbitrary) representation vectors described in Section 2.2 are tried with networks using either two or four hidden nodes. The rationale behind this is that the most difficult tasks (`XOR` and `IFF`) require two hidden nodes to be learned [42] and that a network with four hidden nodes is exactly twice as large. Since the network has to learn two tasks, it could potentially divide such a larger network in two smaller ones, learn both tasks and switch the correct half of the network on with the RV. The results of this experiment are given in Table 2.

Size of L_2	Method	Learn Success	Relearn Success	Initial Epochs	Exact Error	Relearn Rate	Error Benefit	Relearn Benefit
2	distributed	92.1%	86.1%	13.5	2.68	49.3	0.86	-3.48
4	distributed	96.7%	89.5%	10.2	2.68	37.8	0.97	-4.53
2	local	90.3%	81.2%	14.9	2.37	57.6	1.09	-4.68
4	local	95.6%	86.0%	11.7	2.36	46.9	1.24	-8.24

Table 2: This table shows training statistics for a number of MLPs with different numbers of hidden nodes and representation strategies averaged over 25600 trials with all combinations of tasks from Table 1. The learn success indicates the percentage of cases where the second task could be learned after the first and the relearn success indicates how often the first task could be relearned after successfully training on the second. Other statistics include the number of epochs required to learn the first task (initial epochs), the error on that task after successfully learning another task (exact error) and the number of epochs needed to relearn the first task after that (relearn rate). The error benefit and relearn benefit were obtained by subtracting the exact error and relearn rate for the cases where the network was not trained on the initial task first from those numbers for the cases where it was.

Table 2 clearly indicates that catastrophic interference occurs in the tested MLPs. If this had not been the case, the relearn rates would have been smaller than the initial epochs. It is rather surprising that the relearn rates are actually a lot higher, because the first time the network was trained until it had an error under 0.001 whereas the second time it only needed to be correct, which is much easier and can theoretically be accomplished with an error of slightly less than 2. The relearn rate also does not appear to benefit at all from the network having known about the original task after another task is learned on top of it. Adding more hidden nodes to the network seems to have a beneficial effect on the number of training epochs, whereas adding more representation nodes by using the solo RV generation method instead of the binary approach seems to make training slower.

If no interference had occurred, the exact errors would have remained below 0.001. Table 2 shows that this is clearly not the case. However, it does appear that the networks did not forget *everything* they knew about the task that they learned first, since the error benefit is positive. For a network without representation nodes, the exact error should be 4 (because two tasks have on average two different targets). Using RNs – even arbitrary ones – decreases this number, because they have connections that will move performance towards their associated task. The more of these nodes are used, the smaller the error becomes. The number of hidden nodes does

not appear to affect the exact error much. However, networks with more hidden nodes do benefit more from having learned the initial task first. Although the network does not appear to have forgotten *everything* it knew about the first task, it did not remember much either, so catastrophic interference is definitely occurring.

The next step is to see if using SMRL can prevent this.

4.2 Using SMRL

The basic idea of SMRL is that after the initial knowledge acquisition (IKA) phase, the weights are fixed. In the subsequent novelty learning (NL) phase new tasks are learned by finding appropriate representations for them in the context of that knowledge. Because the network itself cannot change anymore, catastrophic interference cannot occur. The real question is whether it is possible to learn representations for new tasks that are capable of changing the network's behavior in such a way that these new tasks can actually be performed. This clearly depends both on the initial knowledge and on the new task(s) and the relation between them.

To keep things simple, the initial knowledge acquisition phase will consist of training the network on one of the tasks in the task domain using back propagation. That task is given an initially meaningless (random) task representation vector which is trained, along with the rest of the network, on the task. After initial training, the weights of the network are fixed. Next, the network attempts to learn representations for all of the defined tasks in the context of its knowledge of the task that it was initially trained on. The learning of a task representation is considered successful if the network behaves correctly for that task.

Subsequent subsections will discuss several different IKA approaches that differ in the way that the meaningful representation nodes (MRNs) are attached to the network. Differences in the number of hidden layers, hidden nodes and MMRNs are tested as well as different ways of connecting the MRNs to the network. Since having more MRNs means that the hidden nodes can be manipulated more flexibly, it is expected that having more MRNs will result in better performance. Having more hidden nodes also increases the number of weights between the hidden layer and the MRNs, so they are expected to have a positive effect on performance as well. Adding more layers to the network makes the transformation from input into output more gradual and it could be that the meaningful representation vector (MRV) can manipulate network behavior better at an earlier stage of this transformation.

4.2.1 Implicit parametric bias network

The first network architecture employing SMRL idea that is tested is simply a regular MLP that uses PB nodes for task representation (see Figure 2). It is called the "implicit parametric bias (IPB) network" because the PB nodes that it uses can be thought of as indirectly adjusting the biases of the nodes in the first non-input layer L_2 (see Section 2.2).

Results of this experiment will be presented in Section 4.2.4. It turns out that some task combinations can only simetimes be learned by an IPB network. Visual analysis of networks that failed to learn these combinations revealed that they usually had very small weights between the representation nodes and the hidden layer. These weights will be referred to as *representation weights (RWs)*. It is actually not surprising that this happens, because technically the PR nodes are completely unnecessary for learning the initial task. If these weights approach zero, the network will still be able to function fine because these nodes are only necessary for learning multiple tasks,

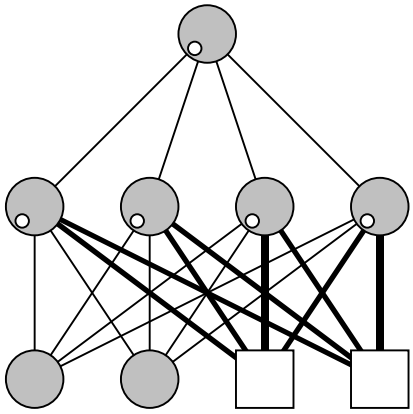


Figure 2: This implicit parametric bias network uses two parametric bias nodes (the squares) to learn to represent different tasks. The thick lines represent the representation weights. For the regular IPB network they are no different than any of the other weights. For the FWIPB network they are semi-randomly initiated to significant values and are not altered by any training otherwise.

and initially the network is only learning one. The next section introduces a variation of the IPB network that addresses this issue.

4.2.2 Fixed weight implicit parametric bias network

The simplest way of ensuring that the MRV is not ignored, is to just set the RWs to significantly large values and then not allow them to change during training. That way, the MRV will always be able to have an effect on the network. These networks will be called "fixed weight IPB (FWIPB) networks" (see Figure 2).

It is however very hard to determine what to use as values for these weights because the importance and role of the hidden nodes (and MRNs for that matter) cannot be known in advance. It may be desirable for one MRN to have a positive effect on one hidden node and a negative effect that is twice as large on another hidden node, but that kind of information is not available before training.

Because of this, a simple normal distribution with average 2 and standard deviation 1 was used to randomly generate numbers for the weights that were multiplied by -1 with a chance of 50%. This yielded connections that should be strong enough to enable the MRVs to have a significant effect.

The results in Section 4.2.4 will show that this often helped to improve performance. Fundamentally however, these networks suffer from the same problem as the IPB networks with trainable RWs: it is still very hard to determine optimal values for those RWs. Training does not yield good values and using random, fixed weights is a poor man's solution. The next section will provide a better solution to this problem.

4.2.3 Explicit parametric bias network

Instead of indirectly adjusting the bias of the nodes in L_2, it is also possible to do it more directly with an *explicit parametric bias (EPB) network* (see Figure 3). The simplest way to think about it is that we eliminate the MRNs from the network and that the MRV for a task just replaces the biases of the nodes in L_2. This is equivalent to a special case of the fixed weights IPB network from the previous section, where each node in L_2 gets its own dedicated MRN (that is not connected to other nodes). The weight between a MRN and the node that it is paired with is 1, while all other RWs are 0. The downside is that this means that in most cases the MRV will need to be larger than before.

EPB networks should perform at least as well as (FW)IPB networks with the same number of hidden nodes, because they can mimic them. For (FW)IPB networks, each node receives the dot product of the MRV and that node's RWs as task representation information. EPB networks can mimic (FW)IPB networks by setting the activation of each MRN to the dot product that the corresponding node would receive in an (FW)IPB network.

4.2.4 Results

This section will now present and analyze the results that were obtained from the experiment with the various networks using SMRL. Some results will be omitted for brevity and are only presented in Appendix B.

Table 3 shows the percentages of cases that the networks were able to learn the representation for a new task in the context of some existing knowledge.

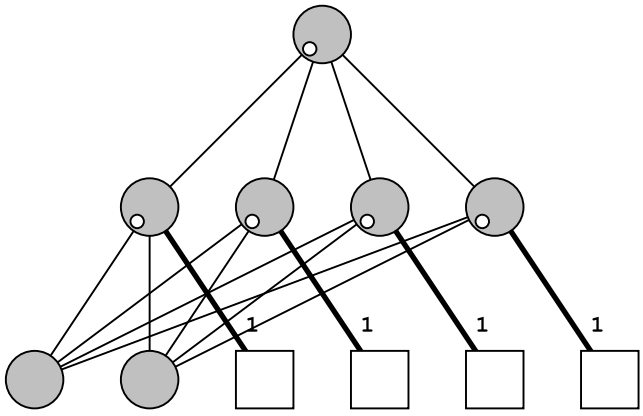


Figure 3: Explicit parametric bias networks use one dedicated parametric bias node for each non-input node that the MRV should directly affect. Each PB node is only connected to one other node with a fixed weight of 1.

# Hidden Nodes	# MRNs	IPB	FWIPB	EPB
2	1	24.9%	26.4%	n/a
4	1	25.6%	26.0%	n/a
2	2	30.4%	31.6%	32.2%
4	2	36.3%	36.7%	n/a
6	2	38.6%	40.5%	n/a
6	6	-	-	54.9%
2+4	2	30.8%	31.6%	32.2%
4+4	2	34.0%	35.7%	n/a
4+4	4	-	-	38.7%
4	4	41.8%	42.3%	42.1%

Table 3: This table shows the percentages of the times that new task representations could be learned in the context of existing knowledge for several different types of networks with different numbers of hidden nodes and MRNs. The sizes of multiple hidden layers are seperated by +-signs, so '2+4' means that there were two hidden nodes in L_2 and four in L_3. For the EPB networks, the number of MRNs has to be equal to the size of L_2, so some of the tested (FW)IPB networks have no real EPB equivalent. Not all of the equivalents of EPB networks were tried either (e.g. the network with six MRNs), because I believe that the strength of the (FW)IPB networks vis-a-vis the EPB networks lies in the fact that they do not need to use as many MRNs.

Performance increases with the number of MRNs, and to a lesser extent the size of the first hidden layer, and decreases when multiple hidden layers are used. As expected, the IPB networks are outperformed by the others, but only slightly. The EPB networks seem to have performed the best, although this might be caused simply by the larger number of MRNs in some cases. When there were four MRNs, the FWIPB network performed a little better. This is surprising, since an EPB network should be able to mimic FWIPB networks. The fact that this does not always happen indicates that EPB networks are more vulnerable to local optima. Further analysis of the data confirms this and indicates that with enough restarts from random representation vectors, FWIPB networks can never outperform EPB networks. Perhaps the potential of the EPB networks can be realized more fully by adjusting the training algorithm or parameters.

Clearly the EPB network with six hidden nodes performs the best. One might argue though, that if that many values are stored, we might just as well store all of the weights of the smallest ANN that could perform each task (which requires seven weights for the hardest tasks and just three or even one for the simpler ones). The disadvantages of such an approach are that it has no psychological foundation and that it will not scale to larger task domains, since the number of weights is a quadratic function of the number of nodes in the network. Storing the four target outputs will not scale either, because the number of target outputs is exponential in the sizes of the input and output layers. Although it needs to be tested if the SMRL approach will work for more interesting tasks, scaling should not be a problem, because even in an EPB network the number of required MRNs increases only linearly with the size of L_2.

Overall the results indicate that sometimes new task representations can be learned in the context of existing knowledge, but that this is definitely not always the case. The rest of this section investigates what distinguishes the situations where sequential learning was effectively achieved from situations where it was not.

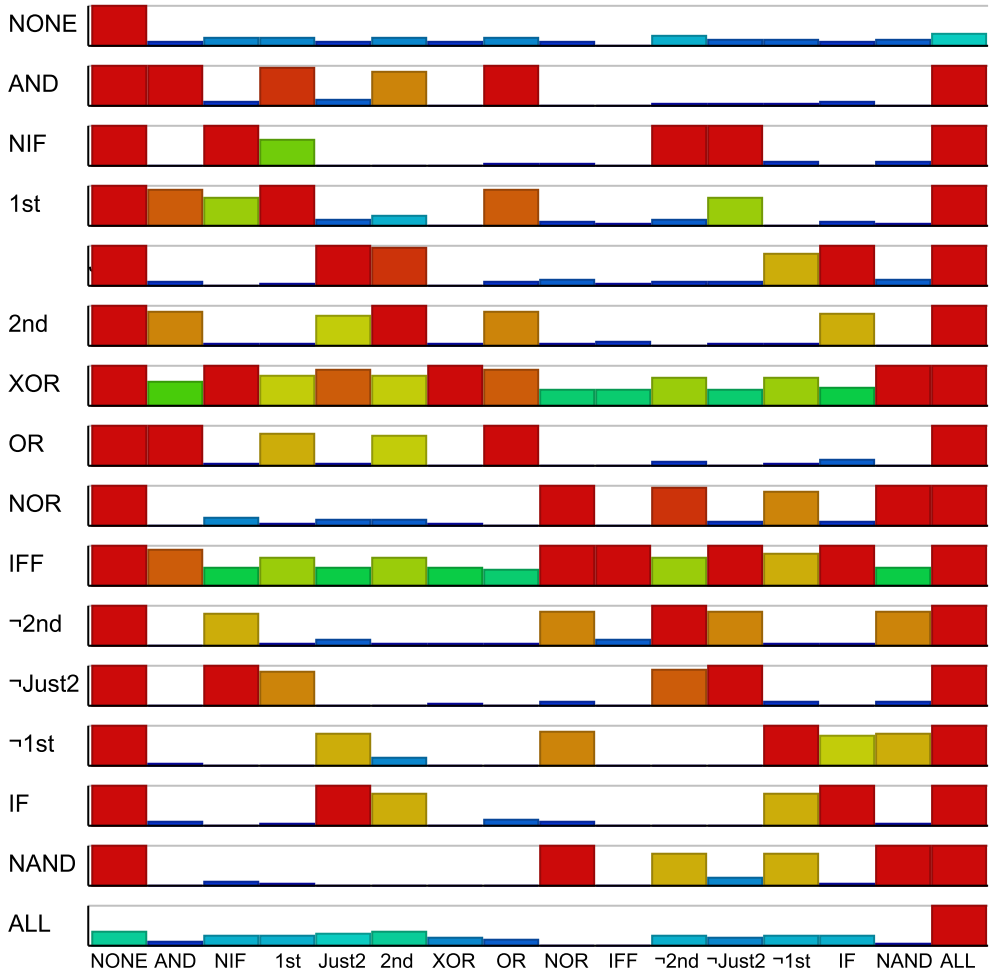


Figure 4: These graphs show the percentages of times that an explicit parametric bias network succeeded in learning the representation for a new task (columns) in the context of initial knowledge of another task (rows).

Figure 4 depicts the success distribution over combinations of tasks for the EPB network with four hidden nodes. Distributions for the other tested networks (see Table 3) are roughly the same (see Appendix B) and clearly show that the combination between initial knowledge (the tasks on the vertical axis) and new tasks (layed out horizontally) matters greatly. There are a lot of combinations of tasks that will succeed or fail almost every time. To use this approach in practical applications it is necessary to be able to predict whether the required (type of) tasks can be learned. Imagine that a system requires six tasks to be learned sequentially and the first five succeed, but the sixth fails. If learning all of the tasks is truly required, the network would now have to start over from scratch (with different initial knowledge) and learn all of those tasks again. This can be very expensive. Also, it is interesting from a theoretical perspective to see if correlations between initial knowledge and new tasks can be found and explained.

Difficulty Figure 4 shows that networks initially trained on `NONE` and `ALL` perform rather poorly and networks initially trained on `XOR` and `IFF` seem to have the potential to learn all other tasks. The other way around, it seems that learning `NONE` and `ALL` as second tasks is easy, but learning `XOR` and `IFF` is almost impossible. This makes sense intuitively, as `NONE` and `ALL` are much easier tasks than `XOR` and `IFF`, because they are both input independent; the output should always give the same value, regardless the input. Networks trained on one of these tasks usually just learn to ignore the inputs (and MRV) and just make the output node always turn on or off. Learning `NONE` and `ALL` in a network trained on another task on the other hand, is generally pretty easy because if the MRV's values are large enough they can overcome the actual task inputs to again make the network relatively independent of those inputs and always have the output node turn on or off. The other way around it appears that having initial knowledge of hard problems like `XOR` and `IFF` is a much better starting point.

For most other tasks, the output can be viewed as a monotonic function of the inputs, which means that the derivative of that function is either always positive or always negative. They are linearly separable, which means that when represented in two dimensions (like is done for the other tasks in Figure 6) the positive points cannot be separated from the negative points by a

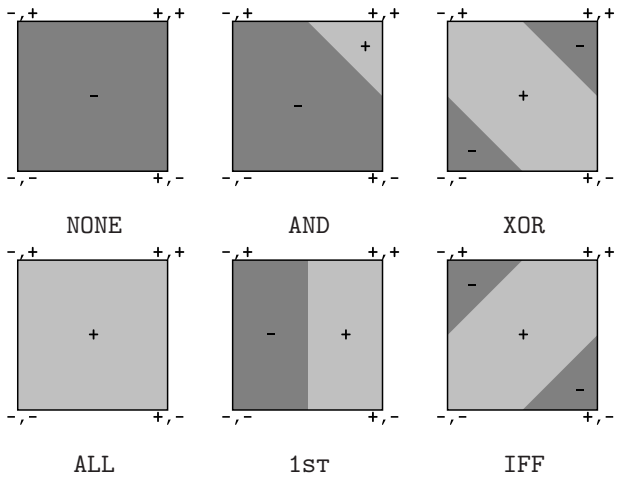


Figure 5: Shows how different (groups of) tasks can be visualized with different numbers of output regions in 2-D input space.

single straight line [26]. The outputs of the linearly inseparable tasks XOR and IFF, on the other hand, depend on their inputs non-monotonically. When visualized in two dimensions, linearly inseparable tasks need at least three output regions to correctly classify all points in input space (see Figure 5). Linear inseparability generally makes learning these tasks a lot harder for ANNs. In fact, these tasks are the only ones in the set that can not be completed without using a hidden layer [42].

Figure 4 shows that knowledge of difficult tasks enables the network to learn representations for the other (easier) tasks most of the time. Table 4 defines the difficulty of each task for this purpose.

Difficulty	Input Dependence	# Regions	Linearly Separable	Tasks
1	none	1	yes	NONE, ALL
2	monotonic	2	yes	AND, NIF, 1ST, JUST2, 2ND, OR, NOR, ¬2ND, ¬JUST2, ¬1ST, IF, NAND
3	non-monotonic	3	no	XOR, IFF

Table 4: This table shows the different difficulty classes that occur in the task domain. The higher the difficulty, the harder the tasks are to learn.

The EPB network in Figure 4 was able to learn new tasks in 42.1% of all of the cases. That percentage was 86.7% for cases where the first task was more difficult than the second and 11.7% when the first task was easier. Similar results were obtained for all of the other networks. This can be interpreted as meaning that in order to learn a representation for a new task, the network has to be 'smart' enough (i.e. trained on a hard(er) task).

Similarity Intuitively it seems that it is easier to learn something that is similar to what you already know, than it is to learn something totally different. This is especially the case for the used neural networks, because the new task has to be learned in terms of the existing knowledge. Furthermore, that existing knowledge is highly specialized, because it is the knowledge used to perform one task. Unfortunately, similarity is in the eye of the beholder, which makes it hard to define [28, 35].

For an ANN "similarity" can be defined as the similarity between the weight distribution of the network and the weight distributions that a network trained on the new task might have [63]. The likely rate of success is defined by the portion of the possible weight distributions for the initial task that look sufficiently like a weight distribution that could encode the second task. However, this is not a very clear and useful definition, because it does not deal with the tasks directly and it is normally not feasible to try and find every possible weight distribution for a task. A similarity measure for the tasks themselves is therefore defined next.

First of all, it should be clear that tasks are similar to themselves. Secondly, each task has an exact opposite to which it should intuitively be dissimilar. Furthermore, the output of the used MLPs is a monotonic function of the inputs for every task except NONE, XOR, IFF and ALL. Each of these monotonic tasks has another task associated with it whose inputs have the exact same effect on the output. For instance, for the AND and OR tasks both inputs have a positive effect on the output, which is simply larger for OR. Table 5 shows the effects of the two inputs on the output. Tasks for which these effects are the same are considered *parallel*, except for NONE and ALL which

are each other's opposites. Tasks for which at most one input effect differs and is zero for one of the tasks are considered *similar*.

Inputs	AND	NIF	1ST	JUST2	2ND	OR	NOR	¬2ND	¬JUST2	¬1ST	IF	NAND
I_1	+	+	+	−	0	+	−	0	+	−	−	−
I_2	+	−	0	+	+	+	−	−	−	0	+	−

Table 5: This table shows the correlations between the input nodes and the target output for all monotonic tasks (i.e. all tasks except NONE, ALL, XOR and IFF). Positive and negative correlations are indicated by +'s and −'s, while a 0 signifies the absence of any correlation.

Another way to look at it is to visualize the input space in two dimensions like in Figure 6. The corners of the grey square represent the different inputs and the black lines divide areas of the input space where the output should be positive and negative for each task. The arrows point in the direction of the positive area and represent the effects that the input values have on the target output for the associated task. If the arrows of two tasks point in the exact same direction they are parallel and they are similar if their directions are roughly the same (i.e. there is less than 90° difference). The tasks not shown in Figure 6 —NONE, ALL, XOR and IFF— are only parallel and similar to themselves.

The EPB network from Figure 4 could learn new representations for tasks in 100% of the cases if they were parallel to the initially learned task and in 62.9% of the cases where they were similar but not parallel. Table 3 indicates that the expected success percentage for any combination of tasks is only 42.1%, so it appears that it is indeed easier to learn representations for tasks that

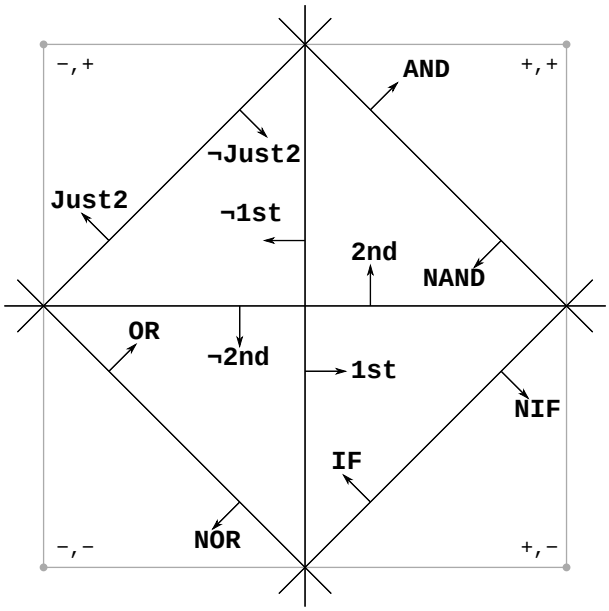


Figure 6: Visualization of all monotonic tasks defined in Table 1 (i.e. all tasks except NONE, ALL, XOR and IFF). The corners of the grey square represent the different inputs. The black lines separate the input space into a part where the output should be positive (the side where the arrow points) and a part where it should be negative.

are similar to the existing knowledge. Similar results were obtained for the rest of the networks as well.

Initial knowledge It might be said that the percentage of correctly learned task representations is not a good measure of success, because it should be attempted to use a network with the best initial knowledge for the job. If a network architecture increases performance for networks with initial knowledge of `NONE` or `ALL` from 10% to 20% it is going to improve in the overall performance of that architecture, but it is not going to matter, since these are still low percentages and `NONE` and `ALL` should quite simply not be used as initial knowledge in any practical application. It appears to be just as, if not more, important to use good initial knowledge for the target application as it is to optimize the network architecture. The only performance measure that counts is the one for the combination of network architecture and initial knowledge that optimizes performance for the set of tasks that a system needs to learn. For instance, if the goal for some application is to read germanic languages aloud, then performance on roman languages is irrelevant. Furthermore, when comparing different networks and different initial knowledge sets, it is the maximal performance using a combination of the two that is important, not a network's average performance for all initial knowledge sets.

The networks in this paper are not meant to perform just a subset of the possible tasks, but all of them, so a good performance measure might be to look at the success rate of the network using the best initial knowledge. I will refer to this measure as the *prodigy* measure. Figure 7 shows the success rates for each task used as initial knowledge. Basically, the harder the task, the higher the success rate. Apparently, the `XOR` network scores the highest, so its performance will be used in the prodigy measure.

Some cognitive scientists argue that we need to assume a lot of innate cognitive structure in order to explain the speed and efficacy of human learning [10]. Analogously, it may be that humans are able to encode multiple tasks on the same neural substrate by having the right kind of inborn structure in place. If this is true, networks using poorly structured initial knowledge can be expected to have trouble learning. It is of interest, then, to study the performance of networks with well structured initial knowledge. The results show that ANNs that encode in their initial weights knowledge of `XOR` and `IFF`, learn best. The knowledge encoded by these networks may in a sense be analogous to the inborn knowledge that facilitates human learning according to some cognitive scientists.

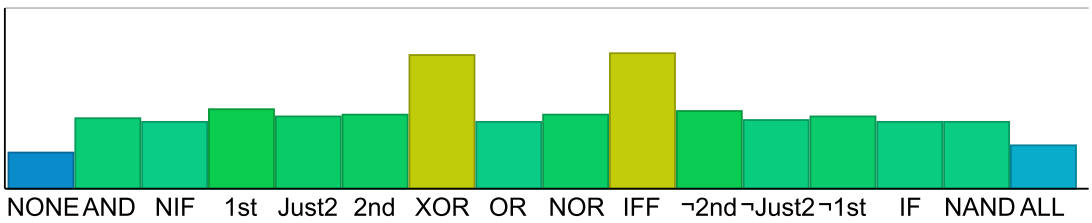


Figure 7: Success percentages of learning all tasks in the context of each individual task as initial knowledge for an EPB network with four hidden nodes.

Measuring performance Table 6 shows the performance of several networks using different measures. Each measure takes into account only a number of task combinations and calculates in how many percent of those cases a new task representation could be learned:

Difficulty combinations where the initial task is more difficult than the second

Parallel all task pairs for which the inputs affect the output in exactly the same way

Similar all task pairs for which the inputs affect the output in roughly the same way

Prodigy combinations between the best initial knowledge and all tasks

Overall all task combinations

Network Type	# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
IPB	4	2	77	98	72	66	36
IPB	4	4	88	100	79	79	42
FWIPB	4	2	81	98	72	69	37
FWIPB	4	4	91	100	79	88	42
EPB	2	2	73	100	68	51	32
EPB	4	4	87	100	80	75	42

Table 6: This table shows the success rates of the best network architectures using several performance measures.

Table 6 shows that the presented notions of similarity and difficulty really do apply to these networks, because the scores on those measures are much higher than the overall scores. It appears that the difficulty notion affects FWIPB networks the most, whereas EPB networks might be the most sensitive to the similarity notion.

The IPB networks seem to be outperformed by the FWIPB networks almost uniformly as expected. On the similarity measures EPB networks score the best, but they perform relatively bad on the difficulty and prodigy measures of success compared to IPB and FWIPB networks with the same number of MRNs.

Depending on the goal of letting an MLP learn multiple tasks, it might be prudent to optimize different measures of success. For instance, if good initial knowledge is available, the network should be able to take advantage of that and performance for cases where the initial knowledge was worse is irrelevant. In that case, the prodigy measure of success can be used.

Finding out that task difficulty and similarity affect success can help to find initial knowledge that works well. For instance, the difficulty measure indicated that the `XOR` and `IFF` tasks would provide the best initial knowledge. The network's initial knowledge does not necessarily have to be defined by (just) one of the tasks. In fact, it can be any weight distribution. Such a weight distribution could for instance be created by training the network on two tasks in the normal, interleaved way. This might be beneficial, because the network might then be less specialized and behave 'similarly' to more tasks. Alternatively, an MTL learning approach could be taken in order to construct a hidden layer that is at least an intersection of the weight distributions for the tasks it was trained with [6]. Even in this simple domain, there are 240 combinations of two different

tasks that could be tried. The difficulty and similarity measure could help to narrow down the search by suggesting that it will probably be most beneficial to combine hard or dissimilar tasks (since the combination might be similar to more other tasks).

In some other cases it might be prudent to optimize a network to take advantage of similarity between the initial knowledge and tasks that the network should be able to learn. For instance, this method could be made to work together with French's activation sharpening technique [20]. Node sharpening does not perform so well on similar patterns, so if MRNs can be used in a way that optimizes for similarity, the techniques could negate each other's weak spots.

New measures can also be devised to suit different goals. For instance, when the network only needs to be able to perform certain tasks, a suitable measure of success should take only those tasks into account. Another interesting thing would be to research which tasks are difficult for humans to learn consecutively and optimizing an ANN to mimic that behavior.

These are all interesting topics for future work.

4.3 Optimizations

In an effort to further increase performance, a number of optimizations were tried that led to interesting results that require future research. Results of these optimizations are directly compared to the results of corresponding networks that do not use them. More detailed results can be found in Appendix B.

4.3.1 Multi-layer spanning connections

Networks with fewer nodes are more efficient than networks with more nodes, because they are faster and require less memory. Normally when an MLP has to learn one of the linearly inseparable tasks `XOR` or `IFF` it needs a hidden layer with at least two nodes. If connections are allowed that span more than one layer, only one hidden node is needed. This suggests that allowing such connections might enable the network to use the nodes that it has more effectively.

When multi-layer spanning connections are allowed, every non-input node can be connected to the MRNs (see Figure 8). For EPB networks, this means that the number of MRNs must increase to the number of non-input nodes.

Network Type	# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
IPB	4	2	83 (+6)	100 (+2)	70 (-2)	74 (+8)	39 (+3)
IPB	4+4	2	81 (+6)	99 (0)	71 (-3)	74 (+16)	39 (+5)
IPB	4	4	94 (+6)	100 (0)	74 (-5)	90 (+11)	43 (+1)
FWIPB	4	2	83 (+3)	100 (+1)	73 (+2)	73 (+4)	40 (+3)
FWIPB	4+4	2	85 (+5)	99 (+0)	76 (+0)	78 (+13)	41 (+5)
FWIPB	4	4	96 (+5)	100 (+0)	80 (+1)	93 (+5)	44 (+2)
EPB	2	3	84 (+11)	100 (0)	69 (+1)	70 (+19)	38 (+6)
EPB	4	5	91 (+4)	100 (0)	73 (-7)	85 (+10)	43 (+1)
EPB	4+4	9	95 (+9)	100 (0)	80 (-1)	91 (+19)	47 (+8)

Table 7: This table shows the increase in success rates for networks where each node had connections to every node downstream from it.

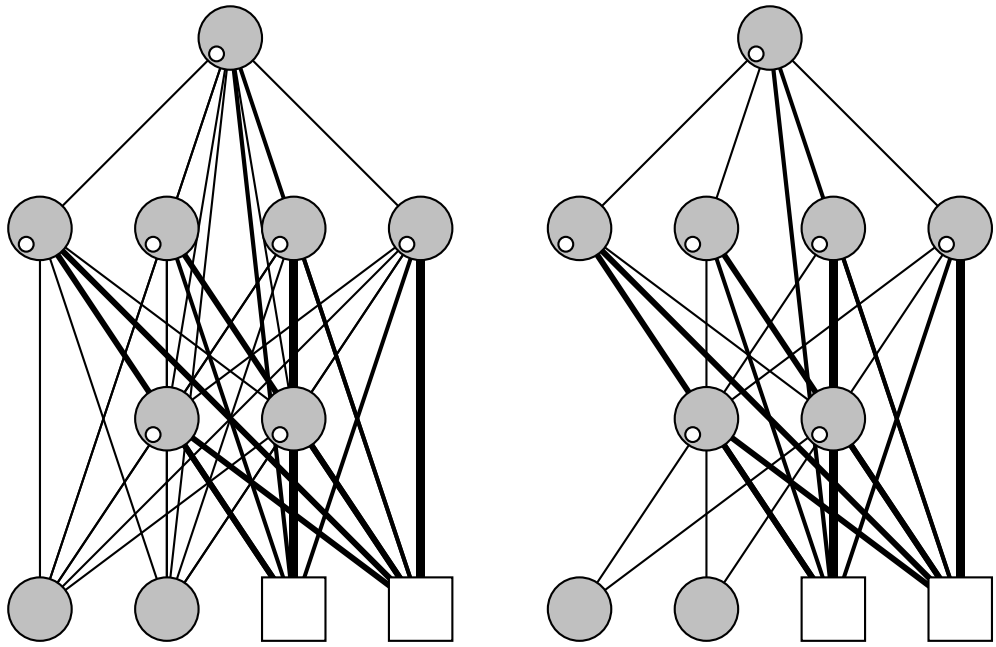


Figure 8: Two FWIPB networks with multi-layer spanning connections. The right network only allows connections from the MRNs to span multiple layers.

Table 7 shows the performance increase of using connections from each node to every other node downstream from it. Overall, difficulty and maximal scores are all increased compared to the case where connections were only allowed to go from one layer to the next, while the similarity score is often decreased. The EPB network with four hidden nodes is the only one that barely benefits from these extra connections overall, which allows IPB and FWIPB networks that even have a MRN less to overtake it in terms of performance. As mentioned before, this is very odd since the EPB networks should be able to mimic IPB and FWIPB networks.

Networks with more than two layers were tested, because this approach was expected to be very dependent on the connections between layers. This did not turn out to be the case however. The only conclusion I can draw is that performance appears to increase with the number of MRNs.

Allowing connections to span multiple layers increases the effect that a presynaptic node can have on a postsynaptic node that it normally would not be connected to. However, since these weights do not change after the initial knowledge acquisition phase, they can also increase the rigidity of the network. The MRNs can use all the power over the output node that they can get, so the increased rigidity might be outweighed by that extra power for them, but the regular task inputs only need to enable the network to differentiate between task states. Because of this it was tried to only connect the MRNs to every non-input node and let other nodes only connect to the nodes in the next layer.

Network Type	# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
IPB	4	2	85 (+8)	98 (+0)	74 (+2)	78 (+12)	41 (+5)
IPB	4+4	2	80 (+5)	99 (-1)	74 (0)	70 (+12)	39 (+5)
IPB	4	4	97 (+9)	100 (0)	87 (+8)	97 (+18)	53 (+11)
FWIPB	4	2	84 (+4)	99 (+1)	77 (+5)	77 (+8)	41 (+5)
FWIPB	4+4	2	83 (+3)	99 (0)	78 (+3)	72 (+7)	40 (+5)
FWIPB	4	4	95 (+4)	100 (0)	90 (+11)	93 (+5)	49 (+7)
EPB	2	3	84 (+11)	100 (0)	74 (+6)	74 (+23)	41 (+9)
EPB	4	5	93 (+7)	100 (0)	89 (+9)	89 (+14)	53 (+11)
EPB	4+4	9	96 (+11)	100 (0)	92 (+11)	92 (+20)	59 (+20)

Table 8: This table shows the performance increase for networks where each MRN had connections to every node downstream from it. All other nodes were only connected to the nodes in the next layer.

Table 8 shows that just allowing the MRNs to have connections that span multiple layers further increases performance – this time on virtually every measure of success. FWIPB networks seem to benefit the least, indicating that the added weights are more sensitive and should not be randomly defined. Contrary to the case where every representation node was given multi-layer spanning connections, EPB networks benefit from this change a lot. It should be noted that while these networks perform better than the other overall, IPB and FWIPB networks perform better on most of the defined measures of success. This indicates that the notions of difficulty and similarity apply better to those networks.

One especially spectacular result is that the IPB network with four hidden nodes with initial knowledge of the `XOR` task learned on average 15.5 of the 16 tasks in the task domain. In the experiment using arbitrary task representations in Section 4.1 the networks also occasionally failed to learn the second task on top of the first. Even though these networks were allowed to change all of their weights, only in 4 of the tested 64 combinations of initial task and network architecture could the network learn 15.6 or 15.7 of the 16 new tasks. In these cases the network of course suffered from catastrophic interference. The network using meaningful task representations on the other hand, was able to learn all of the tasks together in 70% of the cases, without suffering from catastrophic interference at all. It might be suggested that these networks are capable of lifelong learning, because after the IKA phase, they can sequentially learn every possible (legal) task in the domain.

4.3.2 Activation functions

In order to determine the activation of a postsynaptic neuron, the dot product of the activations (the net input) of the presynaptic neurons and the weights between the two is fed into an *activation function* (see Equations 1 and 2). The activation function often bounds the activation of the node between two values (in our case between -1 and $+1$). So far all networks have used a scaled version of the commonly used log sigmoid function (as given in Figure 9a). This monotonically ascending function is S-shaped and is -1 if the input is $-\infty$, 0 if the input is 0 and $+1$ for an input of $+\infty$.

The used activation function deeply affects the way a network works. This subsection aims to

investigate how the use of two other common activation functions can affect the performance of the most interesting network architectures from Section 4.2.

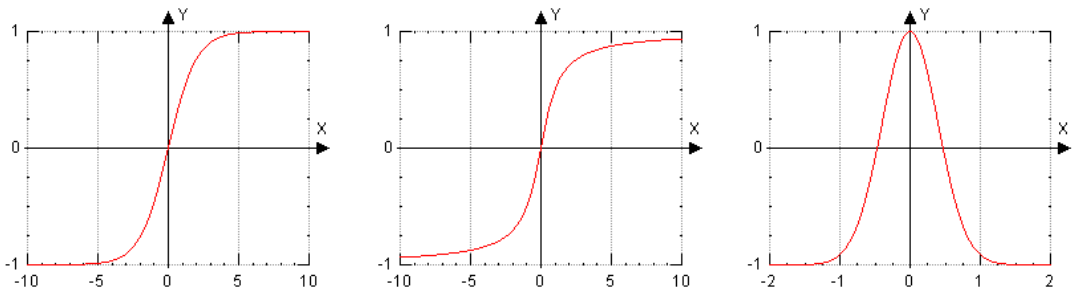
Arctangent Another sigmoid function is the arctangent (Figure 9b). The main differences are that the log sigmoid is saturated much faster (i.e. output values near -1 and +1 are reached earlier) and that the arctangent generally has a steeper slope for large (positive or negative) inputs. Because of this, networks using the arctangent are easier to change and learn slower than those trained with the log sigmoid.

Network Type	# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
IPB	4	2	83 (+6)	100 (+2)	70 (-2)	73 (+7)	37 (+1)
IPB	4	4	90 (+2)	100 (0)	75 (-4)	83 (+4)	40 (-1)
FWIPB	4	2	87 (+6)	100 (+1)	72 (0)	80 (+11)	38 (+1)
FWIPB	4	4	96 (+5)	100 (+0)	78 (-1)	93 (+5)	43 (+0)
EPB	2	2	77 (+4)	100 (0)	64 (-4)	57 (+6)	32 (0)
EPB	4	4	96 (+9)	100 (0)	80 (-1)	93 (+18)	44 (+2)

Table 9: This table compares the success rates of several network architectures using the arctangent activation function with equivalent networks using the log sigmoid.

Table 9 shows that using the arctangent activation function seems to increase the effect that task difficulty has on performance. This also has the effect of dramatically increasing the network's scores on the maximal measure of success. Performance on similar tasks is decreased on the other hand. It appears that some networks can take advantage of the arctangent activation function, but others cannot. Future research should answer the question about in which circumstances the arctangent should be used.

Another thing about these networks should be noted though: training is an order of magnitude slower. The distribution of success across task combinations is very similar to when the log sigmoid was used, so all of the observations made in Section 4.2.4 apply here as well. If a situation arises where use of the arctangent could be beneficial, it might therefore be a good idea to optimize



$\boldsymbol{A}(\boldsymbol{net}_j) = \frac{2}{1+e^{-\boldsymbol{net}_j}} - 1$ (a)

$\boldsymbol{A}(\boldsymbol{net}_j) = \frac{2}{\pi}\boldsymbol{arctan}(\boldsymbol{net}_j)$ (b)

$\boldsymbol{A}(\boldsymbol{net}_j) = 2e^{(-\pi \boldsymbol{net}_j{}^2)} - 1$ (c)

Figure 9: Three activation functions: (a) the log sigmoid function, (b) the arctangent and (c) the Gaussian function.

parameters using the log sigmoid function and use the arctangent in the final application.

Gaussian Unlike the other two activation functions, the Gaussian function is not a sigmoid, but it is bell-shaped like the normal distribution (see Figure 9c). The output is -1 for both $+\infty$ and $-\infty$ and increases to $+1$ when the input gets closer to 0. Networks using this non-monotonic activation function are called "value unit networks" [12]. The Gaussian is more powerful than the other activation functions because it has both ascending and descending parts [41]. Value unit networks for instance do not need a hidden layer to perform the `XOR` task.

[13] observed that Gaussian using neural networks tend to settle in a local optimum where the output will always be false, because for large (positive or negative) net inputs, the derivative of the Gaussian is close to 0. They proposed to use an augmented error function to deal with this problem. [17] argues that it would be even better to augment each derivative separately (see Equation 10), which is the approach that is taken in this paper.

$$\delta_j = \frac{\partial \boldsymbol{E}}{\partial \mathtt{net}_j} - 0.1 \times \boldsymbol{sign}(\mathtt{net}_j) \tag{10}$$

Network Type	# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
IPB	4	2	74 (-3)	86 (-11)	78 (+6)	88 (+22)	73 (+37)
IPB	4	4	87 (-1)	90 (-10)	89 (+9)	97 (+18)	84 (+42)
FWIPB	4	2	74 (-6)	85 (-14)	78 (+6)	86 (+16)	74 (+37)
FWIPB	4	4	82 (-9)	91 (-9)	86 (+7)	92 (+4)	81 (+39)
EPB	2	2	65 (-8)	83 (-17)	69 (+1)	76 (+25)	60 (+28)
EPB	4	4	88 (+1)	95 (-5)	94 (+14)	97 (+22)	89 (+47)

Table 10: This table compares the success rates of several network architectures using the Gaussian activation function with equivalent networks using the log sigmoid.

Table 10 shows that using the Gaussian function greatly increases overall and prodigy performance. The fact that performance on the similarity and difficulty notions is not really that much higher than the overall performance suggests that these notions defined in Section 4.2.4 are a lot less applicable to value unit networks.

Figure 10 shows the success distribution for an explicit parametric bias value unit network with four hidden units. Both the scores in Table 10 and the success distribution in this figure confirm that the concepts of task difficulty and similarity are either very different from those concepts in the networks that use sigmoids or that they do not apply at all. Since value unit networks ignore task monotonicity, it is difficult to devise a difficulty and similarity measure in the same way that was done for sigmoid using networks.

The Gaussian activation function has a lot of potential, but needs to be researched further in order to better understand relations between task combinations and chance of successful learning so that it can be utilized even more effectively.

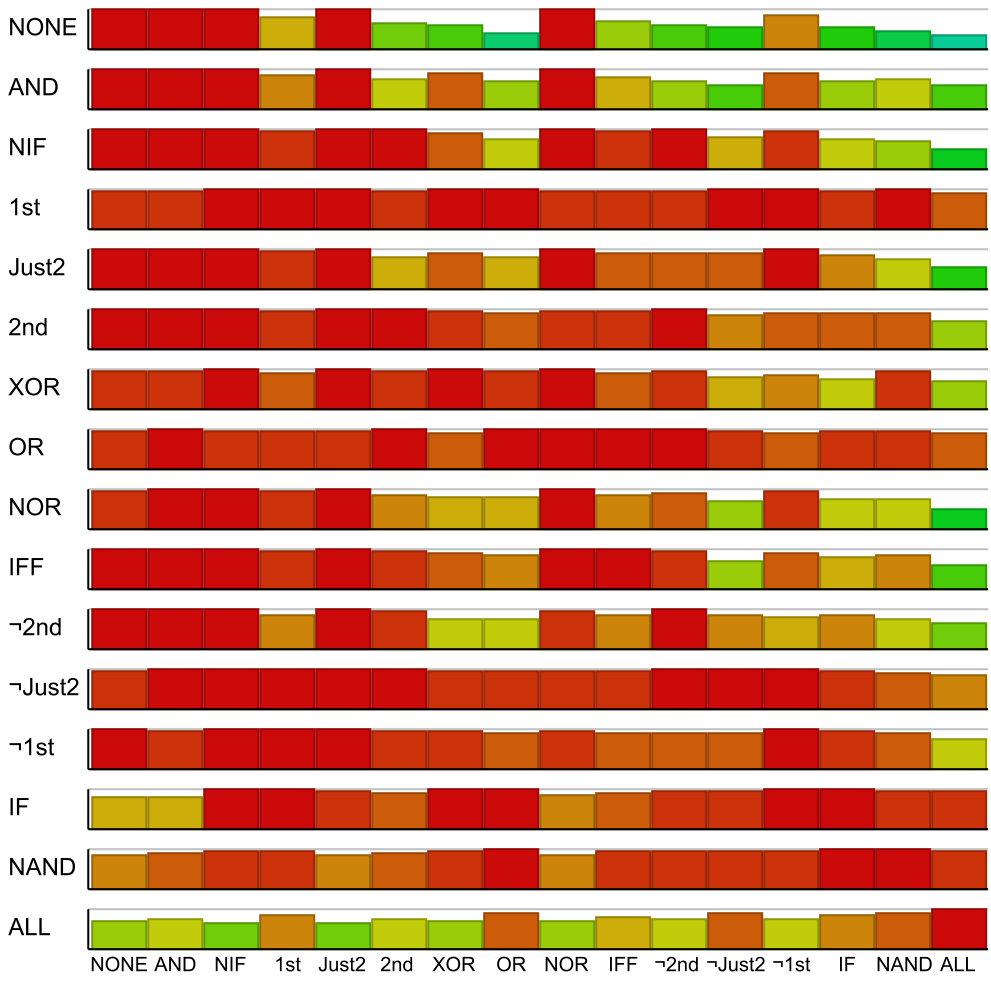


Figure 10: These graphs show the percentages of times that an EPB value unit network with four hidden nodes succeeded in learning the representation for a new task (columns) in the context of initial knowledge of another task (rows). Note that the success distribution is very different from the one in Figure 4.

5 Evaluation

In Section 4.1 the first set of experiments clearly concluded that catastrophic interference occurs in normal MLPs when they attempt to learn even as few as two tasks sequentially. When humans are faced with the challenge of learning multiple similar tasks, they have no such trouble. If neural networks are to be psychologically valid, they should be able to learn multiple tasks just as easily. The goal of this paper was then to devise a method where using meaningful rather than arbitrary task representations would facilitate learning multiple tasks without suffering from catastrophic interference.

In order to accomplish that goal several architectures were tested that made use of SMRL in Section 4.2. The implicit parametric bias network was based on work by Tani et al. [71] and in general performed pretty well on the predicted task combinations. By fixing the weights from the parametric bias nodes to the hidden layer to significantly large values, performance was increased. The explicit parametric bias networks went one step further by having exactly one dedicated representation node for each hidden unit. This more flexible approach eliminates the problem of having to set the representation weights appropriately, but surprisingly did not always increase performance.

Allowing connections to span multiple layers appears to have great potential for increasing performance. Using the arctangent activation function instead of the log sigmoid does, but also makes training a lot slower. Networks employing the Gaussian function behave very differently from networks using a sigmoid. Overall performance is increased, but notions like similarity and difficulty as defined in Section 4.2.4 for sigmoid using networks do not seem to be applicable in the same way. Researching if and how these notions might be defined for value unit networks might give valuable insights and make it possible to benefit from the Gaussian function even more.

Although there is still a lot of room for improvement, the experiments in this paper have clearly shown that SMRL can be very powerful. In many cases it enables a neural network to learn additional new tasks without changing its weight distribution. By letting ANNs avoid catastrophic interference SMRL increases their plausibility as models of the human mind. It is not suggested however that humans learn in the same way as the networks in the paper. Humans are making, strengthening and weakening connections in their brain all the time. Because of this, using meaningful representations is just a start and it will probably be beneficial to use this idea in addition to other methods of sequentially learning. Suggestions for future research on this topic are presented in the next section.

6 Future work

Although the final performance of SMRL in the used task domain is fairly satisfactory, there is still room for improvement. A number of ways in which performance may be improved will be presented in this section. Also, some questions are raised that might be interesting topics of future research.

One thing that should definitely be looked into is why EPB networks can sometimes be outperformed by IPB and FWIPB networks. As mentioned before, this should not happen, because they should be able to mimic them. Overcoming this quirk would make EPB networks use the best of the (FW)IPB networks as a baseline and in some occasions improve from there. This could lead to some very promising results.

Generalizability The networks in this study were trained using training sets that contained every legal input vector. This means that there was no way to see how the networks generalized with respect to unseen inputs. Future research should look into the generalizability of SMRL.

Scalability Another interesting question is whether the results for this simple task domain will scale to more interesting tasks. This paper attempted to provide a proof of concept for SMRL, so it used one of the simplest task domains in existence: fully specified tasks with two Boolean inputs and one Boolean output. The task representations were usually able to transform the behavior of a network in such a way that it would compute the right output for these simple tasks. However, things might quickly get more complex when a task requires multiple outputs, because that would put a lot of extra constraints on each representation value. Also, in this study tasks were considered to be learned when the network would be correct (see Section 4.1) and always give a positive output when the target was +1 and a negative output when the target was -1. If the network needs to be able to produce an output that resembles a target value exactly, the fact that this new method uses only one input independent vector may prove to be too inflexible.

Similarity and difficulty A related issue is that of the similarity and difficulty notions. These notions are either not applicable to networks using the Gaussian activation function or should be defined differently. It could very well be the case that these notions do not apply to other networks in the way that they were defined here. And even if they are, similarity and difficulty might be a lot harder to define for more interesting task domains. If it turns out that the similarity notion is indeed applicable in those domains, it may be the case that the number of similar tasks for any given task is very small, because the percentage of linearly separable tasks quickly decreases as the dimensions of the tasks increase [26]. This does not bode well for the scalability of this concept.

Psychological validity It will also be necessary to investigate the psychological plausibility of neural networks that learn multiple tasks in the way that was used in this paper. The approach avoided some common pitfalls: the use of arbitrary task representations, catastrophic interference and interleaved learning. The pitfall of the presented approach is that no weights changed in the network when it was learning new tasks. Every new task was learned as just a variation on what the network already knew. It seems unlikely that this is how humans learn.

Another thing that might be worth investigating is whether networks can be created that have the same behavior as humans with respect to which things can easily be learned together and whether the order in which these are learned matters. The paper introduced the notions of similarity and difficulty, but although it seems likely that these notions play a part in human learning as well, it is unclear if these notions should be defined in the same way. For instance, the human notion of similarity is probably different from an MLP's notion.

Meaningful RWs In both implicit parametric bias networks (both the one with fixed weights and the one with variable weights) there was a problem with determining useful values for the weights between the MRV and the hidden layer. It was discussed that this is a hard problem, but solving it might yield great performance increases. One possible solution might be to interleave the training of just the MRV with the training of the entire network for the initial training run. That way the MRV always has a meaningful value when the network is training, whereas normally in the beginning the MRV does not mean anything yet. Another way to create meaningful connections

might be to add one or more output nodes to the network that really do depend on the values of the MRNs. For instance, it might be a good idea to use one output node for every MRN with the goal to reproduce that MRN's value. Every once in a while a bogus MRV should be used that would need to be reproduced (but that does not require anything from the regular outputs) to prevent that the network learns to ignore the MRV when it has converged (because then it can always output the same values). The extra output nodes can be removed or ignored after the network has finished training on the initial task.

Initial knowledge An issue related to determining good RW values is the issue of determining good connection strengths in general, or in other words: good initial knowledge. Essentially what we are looking for is some initial knowledge that allows a network to arbitrarily change its behavior based on the values of the MRV. One idea for obtaining this knowledge is to 'cheat' and train on all 16 tasks interleaved, or using MTL. Analyzing the differences between these more optimal networks and others might give valuable insight into how to train an initial network without cheating.

Another suggestion is to initially train the network on two different tasks in an interleaved fashion. Such a network would then be less specialized for just one task and it could even be the case that such a network would learn task independent skills that might enable it to more easily learn other tasks as well.

Variations One of the big ideas behind SMRL is that it always performs perfectly on traditional catastrophic interference measures like relearn rate and exact error because the original network does not really change after it is done learning to perform an initial task. It might be possible to invent a way to slightly change a network's weights to make learning new tasks easier (maybe in addition to changing the initial task's representation vector) without compromising performance on the initial task.

One issue with the explicit PB network was that it used a large representation vector. It might be worthwhile to look into smart ways to determine which nodes (or connections) in the network should contribute to the MRV. This paper simply used one value for each node in the first hidden layer, but maybe it would be sufficient to only have values for the N nodes with, for instance, the largest biases. Another problem for the EPB network was the big influence that the initial values of the untrained MRV had on whether a task could be learned or not, which caused it to perform worse than similar (FW)IPB networks on some occasions. It is not clear how this problem can be remedied but it is obvious that solving this problem would increase EPB network performance.

Also, there are other ways of using a MRV in a network [74]. In the networks in this paper the activation of a MRN was simply multiplied by a weight and added to the net input for a node it connected to. By allowing a MRN to multiply the net input rather than just add some value to it, it might be able to complete certain tasks more easily. For instance, in the performed experiments the network usually had some trouble to learn two tasks that were each other's opposites. With multiplication however, the MRV of one could simply be the negation of the other, which should be fairly simple to learn.

It may also be possible to make the MRV's effect dependent on the regular task inputs. Maybe some task inputs could turn some MRNs on or off and thereby increase flexibility and specialization within the MRV.

Combination Both for increasing performance and accounting for even more phenomena in the human brain it may be beneficial to combine the approach taken in this paper with others. According to French [20] catastrophic interference can be greatly reduced by making sure that the network is sparsely connected using a method called "activation sharpening". Activation sharpening does not work very well with similar input patterns, while this is one of the strengths of using meaningful representations, so it stands to reason that both methods will work well together.

Activation functions Section 4.3.2 showed that using activation functions other than the log sigmoid might also yield some benefits (see also [16]). Using the arctangent was a lot slower than using the log sigmoid, but it did increase the ultimate learning performance. By tweaking some parameters, the tangents slope can for instance be made steeper, which would hopefully allow it to increase training speed while retaining its benefits. The Gaussian activation function, even more so than the arctangent, showed a lot of potential. Research in how to better utilize this function is required. One other idea to explore is to have a heterogeneous network with several different activation functions. This might increase flexibility, which is key in empowering the MRVs.

Parameter optimization Finally, all of the networks and training algorithms depended on parameters such as momentum, learn rate and the range of initial random values for the weights in the network. This paper has mostly used either the default values found in the literature or stopped looking for better values after something satisfactory was found. It is likely that performance can be increased by tweaking these parameters.

7 Conclusion

Artificial neural networks are inspired by the human brain and can be taught to match patterns, solve problems and perform tasks. Catastrophic interference is the phenomenon that occurs in ANNs when they attempt to sequentially learn more than one task. Training on any task completely overwrites all of the network's weights, which results in the forgetting of everything the network could previously do. Since people do not seem to have this problem, this phenomenon makes ANNs less psychologically valid as models of human intelligence. Furthermore, it is obviously very inefficient.

Letting an ANN learn multiple tasks can be accomplished by adding an arbitrary representation vector to the input in order to identify each task. This paper showed that catastrophic interference occurs in multi-layer perceptrons using this paradigm if the tasks are learned sequentially. The underlying idea of static meaningful representation learning (SMRL) is that it would be a lot better if the network would use representations that are meaningful in the context of the knowledge that it already possesses. Learn such representations would not affect the existing knowledge so catastrophic interference would be avoided.

It was shown that this approach can successfully enable networks to learn multiple tasks from a very simple task domain. This approach avoids some of the flaws of MLPs as models of the mind, because it uses meaningful representations, avoids catastrophic interference and facilitates sequential learning. Other aspects of the model are less psychologically plausible, such as the fact that it learns new things without changing its weight distribution. A lot more research needs to

be done to further optimize the method, find out whether it can scale to more interesting task domains and combine the method with others to create more plausible models of the mind.

Bibliography

[1] W.-K. Ahn and W. F. Brewer. Psychological studies of explanation-based learning. *Investigating Explanation-Based Learning*, 1993. 2.5

[2] Bernard Ans, Stéphane Rousset, Robert M. French, and Serban Musca. Preventing catastrophic interference in multiple-sequence learning using coupled reverberating Elman networks. In *Proceedings of the 24th Annual Conference of the Cognitive Science Society*, pages 71–76. Erlbaum, 2002. 2.4

[3] Guang bin Huang, P. Saratch, and Narasimhan Sundararajan. An efficient sequential learning algorithm for growing and pruning RBF (FAP-RBF) networks. *IEEE Transcript on System, Man, and Cybernetics-Part B: Cybernetics*, 34:2284–2292, 2004. 2.4

[4] Renate Nummela Caine and Geoffrey Caine. *Making Connections: Teaching and the Human Brain*. Association for Supervision and Curriculum Development, 1991. 2.4

[5] Richard A. Caruana. Multitask connectionist learning. In *Proceedings of the 1993 Connectionist Models Summer School*, pages 372–379, 1993. 2.4, 2.5, 3

[6] Richard A. Caruana. *Multitask Learning*. PhD thesis, Carnegie Mellon University, 1997. 2.4, 2.5, 4.2.4

[7] Mara Jos Castro, Federico Prat T, and Francisco Casacuberta. MLP emulation of N-gram models as a first step to connectionist language modeling. In *Proceedings of the ICANN '99*, pages 910–915, 1999. 2.2

[8] D. Chalmers, R. French, and D. Hofstadter. High-level perception, representation, and analogy: A critique of artificial intelligence methodology, 1991. 2.2

[9] Chih-Liang Chen and Roy S. Nutter. Improving the training speed of three-layer feedforward neural nets by optimal estimation of the initial weights. In *International Joint Conference on Neural Networks*, pages 2063–2068, 1991. 2.4, 3

[10] N. Chomsky. *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA, 1965. 4.2.4

[11] Aaron Davidson. Using artifical neural networks to model opponents in Texas Hold'em. http://spaz.ca/aaron/poker/nnpoker.pdf, 1999. 2.2

[12] Michael R. W. Dawson and C. Darren Piercey. On the subsymbolic nature of a PDP architecture that uses a nonmonotonic activation function. *Minds and Machines*, 11(2):197–218, 2001. 4.3.2

[13] Michael R. W. Dawson and Donald P. Schopflocher. Modifying the generalized delta rule to train networks of non-monotonic processors for pattern classification. *Connection Science*, 4(1):19–31, 1992. 4.3.2

[14] Kenji Doya, Hiroyuki Nakahara, Raju S. Bapi, and Okihide Hikosaka. Multiple representations and algorithms for sequence learning. In *2nd International Conference on Cognitive Science*, pages 17–19, 1999. 2.3

[15] Kenji Doya and Kazuyuki Samejima. Multiple model-based reinforcement learning. *Neural Computation*, 14:1347–1369, 2002. 2.3

[16] Wlodzislaw Duch and Norbert Jankowski. Transfer functions: hidden possibilities for better neural networks. In *9th European Symposium on Artificial Neural Networks (ESANN), Brugge 2001. De-facto publications*, pages 81–94, 2001. 6

[17] Gary William Flake. *Nonmonotonic activation functions in multilayer perceptrons*. PhD thesis, University of Maryland, College Park, MD, USA, 1993. 4.3.2

[18] Graham P. Fletcher. Adaptive internal activation functions and their effect on learning in feed forward networks. *Neural Processing Letters*, 4(1):29–38, 1996. 2.1

[19] R. M. French. Dynamically constraining connectionist networks to produce distributed, orthogonal representations to reduce catastrophic inference. In *Proceedings of the 16th Annual Cognitive Science Society Conference*, 1994. 2.4

[20] Robert M. French. Using semi-distributed representations to overcome catastrophic forgetting in connectionist networks. In *Proceedings of the 13th Annual Cognitive Science Society Conference*, pages 173–178. Erlbaum, 1991. 2.2, 2.4, 4.2.4, 6

[21] Robert M. French. Interactive tandem networks and the sequential learning problem. pages 222–227, 1995. 2.4

[22] Robert M. French. Pseudo-recurrent connectionist networks: An approach to the 'sensitivity-stability' dilemma. *Connection Science*, 9(4):353–380, 1997. 2.4

[23] Robert M. French. Catastrophic forgetting in connectionist networks: Causes, consequences and solutions. In *Trends in Cognitive Sciences*, pages 128–135, 1999. 1, 2.4, 4.1

[24] Robert M. French and Nick Chater. Using noise to compute error surfaces in connectionist networks: a novel means of reducing catastrophic forgetting. *Neural Computing*, 14(7):1755–1769, 2002. 3

[25] Hussein Alnuweiri Gang Li and Yuejian Wu. Acceleration of back propagations through initial weight pretraining with delta rule. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 1, pages 580–585, 1993. 2.4, 3

[26] Nicole Gruzling. *Number of Linearly Separable Boolean Hypercubes in Each Dimension*. M.Sc Thesis, University of Northern British Columbia, 2006. 4.2.4, 6

[27] Martin T. Hagan, Howard B. Demuth, and Mark Beale. *Neural Network Design*. PWS Publishing Co., Boston, MA, USA, 1996. 2.1

[28] Ulrike Hahn, Nick Chater, and Lucy B. Richardson. Similarity as transformation. *Cognition*, 87(1):1–32, February 2003. 4.2.4

[29] Olaf Hauk, Ingrid Johnsrude, and Friedemann Pulvermüller. Somatotopic representation of action words in human motor and premotor cortex. *Neuron*, 41(2):301–307, January 2004. 1, 2.2

[30] L. G. Heins and D. R. Tauritz. Adaptive resonance theory (ART): An introduction. Technical Report 95–35, Leiden University, 1995. 2.4

[31] Geoffrey E. Hinton and David C. Plaut. Using fast weights to deblur old memories. In *Proceedings of the 9th Annual Conference of the Cognitive Science Society*, pages 177–186. Erlbaum, 1987. 2.4

[32] E. Kohler, C. Keysers, M. A. Umiltà, L. Fogassi, V. Gallese, and G. Rizzolatti. Hearing sounds, understanding actions: action representation in mirror neurons. *Science*, 297(5582):846–848, August 2002. 2.2, 2.2

[33] Boicho Kokinov and Robert M. French. Computational models of analogy-making. http://www.nbu.bg/cogs/events/2004/materials/Boicho/ENCYCL98.pdf, 1998. 3

[34] J. K. Kruschke. ALCOVE: An exemplar-based model of category learning. *Psychological Review*, 99:22–44, 1992. 2.4

[35] Levi B. Larkey and Arthur B. Markman. Processes of similarity judgment. *Cognitive Science*, 29(6):1061–1076, 2005. 4.2.4

[36] T.-C. Lee, A.M. Peterson, and J.-C. Tsai. A multi-layer feed-forward neural network with dynamically adjustable structures. *International Conference on Systems, Man and Cybernetics, 1990. Conference Proceedings., IEEE*, pages 367–369, 1990. 2.1, 2.4, 3

[37] A. Martin and L. L. Chao. Semantic memory and the brain: structure and processes. *Curr Opin Neurobiol*, 11(2):194–201, April 2001. 2.2

[38] J. L. McClelland, B. L. McNaughton, and R. C. O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, 102(3):419–457, July 1995. 2.4, 2.5, 3

[39] Michael McCloskey. Networks and theories: The place of connectionism in cognitive science. *Psychological Science*, 2:387–395, 1991. 2.1

[40] Ken McRae and Phil A. Hetherington. Catastrophic interference is eliminated in pretrained networks. In *Proceedings of the 15h Annual Conference of the Cognitive Science Society*, pages 723–728. Erlbaum, 1993. 2.4

[41] David A. Medler. A brief history of connectionism. *Neural Computing Surveys*, 1:61–101, 1998. 4.3.2

[42] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, Massachusetts, 1969. 2.1, 4.1, 4.2.4

[43] Tom M. Mitchell. The need for biases in learning generalizations, 1980. 2.5

[44] Marco Muselli. On sequential construction of binary neural networks. *IEEE Transactions On Neural Networks*, 6(3):678–690, 1995. 2.4

[45] Derrick Nguyen and Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 21–26, 1990. 2.4, 3

[46] Kenneth A. Norman, Ehren L. Newman, and Adler J. Perotte. Methods for reducing interference in the complementary learning systems model: oscillating inhibition and autonomous memory rehearsal. *Neural Networks*, 18(9):1212–1228, 2005. 2.4

[47] David C. Plaut, Steven J. Nowlan, and Geoffrey E. Hinton. Experiments on learning by back propagation. Technical report, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1986. 2.1

[48] L. Y. Pratt. Non-literal transfer among neural network learners. In *Mammone, R. (Ed.), Artificial Neural Networks for Speech and Vision*, pages 92–04. Chapman and Hall, 1992. 2.4, 2.5

[49] L. Y. Pratt, S. J. Hanson, C. L. Giles, and J. D. Cowan. Discriminability-based transfer between neural networks. In *Advances in Neural Information Processing Systems 5*, pages 204–211. Morgan Kaufmann, 1993. 2.4, 2.5

[50] David B. Pritchard. *The Encyclopedia of Chess Variants*. Games and Puzzles Publications, 1994. 2.2

[51] Friedemann Pulvermüller. Words in the brain's language. *Behavioral and Brain Sciences*, 22:253–336, 1999. 2.2

[52] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999. 2.1

[53] Roger Ratcliff. Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, 97:285–308, 1990. 2.4, 3, 3

[54] Alex Röbel. Neural network modeling of speech and music signals. In *NIPS*, pages 779–785, 1996. 2.2

[55] Anthony Robins. Catastrophic forgetting in neural networks: The role of rehearsal mechanisms. In *The First New Zealand International Two-stream Conference on Artificial Neural Networks and Expert Systems*, pages 65–68, 1993. 3

[56] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7:123–146, 1995. 2.4

[57] Anthony Robins. Sequential learning in neural networks: A review and a discussion of pseudorehearsal based methods. *Intell. Data Anal.*, 8(3):301–322, 2004. 2.4

[58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958. 1

[59] Vicente Ruiz de Angulo and Carme Torras. A framework to deal with interference in connectionist systems. *AI Community*, 13(4):259–274, 2000. 2.4, 3

[60] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *MIT Press Computational Models Of Cognition And Perception Series*, pages 318–362, 1986. 1

[61] David E. Rumelhart and James L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, July 1993. 2.1

[62] Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168, 1987. 2.2, 2.4

[63] Daniel L. Silver and Robert E. Mercer. The parallel transfer of task knowledge using dynamic learning rates based on a measure of relatedness. In *Connection Science Special Issue: Transfer in Inductive Systems*, pages 277–294, 1996. 2.4, 2.5, 4.2.4

[64] Daniel L. Silver and Robert E. Mercer. The task rehearsal method of life-long learning: Overcoming impoverished data. In *AI '02: Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pages 90–101, London, UK, 2002. Springer-Verlag. 2.4

[65] Daniel L. Silver and Ryan Poirier. Sequential consolidation of learned task knowledge. In *Canadian Conference on AI*, pages 217–232, 2004. 2.5

[66] Daniel L. Silver and Ryan Poirier. Context-sensitive MTL networks for machine lifelong learning. In David Wilson and Geoff Sutcliffe, editors, *FLAIRS Conference*, pages 628–. AAAI Press, 2007. 2.2, 2.4

[67] Daniel L. Silver and Ryan Poirier. Requirements for machine lifelong learning. In *IWINAC '07: Proceedings of the 2nd international work-conference on The Interplay Between Natural and Artificial Computation, Part I*, pages 313–319, Berlin, Heidelberg, 2007. Springer-Verlag. 2.5

[68] Daniel L. Silver, Ryan Poirier, and Duane Currie. Inductive transfer with context-sensitive neural networks. *Mach. Learn.*, 73(3):313–336, 2008. 2.5

[69] Satinder P. Singh. The efficient learning of multiple task sequences. In *Advances in Neural Information Processing Systems 4*, pages 251–258. Morgan Kaufmann, 1992. 2.3

[70] Yan-jing Sun, Shen Zhang, Chang-xin Miao, and Jing-meng Li. Improved BP neural network for transformer fault diagnosis. *Journal of China University of Mining and Technology*, 17(1):143–146, 2007. 2.1

[71] Jun Tani, Masato Ito, and Yuuya Sugita. Self-organization of distributedly represented multiple behavior schemata in a mirror system: reviews of robot experiments using RNNPB. *Neural Networks*, 17(8-9):1273–1289, 2004. 2.2, 5

[72] Sebastian Thrun. Lifelong learning: A case study. Technical Report CMU-CS-95-208, Computer Science Department, Pittsburgh, PA, 1995. 2.5

[73] Sebastian Thrun. Is learning the n-th thing any easier than learning the first. In *Advances in Neural Information Processing Systems*, volume 8, pages 640–646, 1996. 2.4

[74] Peter D. Turney. The identification of context-sensitive features: A formal definition of context for concept learning. *CoRR*, cs.LG/0212038, 2002. 2.2, 6

[75] Alex Waibel, Hidefumi Sawai, and Kiyohiro Shikano. Modularity and scaling in large phonemic neural networks. Technical Report TR-I-0034, Advanced Telecommunication Research Institute, International Interpreting Telephony Research Laboratories, Kyoto, Japan, 1989. 2.3

[76] Lu Yingwei, N. Sundararajan, and P. Saratchandran. Performance evaluation of a sequential minimal radial basis function (RBF) neural network learning algorithm. *IEEE Transactions on Neural Networks*, 9(2):308–318, 1998. 2.4

Appendix

A Algorithms

This appendix describes some of the algorithms used in the paper, as well as the parameters that they were used with. Java program code and documentation are available upon request. The following definitions were used:

$\boldsymbol{Upstream}(i)$	all the neurons that are upstream from $neuron_i$ (i.e. in layers closer to the input layer)
$\boldsymbol{Downstream}(i)$	all the neurons that are downstream from $neuron_i$ (i.e. in layers closer to the output layer)
$\boldsymbol{A}(netinput_i)$	the used activation function (by default the log sigmoid from Equation 2)
$\boldsymbol{E}(network)$	the error function (by default the squared error function from Equation 3)
$\boldsymbol{A'}(netinput_i)$	the derivative of the activation function
$\boldsymbol{E'}(activation_i)$	the derivative of the error function with respect to the activation of output $neuron_i$

Algorithm 1 shows how the activation for a neuron is determined. It assumes that $activation_i$ is clamped for all input nodes $neuron_i$.

Algorithm 1 Get the activation value for a node

function ***get_activation***(i)
  if $neuron_i \notin L_1$ **then**
    $netinput_i := 0$
    for each $neuron_j \in \boldsymbol{Upstream}(i) \cup \{bias\}$ **do**
      $netinput_i \leftarrow netinput_i + weight_{ji} \times \boldsymbol{get\_activation}(j)$
    end for
    $activation_i := \boldsymbol{A}(netinput_i)$
  end if
  return $activation_i$

Algorithms 2, 3, 4 and 5 show the back propagation training algorithm using a momentum term and the variable learn rate algorithm. The ***train*** algorithm trains the network until either some goal is reached (i.e. the network makes a squared error of 0.001 or is *correct* as defined in Section 2.1), or until it has trained for a specified number of epochs. If the latter happens, then the training algorithm tries to restart a specified number of times with different starting values for the weights and PB nodes.

Algorithm 2 The back propagation training algorithm

function ***train***(goal, epochs, restarts)
  epoch := *epochs*
  while *restarts* ≥ 0 and *epoch* = *epochs* **do**
    epoch $\leftarrow 0$
    while *goal* not satisfied and *epoch* < *epochs* **do**
      initialize
      trainepoch
      epoch $\leftarrow$ *epoch* + 1
    end while
    restarts $\leftarrow$ *restarts* $- 1$
    reset each non-fixed weight and PB node to new random values
  end while

function ***initialize***
  `momentum` := 0.9
  `learnrate` := 0.7
  `min_learnrate` := 0.001
  `max_learnrate` := 5000
  `decrement_learnrate` := 0.7
  `increment_learnrate` := 1.05
  `max_error_increase` := 1.04
  `min_error_decrease` := 1
  for each $\texttt{weight}_{ji}$ in the network **do**
    $\Delta_{ji} := 0$
  end for
  for each $\texttt{neuron}_i \in$ `PB nodes` **do**
    $\Delta_i := 0$
  end for

Algorithm 3 Train one epoch

function ***trainepoch***
  for each $\texttt{inputs}_i$ in the training set **do**
    clamp input nodes to $\texttt{inputs}_i$
    traincase
  end for
  learn

The ***traincase*** function 'trains' the network for one example, but because batch mode was used, the weights and PB nodes are not actually changed yet until the ***learn*** function is called. The ***traincase*** function just prepares the network for change by adjusting a Δ parameter for each trainable value (weights and PB nodes). This is done by calculating δ values for every non-input node and differentiating the error function towards that node's activation.

Algorithm 4 Train one example from the training set

function ***traincase***
  for each $neuron_j \in L_{|L|}$ **do**
    $\delta_j := \boldsymbol{E'}(\textbf{\textit{get\_activation}}(j)) \times \boldsymbol{A'}(netinput_j)$
  end for
  for $l \leftarrow [2 \ldots |L| - 1]$ **do**
    for each $neuron_j \in L_l$ **do**
      $error := 0$
      for $neuron_j \in \textbf{\textit{Downstream}}(i)$ **do**
        $error \leftarrow error + weight_{ji} \times \delta_i$
      end for
      $\delta_j := error \times \boldsymbol{A'}(netinput_j)$
    end for
  end for

  for $weight_{ji}$ in the network **do**
    $\Delta_{ji} \leftarrow \Delta_{ji} + activation_i \times \delta_j$
  end for
  for each $neuron_i \in PB\ nodes$ **do**
    for each $neuron_j \in \textbf{\textit{Downstream}}(i)$ **do**
      $\Delta_i \leftarrow \Delta_i + weight_{ji} \times \delta_j$
    end for
  end for

Without the variable learn rate algorithm, the ***learn*** function would just be the ***trylearn*** function from Algorithm 5, but since it *is* used, the function is a lot more complicated. First, a backup is created of the current state of the network so that any changes can be undone. Then the network is changed with the ***trylearn*** function. Because the weights and activations of PB nodes are moved in the direction of the error gradient, the error can never increase if no momentum is used and the move is small enough. If the error did increase by a large enough margin, the back up of the network is restored and the learn rate and momentum are decreased (the learn rate permanently and the momentum momentarily). Next, learning is tried again and this cycle repeats until either the network's error is small enough or the learn rate can not decrease anymore. If, on the other hand, the error decreased immediately and sufficiently, the learn rate is increased so that learning in the next epoch may be faster.

Algorithm 5 Perform the actual learning using the variable learn rate algorithm

function ***learn***
  `backup` := backup of the network state
  `old_error` := $E(network, trainset)$
  trylearn(`learnrate`, `momentum`)
  `new_error` := $E(network, trainset)$
  `multiplier` := 1
  while `new_error` > `max_error_increase` × `old_error` **do**
    `multiplier` ← `multiplier` × `decrement_learnrate`
    $network$ ← `backup`
    `learnrate` ← `learnrate` × `decrement_learnrate`
    if `learnrate` < `min_learnrate` **then**
      `learnrate` ← `min_learnrate`
      trylearn(`learnrate`, 0)
      return
    end if
    trylearn(`learnrate`, `momentum` × `multiplier`)
    `new_error` ← $E(network, trainset)$
  end while
  if `multiplier` = 1 and `new_error` < `min_error_decrease` × `old_error` **then**
    `learnrate` ← max(`max_learnrate`, `learnrate` × `increment_learnrate`)
  end if

function ***trylearn***(`learnrate`, `momentum`)
  for $weight_{ji}$ in the network **do**
    $weight_{ji}$ ← `learnrate` × Δ_{ji} + `momentum` × $weight_{ji}$
    Δ_{ji} ← 0
  end for
  for each $neuron_i \in$ `PB nodes` **do**
    $activation_i$ ← `learnrate` × Δ_i + `momentum` × $activation_i$
    Δ_i ← 0
  end for

B Raw results

This appendix will list all of the results that were obtained while doing the experiments in Sections 4.2 and 4.3. Some of these results in the paper were omitted for brevity or presented in a comparative way to support the narrative. Here they are presented in a more complete and raw form.

The Figures show the performance of several networks for each combination of problems. Each bar represents the percentage of times that a new problem (on the horizontal axis) could be learned in the context of the knowledge of the problem on the vertical axis.

The Tables each show the performance of several networks on the measures of success that were defined in Section 4.2.4. Each measure defines a set of problem combinations that it will take into account to determine the score. The percentages represent how many of these combinations the network could successfully learn on average.

Initial setup

This subsection contains the results of the experiments with meaningful problem representation vectors using the initial setup. This means that the networks here used the log sigmoid function and only had connections between adjacent layers.

IPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
2	1	54	93	56	32	25
4	1	56	91	53	39	25
2	2	68	100	67	44	31
4	2	77	98	72	66	36
6	2	79	99	73	66	38
2+4	2	70	100	67	47	31
4+4	2	75	99	74	58	34
4	4	88	100	79	79	42

Table B-1: This table shows the performance of several IPB networks (see Section 4.2.1) using the log sigmoid activation function (see Section 2.1) on the measures of success defined in Section 4.2.4.

from left to right								
# Hidden nodes	2	4	2	4	6	2+4	4+4	4
# PR nodes	1	1	2	2	2	2	2	4

Figure B-1: This figure shows the performance of some IPB networks (see Section 4.2.1) split out across problem representations.

FWIPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
2	1	59	94	57	34	26
4	1	60	91	54	41	26
2	2	73	100	67	50	32
4	2	81	98	72	69	37
6	2	84	99	77	75	41
2+4	2	74	100	66	54	32
4+4	2	80	99	75	65	36
4	4	91	100	79	88	42

Table B-2: This table shows the performance of several FWIPB networks (see Section 4.2.2) using the log sigmoid activation function (see Section 2.1) on the measures of success defined in Section 4.2.4.

from left to right								
# Hidden nodes	2	4	2	4	6	2+4	4+4	4
# PR nodes	1	1	2	2	2	2	2	4

Figure B-2: This figure shows the performance of some FWIPB networks (see Section 4.2.2) split out across problem representations.

EPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
2	2	73	100	68	51	32
4	4	87	100	80	75	42
6	6	93	100	89	88	55
2+4	2	74	100	69	52	32
4+4	4	85	100	81	72	39

Table B-3: This table shows the performance of several EPB networks (see Section 4.2.3) using the log sigmoid activation function (see Section 2.1) on the measures of success defined in Section 4.2.4.

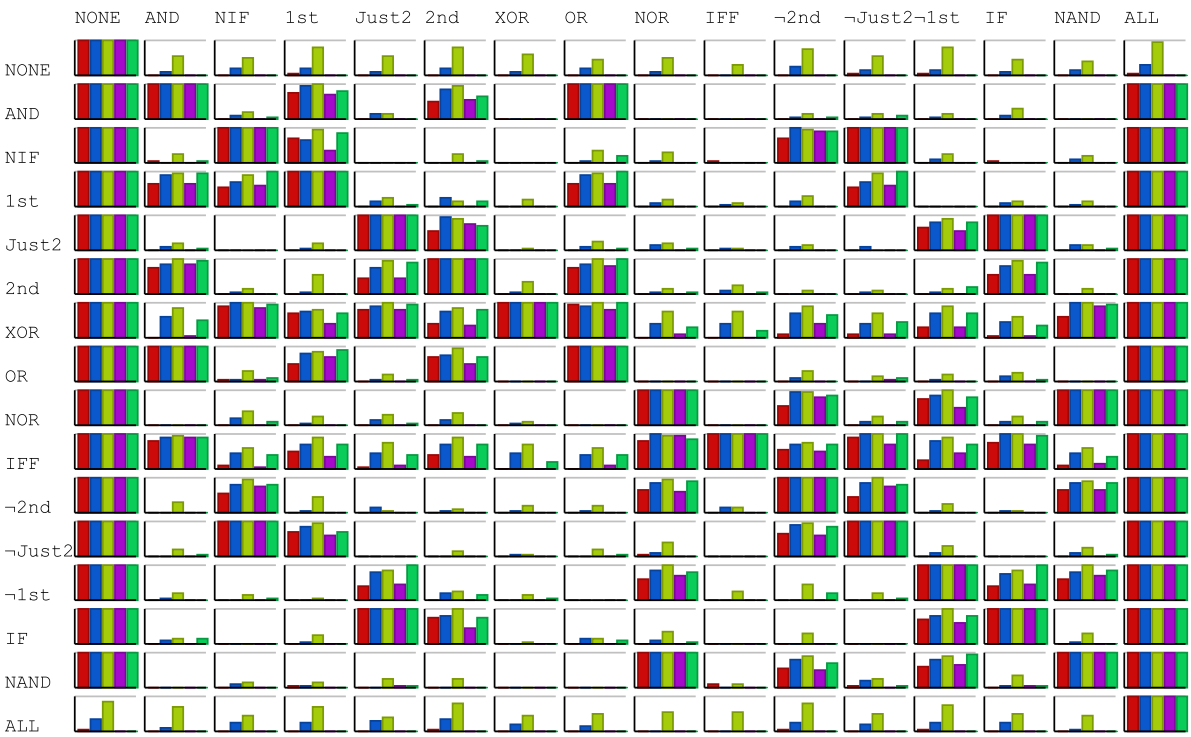


Figure B-3: This figure shows the performance of some EPB networks (see Section 4.2.3) split out across problem representations.

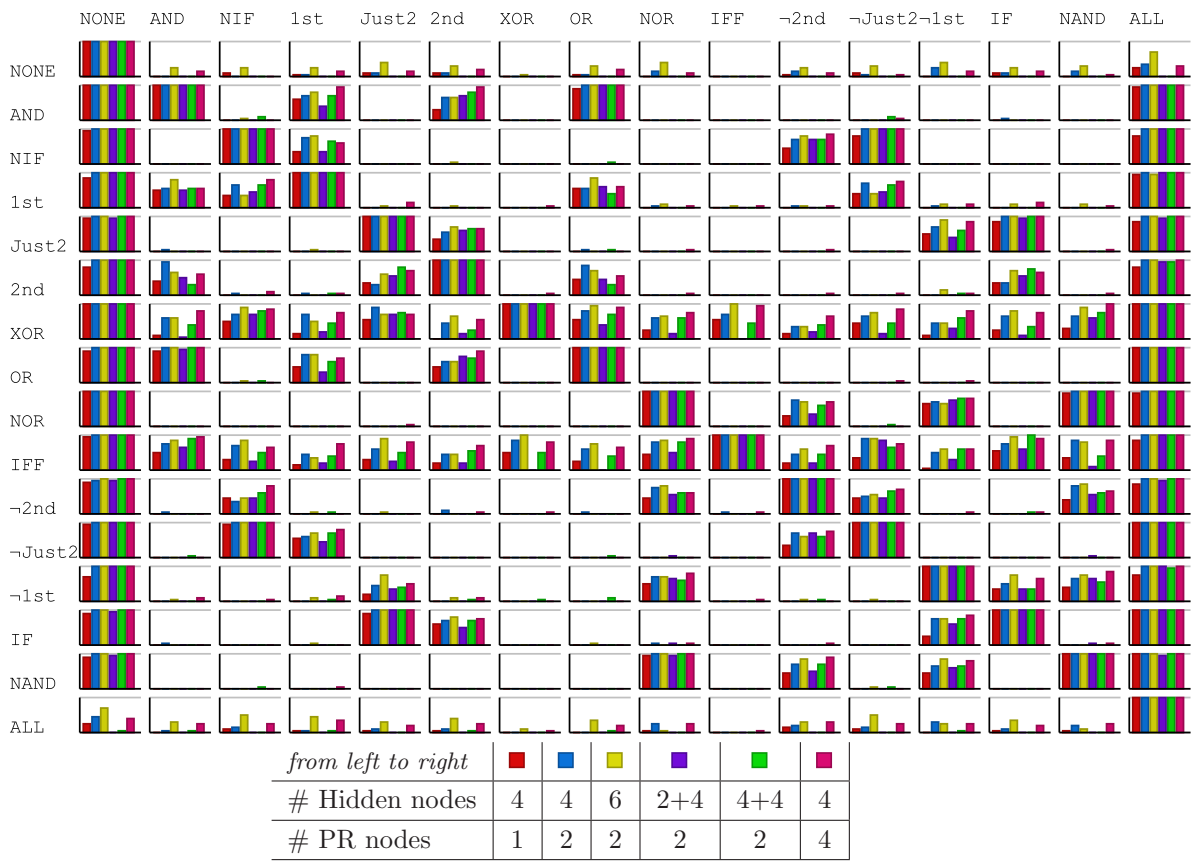
Arctangent activation function

This subsection* lists the results for the case where the arctangent activation function was used.

IPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
4	1	63	97	57	43	28
4	2	83	100	70	73	37
6	2	87	100	74	80	41
2+4	2	68	99	63	44	30
4+4	2	80	100	69	67	34
4	4	90	100	75	83	40

Table B-4: This table shows the performance of several IPB networks (see Section 4.2.1) using the arctangent activation function (see Section 4.3.2) on the measures of success defined in Section 4.2.4.



from left to right						
# Hidden nodes	4	4	6	2+4	4+4	4
# PR nodes	1	2	2	2	2	4

Figure B-4: This figure shows the performance of some IPB networks (see Section 4.2.1) using the arctangent activation function split out across problem representations.

FWIPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
4	1	63	94	57	46	28
4	2	87	100	72	80	38
6	2	89	99	76	85	41
2+4	2	76	100	64	60	31
4+4	2	84	100	70	73	35
4	4	96	100	78	93	43

Table B-5: This table shows the performance of several FWIPB networks (see Section 4.2.2) using the arctangent activation function (see Section 4.3.2) on the measures of success defined in Section 4.2.4.

from left to right						
# Hidden nodes	4	4	6	2+4	4+4	4
# PR nodes	1	2	2	2	2	4

Figure B-5: This figure shows the performance of some FWIPB networks (see Section 4.2.2) using the arctangent activation function split out across problem representations.

EPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
2	2	77	100	64	57	32
4	4	96	100	80	93	44
2+4	2	80	100	64	63	32
4+4	4	90	100	77	80	39

Table B-6: This table shows the performance of several EPB networks (see Section 4.2.3) using the arctangent activation function (see Section 4.3.2) on the measures of success defined in Section 4.2.4.

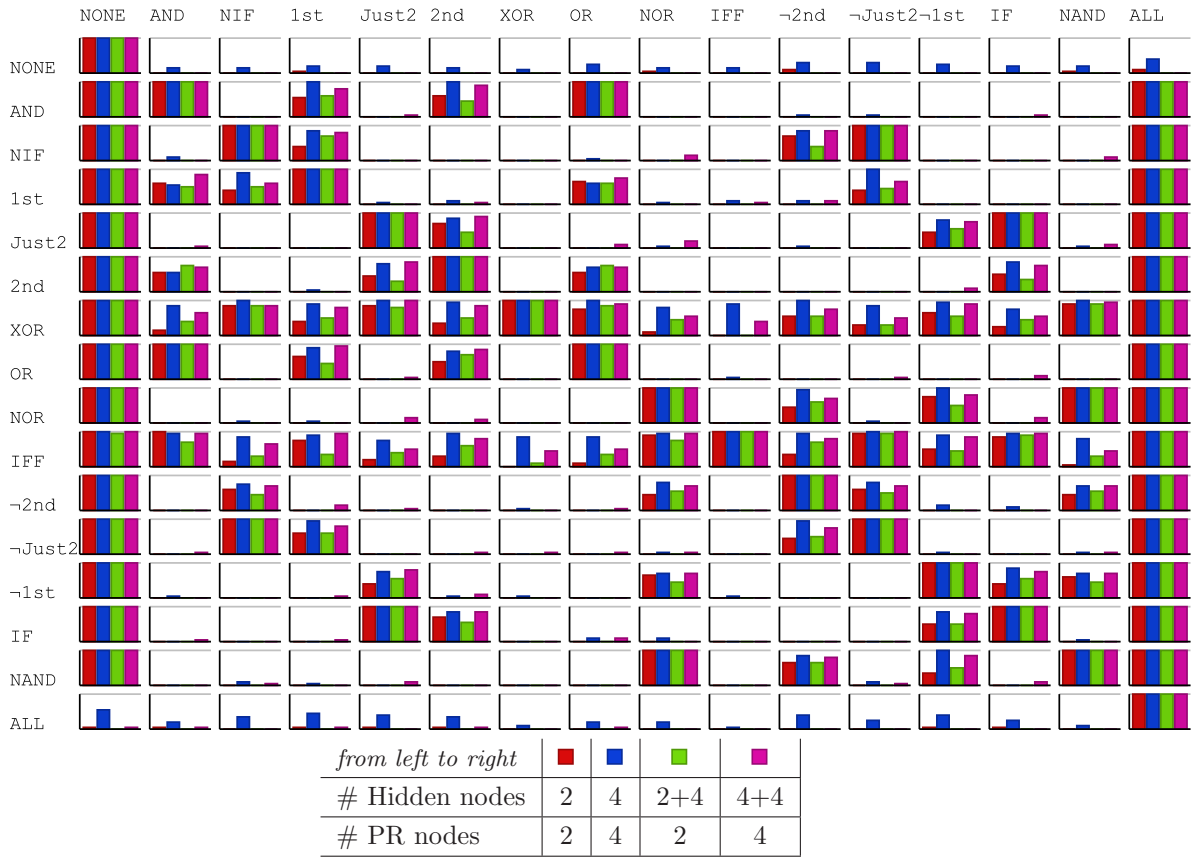


Figure B-6: This figure shows the performance of some EPB networks (see Section 4.2.3) using the arctangent activation function split out across problem representations.

Gaussian activation function

The next couple of results were obtained from ANNs using the Gaussian activation function.

IPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
4	1	47	78	56	51	42
4	2	74	86	78	88	73
6	2	78	88	83	93	78
2+4	2	68	86	75	78	66
4+4	2	78	88	85	87	77
4	4	87	90	89	97	84

Table B-7: This table shows the performance of several IPB networks (see Section 4.2.1) using the Gaussian activation function (see Section 4.3.2) on the measures of success defined in Section 4.2.4.

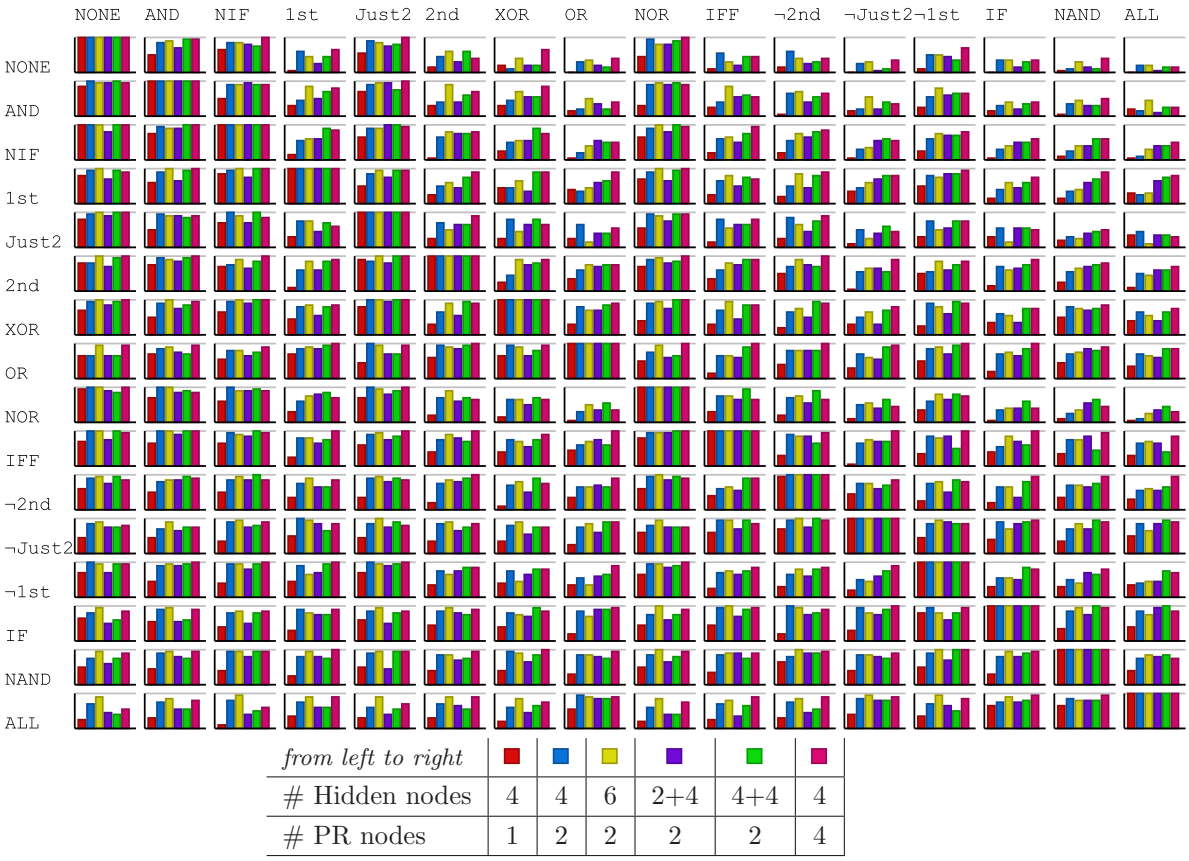


Figure B-7: This figure shows the performance of some IPB networks (see Section 4.2.1) using the Gaussian activation function split out across problem representations.

FWIPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
4	1	58	81	59	60	51
4	2	74	85	78	86	74
6	2	77	89	84	91	78
2+4	2	63	82	70	75	63
4+4	2	80	90	82	85	74
4	4	82	91	86	92	81

Table B-8: This table shows the performance of several FWIPB networks (see Section 4.2.1) using the Gaussian activation function (see Section 4.3.2) on the measures of success defined in Section 4.2.4.

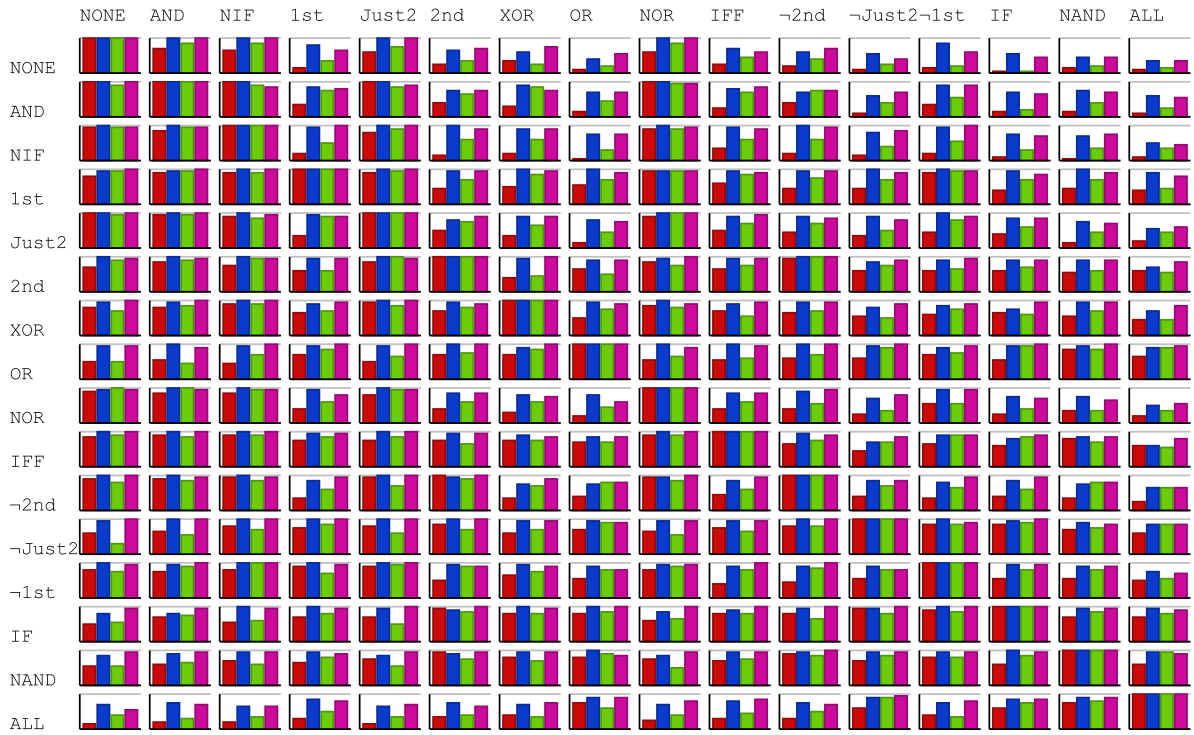
from left to right						
# Hidden nodes	4	4	6	2+4	4+4	4
# PR nodes	1	2	2	2	2	4

Figure B-8: This figure shows the performance of some FWIPB networks (see Section 4.2.2) using the Gaussian activation function split out across problem representations.

EPB network

# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
2	2	65	83	69	76	60
4	4	88	95	94	97	89
2+4	2	72	83	79	83	71
4+4	4	89	95	95	98	89

Table B-9: This table shows the performance of several EPB networks (see Section 4.2.1) using the Gaussian activation function (see Section 4.3.2) on the measures of success defined in Section 4.2.4.



from left to right				
# Hidden nodes	2	4	2+4	4+4
# PR nodes	2	4	2	4

Figure B-9: This figure shows the performance of some EPB networks (see Section 4.2.3) using the Gaussian activation function split out across problem representations.

Multi-layer spanning connections

Next the results are presented for the case where connections between non-adjacent layers were allowed (see Section 4.3.1). These networks again used the log sigmoid activation function.

Completely connected

Network Type	# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
IPB	4	2	83	100	70	74	39
IPB	4+4	2	81	99	71	74	39
IPB	4	4	94	100	74	90	43
FWIPB	4	2	83	100	73	73	40
FWIPB	4+4	2	85	99	76	78	41
FWIPB	4	4	96	100	80	93	44
EPB	2	3	84	100	69	70	38
EPB	4	5	91	100	73	85	43
EPB	4+4	9	95	100	80	91	47

Table B-10: This table shows the performance of several networks where each node was connected to all of the nodes that are further downstream.

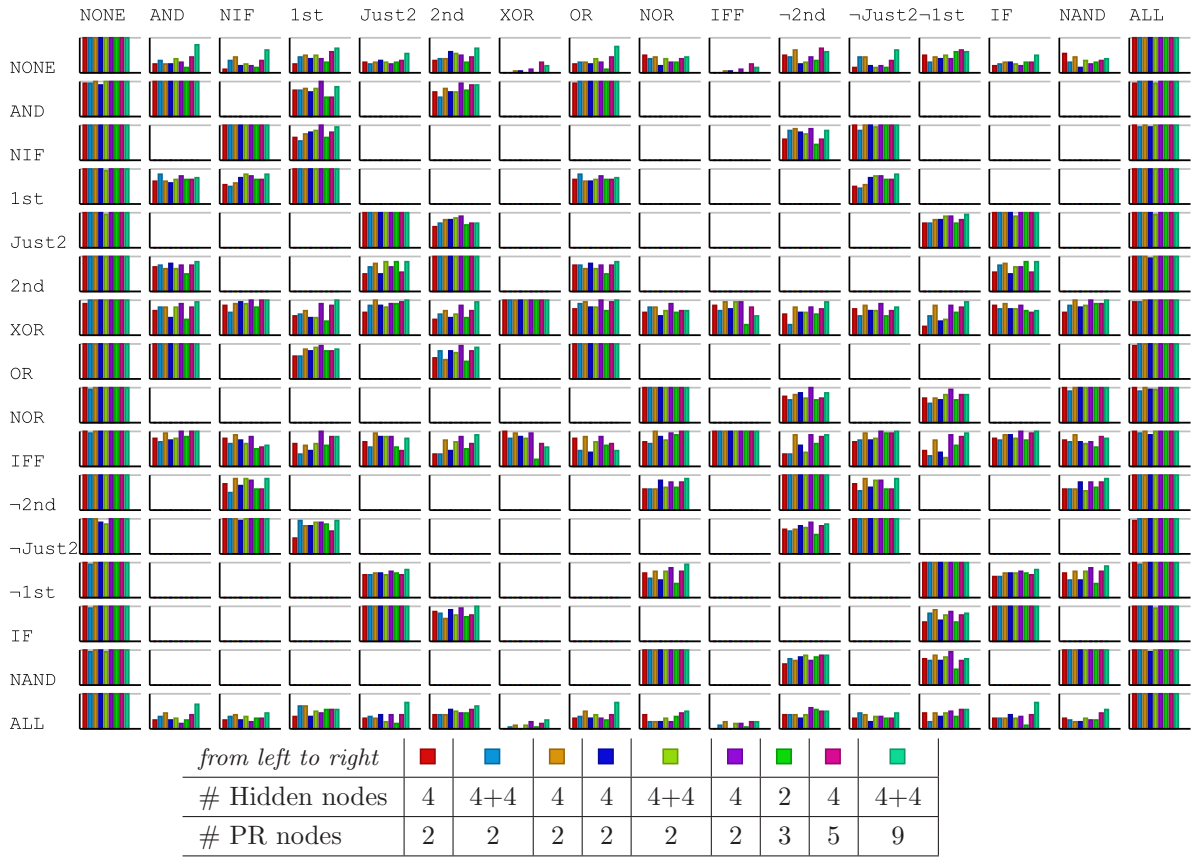


Figure B-10: This figure shows the performance of some networks that allowed multi-layer spanning connections.

Extra PRWs

Network Type	# Hidden Nodes	PRV Size	Difficulty	Parallel	Similarity	Prodigy	Overall
IPB	4	2	85	98	74	78	41
IPB	4+4	2	80	99	74	70	39
IPB	4	4	97	100	87	97	53
FWIPB	4	2	84	99	77	77	41
FWIPB	4+4	2	83	99	78	72	40
FWIPB	4	4	95	100	90	93	49
EPB	2	3	84	100	74	74	41
EPB	4	5	93	100	89	89	53
EPB	4+4	9	96	100	92	92	59

Table B-11: This table shows the performance of several networks that allowed connections between problem representation nodes and every other node, even in non-adjacent layers.

from left to right									
# Hidden nodes	4	4+4	4	4	4+4	4	2	4	4+4
# PR nodes	2	2	2	2	2	2	3	5	9

Figure B-11: This figure shows the performance of some networks that connected the PRNs to every non-input node.