Incremental Learning by Message Passing in Hierarchical Temporal Memory

Davide Maltoni[1] and Erik M. Rehn[2]

[1] Biometric System Laboratory, DEIS - University of Bologna, Italy
davide.maltoni@unibo.it
[2] Bernstein Center for Computational Neuroscience, Berlin, Germany
erik.m.rehn@gmail.com

Abstract. Hierarchical Temporal Memory is a biologically-inspired framework that can be used to learn invariant representations of patterns. Classical HTM learning is mainly unsupervised and once training is completed the network structure is frozen, thus making further training quite critical. In this paper we develop a novel technique for HTM (incremental) supervised learning based on error minimization. We prove that error backpropagation can be naturally and elegantly implemented through native HTM message passing based on Belief Propagation. Our experimental results show that a two stage training composed by unsupervised pre-training + supervised refinement is very effective. This is in line with recent findings on other deep architectures.

Keywords: HTM, Deep architectures, Backpropagation, Incremental learning.

1 Introduction

Hierarchical Temporal Memory (HTM) is a biologically-inspired pattern recognition framework fairly unknown to the research community [1]. It can be conveniently framed into *Multi-stage Hubel-Wiesel Architectures* [2] which is a specific subfamily of Deep Architectures [3-4]. HTM tries to mimic the feed-forward and feedback projections thought to be crucial for cortical computation. Bayesian Belief Propagation is used in a hierarchical network to learn invariant spatio-temporal features of the input data and theories exist to explain how this mathematical model could be mapped onto the cortical-thalamic anatomy [5-6]. A comprehensive description of HTM architecture and learning algorithms is provided in [7], where HTM was also proved to perform well on some pattern recognition tasks, even though further studies and validations are necessary.

One limitation of the classical HTM learning is that once a network is trained it is hard to learn from new patterns without retraining it from scratch. In other words a classical HTM is well suited for a batch training based on a fixed training set, and it cannot be effectively trained incrementally over new patterns that were initially unavailable. In fact, every HTM level has to be trained individually, starting from the bottom: altering the internal node structure at one network level (e.g. coincidences, groups) would invalidate the results of the training at higher levels. In principle, incremental learning could be carried out in a classical HTM by updating only the output level, but this is a naive strategy that works in practice only if the new incoming patterns are very similar to the existing ones in terms of "building blocks". Since incremental training is a highly desirable property of a learning system, we were motivated to investigate how HTM framework could be extended in this direction.

In this paper we present a two-stage training approach, unsupervised pre-training + supervised refinement, that can be used for incremental learning: a new HTM is initially pre-trained (batch), then its internal structure is incrementally updated as new labeled samples become available. This kind of unsupervised pre-training and supervised refinement was recently demonstrated to be successful for other deep architectures [3]. The basic idea of our approach is to perform the batch pre-training using the algorithms described in [7] and then fix coincidences and groups throughout the whole network; then, during supervised refinement adapt the elements of the probability matrices $\mathbf{PCW}$ (for the output node) and $\mathbf{PCG}$ (for the intermediate nodes) as if they were the weights of a MLP neural network trained with backpropagation. To this purpose we derived the update rules based on the descent of the error function. Since the HTM architecture is more complex than MLP, the resulting equations are not simple; further complications arise from the fact that $\mathbf{PCW}$ and $\mathbf{PCG}$ values are probabilities and need to be normalized after each update step. Fortunately we found a surprisingly simple (and computationally light) way to implement the whole process through native HTM message passing. Our initial experiments show very promising results. Furthermore, the proposed two-stage approach not only enables incremental learning, but is also helpful for keeping the network complexity under control, thus improving the framework scalability.

2 Background

An HTM has a hierarchical tree structure. The tree is built up by n_{levels} levels (or layers), each composed of one or more nodes. A node in one level is bidirectionally connected to one or more nodes in the level above and the number of nodes in each level decreases as we ascend the hierarchy. The lowest level, $\mathcal{L}_0$, is the input level and the highest level, $\mathcal{L}_{n_{levels}-1}$, with typically only one node, is the output level. Levels and nodes in between input and output are called intermediate levels and nodes. When an HTM is used for visual inference, as is the case in this study, the input level typically has a retinotopic mapping of the input. Each input node is connected to one pixel of the input image and spatially close pixels are connected to spatially close nodes. Refer to Figure 1 of [10] for a graphical example of HTM.

2.1 Information Flow

In an HTM the flow of information is bidirectional. Belief propagation is used to pass messages/information both up (feed-forward) and down (feedback) the hierarchy as new evidence is presented to the network. The notation used here for belief propagation (Figure 1.a) closely follows Pearl [8] and is adapted to HTMs by George [9]:

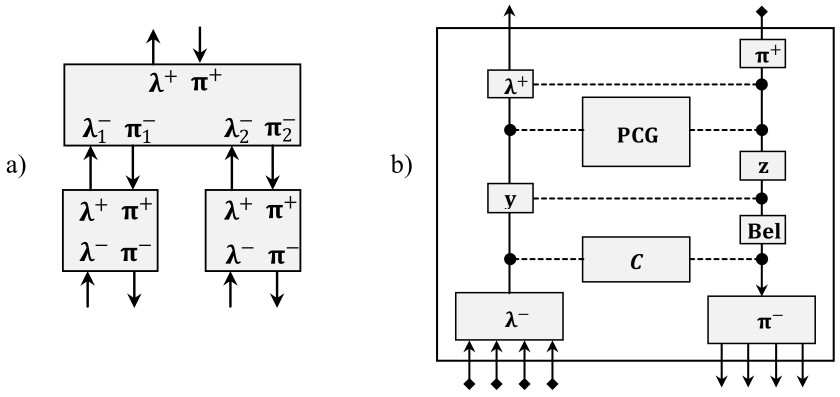


Fig. 1. a) Notation for message passing between HTM nodes. b) Graphical representation of the information processing within an intermediate node.

- Evidence coming from below is denoted e^-. In visual inference this is an image or video frame presented to level $\mathcal{L}_0$ of the network.
- Evidence from the top is denoted e^+ and can be viewed as contextual information. This can for instance be input from another sensor modality or the absolute knowledge given by the supervisor training the network.
- Feed-forward messages passed up the hierarchy are denoted $\boldsymbol{\lambda}$ and feedback messages flowing down are denoted $\boldsymbol{\pi}$.
- Messages entering and leaving a node from below are denoted $\boldsymbol{\lambda}^-$ and $\boldsymbol{\pi}^-$ respectively, relative to that node. Following the same notation as for the evidence, messages entering and leaving a node from above are denoted $\boldsymbol{\lambda}^+$ and $\boldsymbol{\pi}^+$.

When the purpose of an HTM is that of a classifier, the feed-forward message of the output node is the posterior probability that the input e^- belongs to one of the problem classes. We denoted this posterior as $P(w_i|e^-)$, where w_i is one of n_w classes.

2.2 Internal Node Structure and Pre-training

HTM training is performed level by level, starting from the first intermediate level. The input level does not need any training, it just forwards the input. Intermediate levels training is unsupervised and the output level training is supervised. For a detailed description, including algorithm pseudocode, the reader should refer to [7].

For every intermediate node (Figure 1.b), a set $\boldsymbol{C}$, of so called coincidence-patterns (or just coincidences) and a set, $\boldsymbol{G}$, of coincidence groups, have to be learned. A coincidence, $\mathbf{c}_i$, is a vector representing a prototypical activation pattern of the node's children. For a node in $\mathcal{L}_1$, with input nodes as children, this corresponds to an image patch of the same size as the node's receptive field. For nodes higher up in the hierarchy, with intermediate nodes as children, each element of a coincidence, $\mathbf{c}_i[h]$, is the index of a coincidence group in child h. Coincidence groups, also called temporal groups, are clusters of coincidences likely to originate from simple variations of the same input pattern. Coincidences found in the same group can be spatially dissimilar but likely to be found close in time when a pattern is smoothly moved through the node's receptive field. By clustering coincidences in this way, exploiting the temporal smoothness of the input, invariant representations of the input space can be learned [9]. The assignment of coincidences to groups within each node is encoded in a probability matrix $\mathbf{PCG}$; each element $PCG_{ji} = P(\mathbf{c}_j|\mathbf{g}_i)$ represents the likelihood that a group, $\mathbf{g}_i$, is activated given a coincidence $\mathbf{c}_j$. These probability values are the elements we will manipulate to incrementally train a network whose coincidences and groups have previously been learned and fixed.

The output node does not have groups but only coincidences. Instead of memorizing groups and group likelihoods it stores a probability matrix $\mathbf{PCW}$, whose elements $PCW_{ji} = P(\mathbf{c}_j|w_i)$ represents the likelihood of class w_i given the coincidence $\mathbf{c}_j$. This is learned in a supervised fashion by counting how many times every coincidence is the most active one (the winner) in the context of each class. The output node also keeps a vector of class priors, $P(w_i)$, used to calculate the final class posterior.

2.3 Feed-Forward Message Passing

Inference in an HTM in conducted through feed-forward belief propagation (see [7]). When a node receives a set of messages from its m children, $\boldsymbol{\lambda}^- = \{\boldsymbol{\lambda}_1^-, \boldsymbol{\lambda}_2^-, \dots, \boldsymbol{\lambda}_m^-\}$, a degree of certainty over each of the n_c coincidence in the node is computed. This quantity is represented by a vector $\mathbf{y}$ and can be seen as the activation of the node coincidences. The degree of certainty over coincidence i is

$$\mathbf{y}[i] = \alpha \cdot p(e^-|\mathbf{c}_i) = \begin{cases} e^{-(\|\mathbf{c}_i - \boldsymbol{\lambda}^-\|^2/\sigma^2)}, & if\ node\ level = 1 \\ \prod_{j=1}^{m} \boldsymbol{\lambda}_j^-[\mathbf{c}_i[j]], & if\ node\ level > 1 \end{cases} \tag{1}$$

where α is a normalization constant, and σ is a parameter controlling how quickly the activation level decays when $\boldsymbol{\lambda}^-$ deviates from $\mathbf{c}_i$.

If the node is an intermediate node, it then computes its feed-forward message $\boldsymbol{\lambda}^+$ which is a vector of length n_g and is proportional to $p(e^-|\boldsymbol{G})$, where $\boldsymbol{G}$ is the set of all coincidence groups in the node and n_g the cardinality of $\boldsymbol{G}$. Each component of $\boldsymbol{\lambda}^+$ is

$$\boldsymbol{\lambda}^+[i] = \alpha \cdot p(e^-|\mathbf{g}_i) = \sum_{j=1}^{n_c} PCG_{ji} \cdot \mathbf{y}[j] \tag{2}$$

where n_c is the number of coincidences stored in the node.

The feed-forward message from the output node, the network output, is the posterior class probability and is computed in the following way:

$$\boldsymbol{\lambda}^+[c] = P(w_c|e^-) = \alpha \cdot \sum_{j=1}^{n_c} PCW_{jc} \cdot P(w_c) \cdot \mathbf{y}[j] \tag{3}$$

where α is a normalization constant such that $\sum_{c=1}^{n_w} \boldsymbol{\lambda}^+[c] = 1$.

2.4 Feedback Message Passing

The top-down information flow is used to give contextual information about the observed evidence. Each intermediate node fuses top-down and bottom-up information to consolidate a posterior belief in its coincidence-patterns [9]. Given a message from the parent, $\boldsymbol{\pi}^{+}$, the top-down activation of each coincidence, $\mathbf{z}$, is

$$\mathbf{z}[i] = \alpha \cdot P(\mathbf{c}_i | e^{+}) = \sum_{k=1}^{n_g} PCG_{ik} \cdot \frac{\boldsymbol{\pi}^{+}[k]}{\boldsymbol{\lambda}^{+}[k]} \tag{4}$$

The belief in coincidence i is then given by:

$$\mathbf{Bel}[i] = \alpha \cdot P(\mathbf{c}_i | e^{-}, e^{+}) = \mathbf{y}[i] \cdot \mathbf{z}[i] \tag{5}$$

The message sent by an intermediate node (belonging to a level $\mathcal{L}_h, h > 1$) to its the children, $\boldsymbol{\pi}^{-}$, is computed using this belief distribution. The i^{th} component of the message to a specific child node is

$$\boldsymbol{\pi}^{-}[i] = \sum_{j=1}^{n_c} I_{\mathbf{c}_j}\left(\mathbf{g}_i^{(child)}\right) \cdot \mathbf{Bel}[j] = \sum_{j=1}^{n_c} \sum_{k=1}^{n_g} I_{\mathbf{c}_j}\left(\mathbf{g}_i^{(child)}\right) \cdot \mathbf{y}[j] \cdot PCG_{jk} \cdot \frac{\boldsymbol{\pi}^{+}[k]}{\boldsymbol{\lambda}^{+}[k]} \tag{6}$$

where $I_{\mathbf{c}_j}(\mathbf{g}_i^{(child)})$ is the indicator function defined as

$$I_{\mathbf{c}_j}\left(\mathbf{g}_i^{(child)}\right) = \begin{cases} 1, & if\ group\ \mathbf{g}_i^{(child)}\ is\ part\ of\ \mathbf{c}_j \\ 0, & otherwise \end{cases} \tag{7}$$

The top-down message sent from the output node is computed in a similar way:

$$\boldsymbol{\pi}^{-}[i] = \sum_{c=1}^{n_w} \sum_{j=1}^{n_c} I_{\mathbf{c}_j}\left(\mathbf{g}_i^{(child)}\right) \cdot \mathbf{y}[j] \cdot PCW_{jc} \cdot P(w_c | e^{+}) \tag{8}$$

Equation 6 and 8 will be important when we, in the next section, show how to incrementally update the **PCG** and **PCW** matrices to produce better estimates of the class posterior given some evidence from above.

3 Htm Supervised Refinement

This section introduces a novel way to optimize an already trained HTM. The algorithm, called HSR (Htm Supervised Refinement) shares many features with traditional backpropagation used to train multilayer perceptrons and is inspired by weight fine-tuning methods applied to other deep belief architectures [3]. It exploits the belief propagation equations presented above to propagate an error message from the output node back through the network. This enables each node to locally update its internal probability matrix in a way that minimizes the difference between the estimated class posterior of the network and the posterior given from above, by a supervisor.

Our goal is to minimize the expected quadratic difference between the network output posterior given the evidence from below, e^{-}, and the posterior given the evidence from above, e^{+}. For this purpose we employ empirical risk minimization resulting in the following loss function:

$$L(e^{-}, e^{+}) = \frac{1}{2} \sum_{c=1}^{n_w} \left(P(w_c | e^{+}) - P(w_c | e^{-})\right)^2 \tag{9}$$

where n_w is the number of classes, $P(w_c|e^+)$ is the class posterior given the evidence from above, and $P(w_c|e^-)$ is the posterior produced by the network using the input as evidence (i.e., inference). The loss function is also a function of all network parameters involved in the inference process. In most cases e^+ is a supervisor with absolute knowledge about the true class w_{c^*}, thus $P(w_{c^*}|e^+) = 1$.

To minimize the empirical risk we first find the direction in which to alter the node probability matrices to decrease the loss and then apply gradient descent.

3.1 Output Node Update

For the output node which does not memorize coincidence groups, we update probability values stored in the **PCW** matrix, through the gradient descent rule:

$$PCW'_{ks} = PCW_{ks} - \eta \frac{\partial L}{\partial PCW_{ks}} \qquad k = 1..n_c,\ s = 1..n_w \tag{10}$$

where η is the learning rate. The negative gradient of the loss function is given by:

$$-\frac{\partial L}{\partial PCW_{ks}} = -\frac{1}{2}\sum_{c=1}^{n_w} \frac{\partial}{\partial PCW_{ks}} \left(P(w_c|e^+) - P(w_c|e^-)\right)^2 =$$

$$= \sum_{c=1}^{n_w} \left(P(w_c|e^+) - P(w_c|e^-)\right) \frac{\partial P(w_c|e^-)}{\partial PCW_{ks}}$$

which can be shown (see Appendix A of [10] for a derivation) to be equivalent to:

$$-\frac{\partial L}{\partial PCW_{ks}} = \mathbf{y}[k] \cdot Q(w_s) \tag{11}$$

$$Q(w_s) = \frac{P(w_s)}{p(e^-)}\left(P(w_s|e^+) - P(w_s|e^-) - \sum_{i=1}^{n_w} P(w_i|e^-)(P(w_i|e^+) - P(w_i|e^-))\right) \tag{12}$$

where $p(e^-) = \sum_{i=1}^{n_w}\sum_{j=1}^{n_c} \mathbf{y}[j] \cdot PCW_{ji} \cdot P(w_i)$. We call $Q(w_s)$ the error message for class w_s given some top-down and bottom-up evidence.

3.2 Intermediate Nodes Update

For each intermediate node we update probability values in the **PCG** matrix, through the gradient descent rule:

$$PCG'_{pq} = PCG_{pq} - \eta \frac{\partial L}{\partial PCG_{pq}} \qquad p = 1..n_c,\ q = 1..n_g \tag{13}$$

For intermediate nodes at level $\mathcal{L}_{n_{levels}-2}$ (i.e., the last but the output level) it can be shown (Appendix B of [10]) that:

$$-\frac{\partial L}{\partial PCG_{pq}} = \mathbf{y}[p] \cdot \frac{\boldsymbol{\pi}_Q^+[q]}{\boldsymbol{\lambda}^+[q]} \tag{14}$$

where $\boldsymbol{\pi}_Q^+$ is the child portion of the message $\boldsymbol{\pi}_Q^-$ sent from the output node to its children, but with $Q(w_s)$ replacing the posterior $P(w_s|e^+)$ (compare Eqs. 15 and 8):

$$\boldsymbol{\pi}_Q^-[q] = \sum_{c=1}^{n_w} \sum_{j=1}^{n_c} I_{\mathbf{c}_j}\left(\mathbf{g}_q^{(child)}\right) \cdot \mathbf{y}[j] \cdot PCW_{jc} \cdot Q(w_c) \tag{15}$$

Finally, it can be shown that this generalizes to all levels of an HTM, and that all intermediate nodes can be updated using messages from their immediate parent. The derivation can be found in Appendix C of [10]. In particular, the error message from an intermediate node (belonging to a level $\mathcal{L}_h, h > 1$) to its child nodes is given by:

$$\boldsymbol{\pi}_Q^-[q] = -\sum_{t=1}^{n_c}\sum_{f=1}^{n_g} I_{\mathbf{c}_j}\left(\mathbf{g}_q^{(child)}\right) \cdot PCG_{tf} \cdot \frac{\partial L}{\partial PCG_{tf}} = \sum_{t=1}^{n_c}\sum_{f=1}^{n_g} I_{\mathbf{c}_t}\left(\mathbf{g}_q^{(child)}\right) \cdot PCG_{tf} \cdot \mathbf{y}[t] \cdot \frac{\boldsymbol{\pi}_Q^+[f]}{\boldsymbol{\lambda}^+[f]} \tag{16}$$

These results allow us to define an efficient and elegant way to adapt the probabilities in an already trained HTM using belief propagation equations.

3.3 HSR Pseudocode

A *batch* version of HSR algorithm is provided hereafter.

```
HSR( $\mathcal{S}$ )
{  for each training example in $\mathcal{S}$
   {  Present the example to the network and do inference   (eqs. 1,2,3)
      Accumulate $\frac{\partial L}{\partial PCW_{ks}}$ values for the output node         (eqs. 11,12)
      Compute the error message $\boldsymbol{\pi}_Q^-$                               (eq. 15)
      for each child of the output node:
            call BackPropagate(child, $\boldsymbol{\pi}_Q^+$)       (see function below)
   }
   Update PCW by using accumulated $\frac{\partial L}{\partial PCW_{ks}}$                     (eq. 10)
   Renormalize PCW such that for each class $w_s$, $\sum_{k=1}^{n_c} PCW_{ks} = 1$
   for each intermediate node
   {  Update PCG by using accumulated $\frac{\partial L}{\partial PCG_{pq}}$                  (eq. 13)
      Renormalize PCG such that for each group $\mathbf{g}_q$, $\sum_{p=1}^{n_c} PCG_{pq} = 1$
   }
}

function BackPropagate(node, $\boldsymbol{\pi}_Q^+$)
{  Accumulate $\frac{\partial L}{\partial PCG_{pq}}$ values for the node                      (eq. 14)
   if (node level > 1)
      {  Compute the error message $\boldsymbol{\pi}_Q^-$                            (eq. 16)
         for each child of node:
            call BackPropagate(child, $\boldsymbol{\pi}_Q^+$)
      }
}
```

By updating the probability matrices for every training example, instead of at the end of the presentation of a group of patterns, an online version of the algorithm is obtained. Both batch and online versions of HSR are investigated in the experimental section. In many cases it is preferable for the nodes in lower intermediate levels to share memory, so called node sharing [7]. This speeds up training and forces all the nodes of the level to respond in the same way when the same stimulus is presented at different places in the receptive field. For a level operating in node sharing, **PCG** update (eq. 13) must be performed only for the master node.

4 Experiments

To verify the efficacy of the HSR algorithm we performed a number of experiments on the SDIGIT dataset [7]. SDIGIT patterns (16×16 pixels, grayscale images) are generated by geometric transformations of prototypes called primary patterns. The possibility of randomly generating new patterns makes this dataset suitable for evaluating incremental learning algorithms. By varying the amount of scaling and rotation we can also control the problem difficulty.

With $\mathcal{S}_{sdigit\,Train}\langle n, s_{xmin}, s_{xmax}, s_{ymin}, s_{ymax}, r_{max}\rangle$ we denote a set of n patterns, including, for each of the 10 digits, the primary pattern and further $(n/10)-1$ patterns generated by simultaneous scaling and rotation of the primary pattern according to random triplets $\langle s_x, s_y, r\rangle$, $s_x \in [s_{xmin}, s_{xmax}]$, $s_y \in [s_{ymin}, s_{ymax}]$, $r \in [-r_{max}, r_{xmax}]$. The creation of a test set $\mathcal{S}_{sdigit\,Test}\langle n, s_{xmin}, s_{xmax}, s_{ymin}, s_{ymax}, r_{max}\rangle$ starts by translating each of the 10 primary pattern at all positions that allow it to be fully contained in the 16×16 window thus obtaining m patterns; then, for each of the m patterns, $(n/10)-1$ further patterns are generated by transforming the pattern according to random triplets $\langle s_x, s_y, r\rangle$; the total number of patterns in the test set is then $m \times n/10$. Examples of generated patterns are shown in Figure 2.

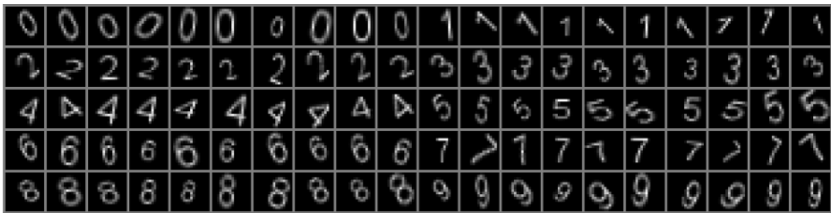


Fig. 2. Example of SDIGIT patterns. Ten patterns for every class are shown.

Table 1 (reprinted from [7]) summarizes HTM performance on the SDIGIT problem and compares it against other well know classification approaches. HTM accuracy is 71.37%, 87.56% and 94.61% with 50, 100 and 250 training patterns, respectively: our goal is to understand if accuracy can be improved by HSR incrementally training. To this purpose we follow the procedure described below:

```
Generate a pre-training dataset $\mathcal{S}_0 = \mathcal{S}_{sdigitTrain} < n, 0.70, 1.0, 0.7, 1.0, 40° >$
Pre-train a new HTM on $\mathcal{S}_0$       (as in [7], leading to Table 1 results)
for each epoch $E_i, i = 1..n_e$
{  Generate a dataset $\mathcal{S}_i = \mathcal{S}_{sdigitTest} < 1000, 0.60, 1.1, 0.6, 1.1, 45° >$   (6,200 patterns)
   Test HTM on $\mathcal{S}_i$
   for each iteration $I_i, i = 1..n_i$
      call HSR($\mathcal{S}_i$)
}
Test HTM on $\mathcal{S}_{n_e}$
```

Table 1. HTM compared against other techniques on SDIGIT problem (reprinted from [7]). Three experiments are performed with an increasing number of training patterns: 50, 100 and 250. The test set is common across the experiments and include 6,200 patterns. NN, MLP and LeNet5 refer to Nearest Neighbor, Multi-Layer Perceptron and Convolutional Network, respectively. HTM refers to a four-level HTM (whose architecture is shown in Figure 1 of [10]).

SDIGIT - test set: $\mathcal{S}_{sdigit\,Test}\langle 1000,0.60,1.10,0.60,1.10,45°\rangle$ (6,200 patterns, 10 classes)						
Training set	Approach	Accuracy (%)		Time (hh:mm:ss)		Size (MB)
		train	test	train	test	
$\mathcal{S}_{sdigitTrain}$ <50,0.70,1.0,0.7,1.0,40°> 1788 translated patterns	NN	100	57.92	< 1 sec	00:00:04	3.50
	MLP	100	61.15	00:12:42	00:00:03	1.90
	LeNet5	100	67.28	00:07:13	00:00:11	0.39
	HTM	100	71.37	00:00:08	00:00:13	0.58
$\mathcal{S}_{sdigitTrain}$ <100,0.70,1.0,0.7,1.0,40°> 3423 translated patterns	NN	100	73.63	< 1 sec	00:00:07	6.84
	MLP	100	75.37	00:34:22	00:00:03	1.90
	LeNet5	100	79.31	00:10:05	00:00:11	0.39
	HTM	100	87.56	00:00:25	00:00:23	1.00
$\mathcal{S}_{sdigitTrain}$ <250,0.70,1.0,0.7,1.0,40°> 8705 translated patterns	NN	100	86.50	< 1 sec	00:00:20	17.0
	MLP	99.93	86.08	00:37:32	00:00:03	1.90
	LeNet5	100	89.17	00:14:37	00:00:11	0.39
	HTM	100	94.61	00:02:04	00:00:55	2.06

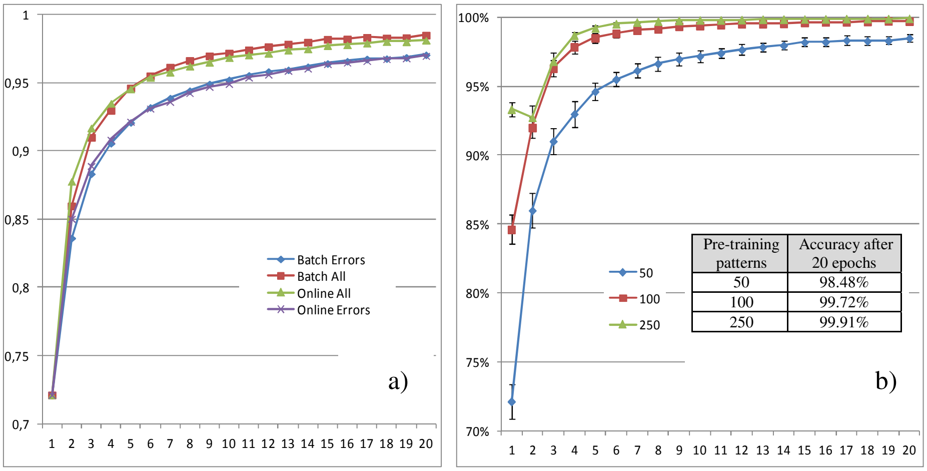
In our experimental procedure we first pre-train a new network using a dataset $\mathcal{S}_0$ (with n patterns) and then for a number of epochs we generate new datasets $\mathcal{S}_i$ and apply HSR. At each epoch one can apply HSR for more iterations, to favor convergence. However, we experimentally found that a good trade-off between convergence time and overfitting can be achieved by performing just two HSR iterations for each epoch. The classification accuracy is calculated using the patterns generated for every epoch but before the network is updated using those patterns. In this way we emulate a situation where the network is trained on sequentially arriving patterns.

4.1 Training Configurations

We assessed the efficacy of the HSR algorithm for different configurations:

- *batch* vs *online* updating: see Section 3.3;
- *errors* vs *all* selection strategy: in *errors* selection strategy, supervised refinement is performed only for $\mathcal{S}_i$ patterns that were misclassified by the current HTM, while in *all* selection strategy is performed over all $\mathcal{S}_i$ patterns;
- learning rate η: see Equations 10 and 13. One striking find of our experiments is that the learning rate for the output node should be kept much lower than for the intermediate nodes. In the following we refer to the learning rate for output node as η_o and to the learning rate for intermediate nodes as η_i. We experimentally found that optimal learning rates (for SDIGIT problem) are $\eta_o = 0.00005$ and $\eta_i = 0.0030$.

Figure 3.a shows the accuracy achieved by HSR over 20 epochs of incremental learning, starting from an HTM pre-trained with $n = 50$ patterns. Accuracy at epoch 1 corresponds to the accuracy after pre-training, that is about 72%. A few epochs of HSR training are then sufficient to raise accuracy to 93-95%. The growth then slow down and, after 20 epochs, the network accuracy is in the range [97.0-98.5%] for the different configurations. It is worth remembering that the accuracy reported for each epoch is always measured on unseen data.



Pre-training patterns	Accuracy after 20 epochs
50	98.48%
100	99.72%
250	99.91%

Fig. 3. a) HSR accuracy over 20 epochs for different configurations, starting with an HTM pre-trained with $n = 50$ patterns. Each point is the average of 20 runs. b) HSR accuracy over 20 epochs when using an HTM pre-trained with 50, 100 and 250 patterns. HSR configuration is *batch*, *all*. Here too HSR is applied two times per epoch. Each point is the average of 20 runs. 95% mean confidence intervals are plotted.

Training over all the patterns (with respect to training over misclassified patterns only) provides a small advantage (1-2 percentage). Online update seems to yield slightly better performance during the first few epochs, but then accuracy of online and batch update is almost equivalent. Table 2 compares computation time across different configurations.

Table 2. HSR computation times (averaged over 20 epochs). Time values refer to our C# (.net) implementation under Windows 7 on a Xeon CPU W3550 at 3.07 GHz.

Configuration	HSR time 6200 patterns - 1 iteration	HSR time 1 pattern - 1 iteration
Batch, All	19.27 sec	3.11 ms
Batch, Error	8,37 sec	1.35 ms
Online, All	22.75 sec	3.66 ms
Online, Error	8.27 sec	1.34 ms

Applying supervised refinement only to misclassified patterns significantly reduces computation time, while switching between *batch* and *online* configurations is not relevant for efficiency. So, considering that accuracy of the *errors* strategy is not far from the *all* strategy we recommend the *errors* configurations when an HTM has to be trained over a large dataset of patterns.

4.2 HTM Scalability

One drawback of the current HTM framework is scalability: in fact, the network complexity considerably increases with the number and dimensionality of training patterns. All the experiments reported in [7] clearly show that the number of coincidences and groups rapidly increases with the number of patterns in the training sequences. Table 3 shows the accuracy and the total number of coincidences and groups in a HTM pre-trained with an increasing number of patterns: as expected, accuracy increases with the training set size, but after 250 patterns the accuracy improvement slows down while the network memory (coincidences and group) continues to grow markedly, leading to bulky networks. Figure 3.b shows the accuracy improvement by HSR (*batch*, *all* configuration) for HTMs pre-trained over 50, 100 and 250 patterns. It is worth remembering that HSR does not alter the number of coincidences and groups in the pre-trained network, therefore the complexity after any number of epochs is the same for all the pre-trained HTMs (refer to Table 3). It is interesting to see that HTMs pre-trained with 100 and 250 patterns after about 10 epochs reach an accuracy close to 100%, and to note that even a simple network (pre-trained on 50 patterns) after 20 epochs of supervised refinement outperforms an HTM with more than 10 times its number of coincidences and groups (last row of Table 3).

Table 3. Statistics after pre-training. The first three rows are consistent with Table III of [7].

Number of pre-training patterns	Accuracy after pre-training	Coincidence and groups
50	71.37%	7193, 675
100	87.56%	13175, 1185
250	94.61%	29179, 2460
500	93.55%	53127, 4215
750	96.97%	73277, 5569
1000	97.44%	92366, 6864

5 Discussion and Conclusions

In this paper we propose a new algorithm for incrementally training HTM with sequentially arriving data. It is computationally efficient and easy to implement due to its close connection to the native belief propagation message passing of HTM.

The term $Q(w_s)$, the error message send from above to the output node (Eq. 12), is the information that is propagated back through the network and lies at the heart of the algorithm. Its interpretation is not obvious: the first part, $P(w_s|e^+) - P(w_s|e^-)$, the difference between the ground truth and network posterior, is easy to understand; while the second part, $-\sum_{i=1}^{n_w} P(w_i|e^-)(P(w_i|e^+) - P(w_i|e^-))$, is more mysterious. It is hard to give a good interpretation of this sum but from our understanding it arises due to the fact that we are dealing with probabilities. None of the parts can be ignored; tests have shown that they are both important to produce good results.

There are some parameters which need tuning to find the optimal setup. In the experiments presented in this paper two iterations per epoch were used, and the optimal learning rate was found therefor. With more iterations a lower learning rate would likely be optimal. The difference in suitable learning rate between the intermediate and the output level is also an important finding and can probably be explained by the fact that the $\mathbf{PCW}$ matrix of the output node has a much more direct influence on the network posterior. The output node memory is also trained supervised in the pre-training while the intermediate nodes are trained unsupervised, which might suggest that there is more room for fine tuning in the intermediate nodes. We ran some experiments where we only updated $\mathbf{PCW}$ in the output node: in this case a small performance gain of a few percent has been observed.

In general HSR has proven to work very well for the SDIGIT problem and the results give us reason to believe that this kind of supervised fine tuning can be extended to more difficult problems. Future work will focus on the following issues:

- applying HSR to other (more difficult) incremental learning problems;
- check whether, for a difficult problem based on a single training set, splitting the training set in two or more parts and using one part for initial pre-training and the rest for supervised refinement, can lead to better accuracy and efficiency;
- extending HSR in order to also finely tune (besides $\mathbf{PCW}$ and $\mathbf{PCG}$) the structure of level 1 coincidences $\boldsymbol{C}$ without altering their number. In fact, while higher level coincidences are "discrete feature selectors" and therefore not applicable to continuous gradient descent optimization, level 1 coincidences are continuous features and their adaption could lead to further performance improvement.

References

1. George, D., Hawkins, J.: A Hierarchical Bayesian Model of Invariant Pattern Recognition in the Visual Cortex. In: IJCNN (2005)
2. Ranzato, M., et al.: Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition. In: CVPR (2007)
3. Bengio, Y.: Learning Deep Architectures for AI. Foundations and Trends in Machine Learning 2(1) (2009)
4. Jarrett, K., Kavukcuoglu, K., Ranzato, M., LeCun, Y.: What is the Best Multi-Stage Architecture for Object Recognition? In: ICCV (2009)
5. George, D., Hawkins, J.: Towards a Mathematical Theory of Cortical Micro-circuits. PLoS Computational Biology 5(10) (2009)
6. Lee, T.S., Mumford, D.: Hierarchical Bayesian inference in the visual cortex. Journal of the Optical Society of America 20(7), 1434–1448 (2003)
7. Maltoni, D.: Pattern Recognition by Hierarchical Temporal Memory. DEIS TR (April 2011), http://bias.csr.unibo.it/maltoni/HTM_TR_v1.0.pdf
8. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan-Kaufmann (1988)
9. George, D.: How the Brain Might Work: A Hierarchical and Temporal Model for Learning and Recognition. Ph.D. thesis, Stanford University (2008)
10. Maltoni, D., Rehn, E.M.: Incremental Learning by Message Passing in Hierarchical Temporal Memory. DEIS TR (May 2012), http://bias.csr.unibo.it/maltoni/HTM_HSR_TR_v1.0.pdf