Universiteit Leiden
Opleiding Informatica

Explorative study on

Hierarchical temporal memory

Name:	Alexander Latenko
Studentnr:	S1427539
Date:	15/06/2016
1st supervisor:	Prof. Thomas Bäck
2nd supervisor:	Hao Wang

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Explorative study on hierarchical temporal memory

Alexander Latenko July 16

Abstract

Hierarchical temporal memory (HTM) is a new machine learning technique with its architecture based on the neocortex in mammals. Through its unique architecture the model should fare well in various problem sets. In this thesis, first an inspection was made on the theory behind the HTM model. Secondly, an empirical comparison has been made to other widespread machine learning methods by applying them to a set of regression use cases. The empirical comparison showed a lacking performance from HTM, not coming close to the performance of most other models. HTM is designed for the recognition of sequences and making predictions. The results show that this does not prove useful in regular regression cases, it might prove to be more useful in cases of time based regression. Also it must be noted that HTM is an incomplete model and it is unclear how well the performance will be once the model and the implementation are completed.

Contents

Abstract **i**

1 Introduction **3**

1.1 Thesis Overview . . . 4

2 Theoretical Overview **5**

2.1 Definitions . . . 5

2.1.1 Artificial Neural Network . . . 5

2.1.2 Complexity . . . 6

2.2 Feedforward Neural Networks . . . 6

2.2.1 Hyperparameters . . . 6

2.3 Support Vector Machines . . . 7

2.3.1 Hyperparameters . . . 7

2.3.2 Overfitting . . . 8

2.4 Gaussian Processes . . . 8

2.4.1 Hyperparameters . . . 9

2.4.2 Limitations . . . 10

2.5 Random Forests . . . 10

2.6 Convolutional Neural Networks . . . 11

2.6.1 Hyperparameters . . . 12

3 Hierarchical temporal memory **13**

3.1 Neuron Structure . . . 13

3.2 Region Structures . . . 16

3.2.1 Sparse Distributed Representations . . . 16

3.2.2 Context Prediction . . . 16

3.3 Spatial Pooling . . . 17

3.3.1 Goals . . . 17

3.3.2 Implementation . . . 17
3.4 Temporal Pooling . . . 18
3.4.1 Implementation . . . 19
3.5 Parameters . . . 20

4 Empirical Investigation **21**
4.1 Specifications . . . 21
4.2 Datasets . . . 22
4.3 Model Tuning . . . 22
4.3.1 Sklearn . . . 23
4.3.2 Swarming . . . 23
4.4 Code . . . 23

5 Results **24**
5.1 Speed . . . 24
5.2 Structure . . . 25
5.3 Discussion . . . 26

6 Conclusions **28**

A Model Parameters **29**

B Code **31**
B.1 Sci-kit models . . . 31
B.2 Feedforward Net . . . 32
B.3 Convolutional Neural Network . . . 32
B.4 HTM . . . 33

Bibliography **34**

List of Tables

5.1 Performance results on airfoil dataset 25
5.2 Performance results on histogram dataset 26
5.3 Performance results on crime dataset 26

List of Figures

2.1 SVM input space transformation 8
2.2 Overfitting graph 8
2.3 Gaussian process 10
2.4 Convolutional neural network 12

3.1 Neurons 14
3.2 HTM learning process 15

Chapter 1

Introduction

Several types of artificial intelligence models have been developed and are being applied in different fields, including transportation, surveillance, customer service and the law to name a few. These models often share the common property of being higly specialised. Hierarchical Temporal Memory is a fairly recent model that distinguishes itself in design purpose by being a general intelligence model.

HTM is a biologically inspired machine intelligence model that is closely based on the architecture and the processes of the neocortex. The neocortex is capable of incredible feats when it comes to simple tasks like image recognition, language processing and spatial reasoning but also provides a great flexibility allowing it to switch, seemingly without any trouble, between these tasks. Being based on the neocortex allows HTM to be applied in various problem spaces. A design with the goal of generalization and flexibility is considered a generalized intelligence by Jeff Hawkins, the creator of HTM [10]. A comparison made is that deep blue could beat Gary Kasparov, the world chess champion, in a game of chess. The computer was said to play chess but not to understand chess. It did not have the intuition of the way the board progresses nor how his opponent tends to play. It only used its computational speed to explore a myriad of options more than its human opponent.

Some emprirical research has been done on the performance of HTM on specific problem spaces. In some cases comparisons have been made to other methods. For an object recognition use-case it has been found that HTM performs significantly better than a standard Support Vector Machine (SVM) classifier [13]. Other research has produced accuracy results for classification, with a variance in the results over a great range [8]. Very little has been done with HTM on regression. A single study found the HTM model performing worse than a linear regression model for a dataset [14].

The model shares many properties with traditional neural networks. It consists of a hierarchy containing simple units or subregions doing the same operation. These units are capable of running in parallel. The

system is capable of producing correct stimulus response reaction through training and is able to generalize [3]. HTM uses many traditional ideas in its model but distincts itself by forming it into a temporal recognizer of sequences with lateral inference. In this thesis we are looking at how this HTM memory prediction framework works conceptually and how it differs in comparison to some other methods, most notably convolutional neural networks. Further, we are going to proceed with testing the performance of HTM empirically compared to some other methods, elaborating the results with the conceptual theory, this will be done for several regression use-cases.

1.1 Thesis Overview

Chapter 2 starts of with the definitions explaining some basic machine learning knowledge together with the theory behind the machine learning methods used to compare to HTM. Chapter 3 will look at the theory behind HTM and how these ideas originated. Further a very general idea is given for the implementation of the learning algorithms used in HTM. Chapter 4 will describe the research setup and the datasets used for the investigation. Lastly, we have chapters 5 and 6 for the results and conclusions respectively.

Chapter 2

Theoretical Overview

This section contains an overview of the theory behind most of the methods used in this thesis together with some definitions to present a clearer image.

2.1 Definitions

This sections presents some of the basic definitions that are relevant to the methods discussed in this thesis. General explanations are provided for the various definitions, in the practical section more elaboration is given concerning specific theory that has been applied.

2.1.1 Artificial Neural Network

The concept for deep learning methods comes from artificial neural network (ANN) research [7]. ANNs are inspired by the neuronal systems in mammal brains. ANNs are constructed using a few key building blocks. Neurons are the basic processing units of an ANN. These processing units are placed in a layer structure with directed connections between them. The output of a neuron is transmitted through the connection to the next neuron. The output of a neuron is a function of all its input connections and corresponding weights. An ANN infers the correct outputs by having the neurons train with a training dataset and adjusting the weights for the connections through a learning function. Once trained, the combination of neurons receiving an input and producing an output, sending it across layers through their connections, will eventually produce accurate outputs if set up correctly.

2.1.2 Complexity

One of the most used standards for looking at algorithm efficiency is complexity. The theoretical complexity of a problem provides bounds for the amount of time or computations needed for an algorithm to solve the problem. It has been shown that the training of a neural network can't be done with a polynomial time algorithm [18]. Especially when there are no prior definitions about the dataset. The ANNs suffer from combinatorial explosions when trained to a reasonable accuracy. Most algorithms suffer in time complexity but there are a few for which trouble arises in the space complexity, the amount of memory that is needed. There is no general complexity approximation when it comes to the various types of machine learning techniques. This requires one to make estimations on the complexity based on the various types of hyperparameters deployed and the general size of the network structure.

2.2 Feedforward Neural Networks

Feedforward neural networks with many hidden layers or MLPs, often referred to as deep neural networks (DNN), are good examples of early models with a deep architecture. These layers consist of an input layer, one or more hidden layers and an output layer [3]. The first feedforward networks are considered the simplest type of ANNs. These ANNs have a specific acyclic structure, meaning once a node activates it will only send signal to nodes that are closer to the output node, hence the name. Even though this is the most simple type of ANN it is not a simple task to optimize the structure.

2.2.1 Hyperparameters

When it comes to creating any machine learning method the hyperparameters play a significant role in the performance. The hyperparameters vary from algorithm to algorithm. However, the most basic hyperparameters found in feedforward networks can also be found in most other machine learning algorithms.

For a neural network to be able to train itself there needs to be a way for it to check its own performance. This is where the loss or cost function comes into play. The goal of the cost function is to compare the output of the network to the true expected output. The cost value, that is computed using this function, allows the learning algorithms to update the weights of the connections. The rate at which these weights are modified is dependent on the learning rate. A low learning rate will be too slow to adapt, a high learning rate might lead to divergence. Finally, there are the activation functions which let the nodes make a computation based on the given input, providing the next layer of nodes with an input.

2.3 Support Vector Machines

Support vector machines (SVM) are a kernel based method for learning. An SVM takes an input set x and maps it with a transformation function to a higher dimension, this can be seen in Figure 2.1. The model in the feature space can be described as

$$f(x, w) = \sum_{j-1}^{m} w_j g_j(x) + b$$

with b as bias and $g_j(x), j = 1, \ldots, m$ as the set of tranformations. A basic concept for SVMs is the use of ϵ to allow certain deviation from the targets. Meaning that every prediction that is within ϵ range of the target i.e. $y - f(x, w) \leq \epsilon$ will not be considered as a loss. In some cases it is not possible to find a function with less than ϵ deviation from every target. Which is why there are slack variables ξ, ξ^*, allowing there to be a certain amount of targets that do deviate more that ϵ from the function. When large slack variables are chosen the model complexity can be reduced by minimizing $\|w^2\|$. The SVM regression function can be formulized as a minimization:

$$\min_{w,b,\xi,\xi^*} \|w^2\| + C \sum_{j-1}^{n} (\xi + \xi^*) \text{ such that}$$

$$\begin{cases} y_j - f(x_j, w) \leq \epsilon + \xi_j^* \\ f(x_j, w) - y_j \leq \epsilon + \xi_j \\ \xi_j, \xi_j^* \geq 0, j = 1, \ldots, n \end{cases}$$

This forms the basic type of SVM regression where C is derived from the number of samples and the learning rate.

2.3.1 Hyperparameters

Three main parameters are influential in the performance of SVMs. Firstly, we have the ϵ parameter mentioned above that is responsible for the size of the area around the function, in which targets are not considered for a loss or cost. A small ϵ will require a very tight fitting from the funtion to the data points. The C parameter is used for determining what the cost is for having data points outside of the zone formed by the ϵ parameter. So a large C will increase the complexity of the function requiring most of the data points to be encompassed in the ϵ zone. A small C will not have a high cost for data points ouside the zone, allowing for a more generalized solution. Finally the kernel function has to be assigned to an SVM to map the original input space into the feature space see Figure 2.1. The kernel function can be of several types like linear,

polynomial or radial basis functions (RBF), to name a few. The kernel parameters are, like the feedforward network cost function, highly dependent on the type of operation to be performed on the data.

2.3.2 Overfitting

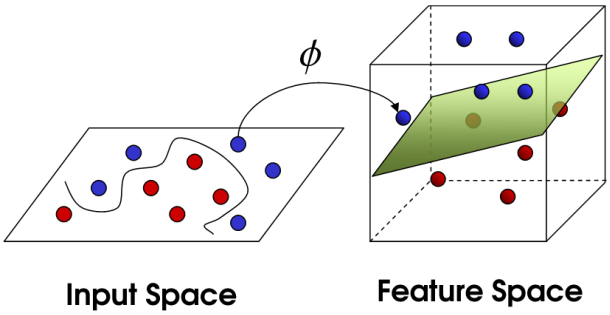


Figure 2.1: An input space for an SVM transformation into a 3-dimensional space allowing the problem to be solved linearly in this case

In order for an SVM to be able to produce good results in regression or classification problems the model, that is going to be used, needs to be fitted to the data. One of the problems that may arise when trying to fit a model to data is that it might be fitted too strongly or overfits to the specific data, making it inflexible for cases that are slightly less similar to the training data.

SVM can suffer from two types of overfitting, in training and in model selection [6]. The training is the most classical type of overfitting and is illustrated in Figure 2.2. This occurs when the parameters are fit too closely to the training set, causing worse performance on the validation set.

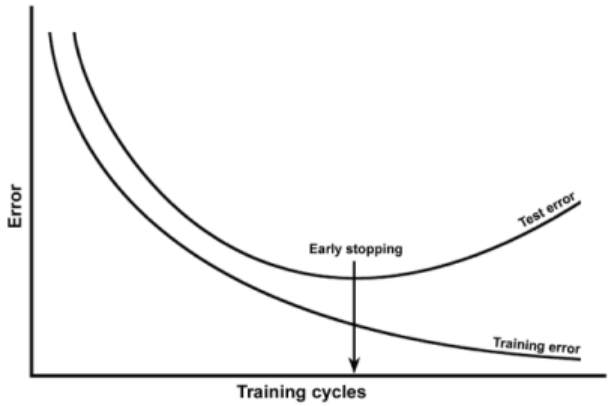


Figure 2.2: A simple graph showing the performance of a machine learning technique after a certain amount of training cycles. As can be seen the, performance is faltering after having done more than a certain amount of cycles. This phenomenon is called overfitting

Overfitting in model selection occurs when the model selection based on a small set of data is directly optimised. This type of overfitting seems to occur the most when the number of hyperparameters is greater in comparison to the size of data used [6]. This problem is especially prevalent in the case of SVM because of the large amount of kernel parameters which are tuneable in the model.

2.4 Gaussian Processes

The idea behind Gaussian processes (GP) has been theorized over for a long time [19]. GP is a Bayesian Network using probabilities to infer a result, allowing a GP to take into account only a finite set of points of a function but still being able to estimate properties of the complete function. However, Gaussian

processes come with two categories of limitations, one being complexity and the other one being the prior requirements that are expected of the data [21].

A GP is defined as a collection of random variables from which any set can be taken which will have a joint gaussian distribution [20]. The GP has two parts that generate the process, a mean and a covariance funtion. The two functions can be used to define gaussian distributions for every point given. A linear covariance function can be writen as:

$$k(x, x') = x^T x'$$

With the use of this function, for every possible combination of the n training records with observation y, a covariance matrix K is created. To calculate the y_* for an input value of x_*. The K_* and K_{**} are also produced with:

$$K_* = [k(x_*, x_1), k(x_*, x_2) \dots k(x_*, x_n)]$$

$$K_{**} = k(x_*, x_*)$$

This leads to a Gaussian distribution of the probability:

$$y_*|y \sim N(K_* K^{-1} K_* y, K_{**} - K_* K^{-1} K_*^T)$$

The best estimate is the mean of the distribution:

$$\overline{y_*} = K_* K^{-1} K_* y$$

and the variance is:

$$var(y_*) = K_{**} - K_* K^{-1} K_*^T)$$

This combines into a simple linear Gaussian process. The figure 2.3 shows an example of a Gaussian process predicting some noisy data. The shaded area is the 95% confidence interval calculated by the variance equation.

2.4.1 Hyperparameters

The hyperparameters of a GP differ greatly from neural networks depending on the model used. Instead of using nodes and layers the model uses covariance matrices and distribution functions. As mentioned above, the covariance function is an essential part for the modelling by a GP. The choice in covariance function brings its own group of hyperparameters. The function represents the view of data, and can take any form of function in order to represent the smoothness and variance in the data. The various parameters can be of any

type depending on the type of function that is chosen and this choice has a significant effect on the final result.

2.4.2 Limitations

One of the problems of GPs is the complexity. For a data set of size n the time complexity becomes $O(n^3)$ making it slower than a lot of methods used for this thesis but this was not the main problem for using this method. The space complexity for storing the whole covariance matrix is $O(n^2)$ requiring an enormous amount of memory for it to be used on larger datasets. Another limitation in the use of GPs is the fact that it assumes that data is to a certain degree distributed as a multivariate Gaussian. Most datasets in this thesis, however, were too big for this too occur [21].

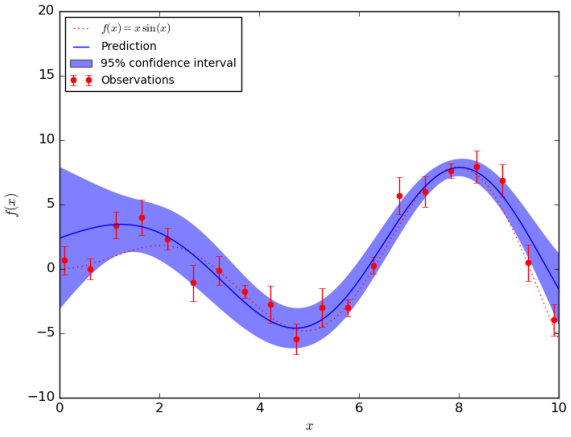


Figure 2.3: An image of a Gaussian process modeling noisy data. The observations are the test samples that were provided for training. Source: Sci-kit

2.5 Random Forests

Random forests is an ensemble method for machine learning. I.e. it derives its conclusions by using the answers provided from multiple learning algorithms, in the case of random forest these algorithms are decision trees. A random forest is a set of decision trees defined by a training set and all the trees have their own random vector with the same distribution. The generalization error for the data converges to a limit as the amount of trees in the forest grows [5]. The convergence of the random forests is one of its greatest attractions and of the limit on the convergence no severe overfitting should occur.

The way random forests come to accurate predictions is by taking the answers provided by the constructed decisions trees and either averaging or taking the median of these results. The training in a random forest works by bagging the decision trees, for a given training set a part is sampled with replacement and used for fitting the trees. This results in the random samples taken to have minimal variance from the mean of the dataset leading to better predictions.

The number of decision trees in random forests has great effect on the computational cost. The complexity for creating a random forest with M trees, N samples, p input variables and K as the number of random features chosen to become children of nodes, the complexity amounts to around $O(0.6MKN^2 \log 0.6N)$ for

the worst case and $\Theta(0.6MKN \log^2 0,6N)$ for the average case, with about 60% unique data sampled because of replacement [15]. The complexity for searching in a forest is significantly less going around $\Theta(MN)$ for the worst case. The amount of computations required thus depends on the hyperparameters like K and M.

2.6 Convolutional Neural Networks

Convolutional neural network (CNN) is a deep learning AI technique. Deep learning techniques are defined as compositions of many layers of adaptive non-linear components [4]. This means that every layer in a deep learning network has its own parameters which can be trained. CNN is the best known method for digit recognition [12]. Convolutional neural networks use a combination of three basic ideas to achieve their results, namely, local receptive fields, shared weights and pooling [17].

The first step a CNN usually performs is the convolution, taking the data and creating local receptive fields within it. This method is specifically good for usage on images, see figure 2.4. The idea of these layers is to divide the original image in a set of overlapping but not identical areas. These areas are mapped to hidden neurons which will only receive input from those areas. The hidden nodes now fullfil the role of a local receptive field, however, the nodes are looking for similar features across the whole image. An eye should be recognized as an eye no matter what part of the image it is placed in. This is where the shared weights come in. The hidden nodes activate when a certain combination of input nodes computed with their weights fullfils its threshold for activation. In order to recognize a feature across the whole image a mapping is required with a single group of weights that is the size of the input of a local receptive field, this is a feature map. Multiple feature maps can be used, allowing the layer to recognize more features.

The CNN alternates between convolution layers and pooling layers. The pooling layers are responsible for simplifying and generalizing the information received from the convolutional layer. One of the techniques for pooling is done by dividing the input nodes in non-overlapping areas and taking the maximum activation from every area, this is known as max-pooling. Other methods are possible, all resulting in creating a more condensed representation of the input as shown in figure 2.4.

Finally the output of the final pooling layer will be send to a fully connected layer which will produce the results, in a fashion not much different from any other ANN. After outputting a result the network will change its weights similar to the way regular ANNs do, with a learning function and possibly a loss value.

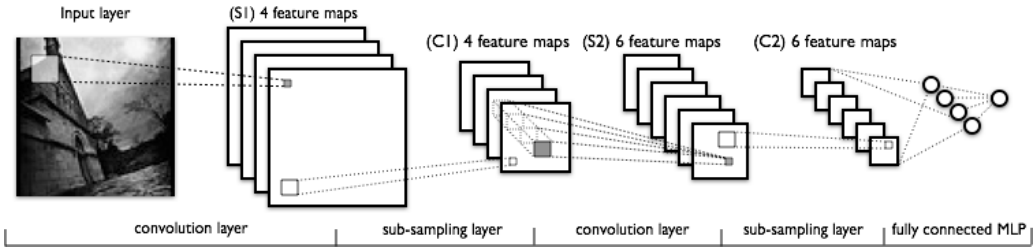


Figure 2.4: An image of a convolutional neural network. Starting with a convolution layer, CNN alternates between convolution and pooling until the eventual results from the final operation is put into a multi-layered perceptron. Image downloaded from <http://deeplearning.net/tutorial/_images/mylenet.pnginaugust2016>

2.6.1 Hyperparameters

The number of parameters available is massive even though this number is greatly reduced by the process of convolution and pooling, the number of combinations remains great. The number of hyperparameters is also no small number. There is the variety in the number of layers and the nature of the final layer. The choice of correct options amongst the myriad of customization possibilities before initializing the program is a task of itself.

Firstly, we have the parameter that can be found in most classical neural networks, the learning rate. As mentioned before, a too high value might cause divergence, while a too value leads to too slow convergence. Similarly, there is also the weight decay and momentum that affect the convergence significantly by influencing the learning patterns of the network in general.

Some parameters affect the behaviour of the layers on their owns aside from the parameters mentioned above. The size of the receptive field is an example, at what level the features are supposed to be analyzed, this size greatly affects at what level of complexity the data is computed in which layer. The pooling size affects the amount of data that is sub-sampled and can also be changed by the user. Choosing large areas of nodes to pool will help with time complexity but might also lead to loss of information.

Chapter 3

Hierarchical temporal memory

To shy away from the concept of having specialized machine learning techniques hierarchical temporal memory (HTM) is created following the structure of the neocortex. This in theory allows HTM to be very flexible in its application. In this section we are taking a deeper look at the theory behind this concept of generalized intelligence presented by Jeff Hawkins [10]. We have to note that the theoretical framework is not complete and as mentioned in the white paper only a subset of the framework is implemented [1]. However, the claim is that this subset already has scientific value [1]. There are many components of the theory that are not yet implemented, including attention, feedback between regions, specific timing, and behavior/sensory-motor integration. These missing components should in the future fit into the framework already created.

The basis of HTM is the same concept of a neural network, consisting of neurons, layers, and connections between those layers. However, HTM expands this concept in a variety of ways combining the ideas of several machine learning techniques into one [16]. HTM, similar to a neural network, is programmed by exposing it to streams of sensory data and retrieving spatial and time based patterns and storing those in memory. This way the identical inputs may be interpreted differently. This is the universal feature of perception and action [1].

3.1 Neuron Structure

The HTM architecture has been influenced a lot by neuroscience. This section serves as an elaboration, for the influence that came from neuroscience, the HTM neuron model and how it differs from classic neurons from ANNs.

The neuron is the simplest and of the most common element of any neural network, be it the neocortex, ANN or HTM. The neuron, as mentioned in chapter 2.1.1, is a basic processing unit which can receive information

from a number of inputs, perform an activation rule and present the output to a number of nodes. The HTM uses a similar structure with each neuron having two components, dendrites and synapses. In figure 3.1 three types of neurons are shown, firstly the simple ANN neuron which has a few synapses (inputs) and no dendrites. The output of the ANN neuron is the weighed sum of these synapses applied to a function. For comparison the HTM and biological neuron are shown side by side. The biological neuron inspired the HTM neuron, the number of synapses in a biological neuron can reach the thousands, and the same goes for the HTM neuron. Learning occurs by forming new synapses and removing obsolete synapes. It is unclear how exactly biological neurons produce output based on the input from synapses and how exactly the dendrites affect this outcome. The HTM neuron is a model based on the known facts of the biological neurons together with some assumptions to recreate the behavior [9].

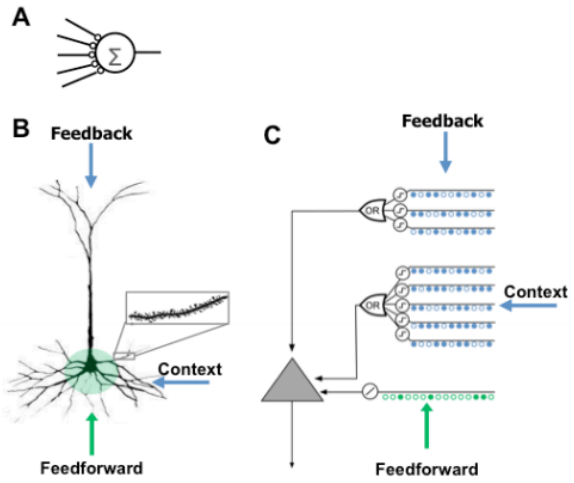


Figure 3.1: Image showing different neurons. A is an artificial neural used in ANNs. B is a pyramidal neuron, the most common neuron in the neocortex. C shows the HTM neuron connected by three types of dendrites. Every dendrite has synapse connections with other cells.

The HTM neuron has dendrites as threshold coincidence detectors that are divided into three groups that function as the proximal, basal and apical dendrites [9]. Figure 3.1 shows the HTM neuron with a few synapses in every single type of dendrite. The proximal dendrites are responsible for the feedforward signals, basal receive contextual information from nearby cells and the apical dendrites are responsible for the feedback signals.

The feedforward input has two simultaneous representations, a single representation for the raw input and another one for the input in a specific temporal context. Figure 3.2 is a representation of how the neurons are activated when given two different sequences. When the sequence is yet to be learned a subset of mini-columns is activated to represent the element, this can be seen in figure 3.2.B. After learning the sequence the amount of neurons activated in every column is reduced for every element. When recognizing sequences instead of activating a complete column, the first cells that fire inhibit the rest of the neurons in the column. Unlike traditional feedforward processing, the information processed by the feedforward connections is mostly an internal representation of the input and has little meaning on its own.

The detection of sequences is done by the context and feedback dendrites. The previous states activate some of these dendrites, which then depolarize cells in certain columns making those cells more likely to activate. When feedforward activation occurs, neurons can inhibit other neurons from firing when the sequence has

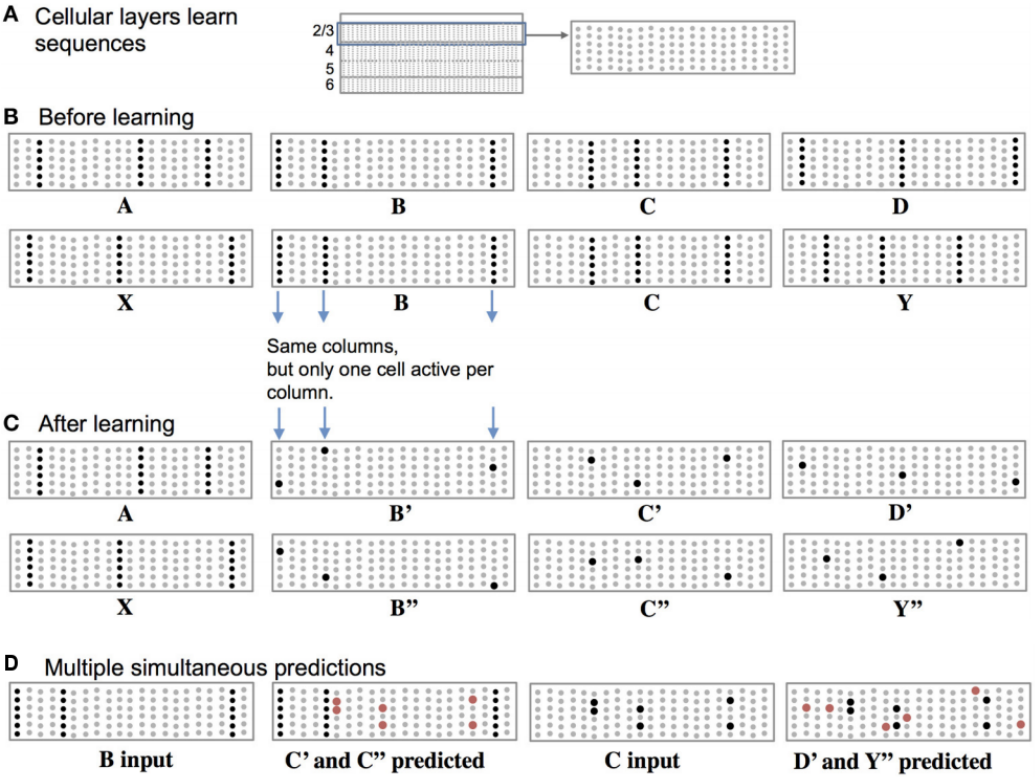


Figure 3.2: A simple representation of the learning process for HTM within a single layer. Presenting only the feedforward signal in B and C. Two sequences are presented, ABCD and XBCY which are learned in B and are validated in C. The neurons are represented in mini-columns as they are amalgamated in the actual model. D shows the context predictions at work with red cells representing the depolarized neurons. Two sequences are being predicted simultaneously.

already been learned. However, they can also depolarize or excite other neurons. This is the contextual information that HTM uses. These predicted neurons do not necessarily activate but when the next input is as predicted the sequence continues. This clarifies that the HTM cells have three states, active, inactive and predictive. In figure 3.2.D an example is given of how the predictions occur for the sequences ABCD and XBCY when only the B is recognized. Noteworthy is that the predictions are not limited to only one sequence.

Lastly the feedback dendrites perform a similar role as the context ones. The HTM models exists out of multiple layers constructed in a hierarchical manner. Information usually goes from the bottom layers progressively to the top layers. When it comes to feedback dendrites activation of neurons in higher layers may affect the output of neurons in lower layers. The feedback layer interprets sequences, when a feedforward input arrives the feedback connections will bias the activation towards certain sequences.

For updating the predictions and keeping up with the information, learning has to occur. The learning method that HTM uses is removing unused synapses and forming new ones. For each dendrite a number of potential synapses is registered, each synapse has a permanence value between 0 and 1 and a threshold in the same range. When a synapse has a permanence value higher than the treshold then its weight is 1, otherwise, it is 0. A synapse can only have one of those two weights. The higher the permanence value the longer the synapse remains useable. The permanence is updated depending on if the cell was predicted correctly, if so,

then the dendrites that caused the prediction will be reinforced. There is also a decay for neurons that had active segments but did not fire; those segments might have received high permanence value unnecessarily.

3.2 Region Structures

The neuron structure is the basis for the regions, the activation of neurons described in the previous section are used to pass information between regions. A hierarchical structure of regions is what forms the HTM network. The use of regions can be very flexible and inputs of two regions of the same level can be combined to form a singular output. In figure 3.2 a splice of a region is illustrated, there are interconnected cells arranged in columns. The regions are layered one on top of the other to create a hierachy of regions. Information from lower regions is usually send up to higher regions while being generalized along the way.

3.2.1 Sparse Distributed Representations

Regions receive inputs consisting of a large number of bits. One of the first manipulations done on this data is to convert it into a sparse distributed representation (SDR). The idea of an SDR is to only use a small percentage of the neurons to represent the data. Only using a small part of the available neurons diminishes the number of possible combinations and thus creates a loss of information in theory when compared to dense representations. However, studies claim that when provided with the right size in parameters, large numbers of patterns can be retrieved perfectly from an SDR, having no loss of information in practice [1, 2].

When columns or cells activate they can differ in the level of activation. Even though, all the cells in a column are activated in figure 3.2.A, once the sequence is learned only a few cells will be activated. Learned sequences are always represented by a small subset of cells that activate. This subset of active cells, for any representation in HTM is always about the same size. About an x% of the total amount of cells will be active to form any representation. The SDR is created by letting active cells inhibit other cells that are nearby, increasing the sparsity.

3.2.2 Context Prediction

Once the input is transformed to an SDR the next step is to add context information based on previous inputs. For the final step, based on the completed representation, a prediction is made for the next input. These steps are represented in the cell level in figure 3.2.

When an input is received and is unpredicted the columns with highest activation will activate. However, when an input is received and there are cells in the predictive states those become active instead. Over time

the input pattern starts to show similarities in the sequence of cells activated. When cells are often activated one after the other they form connections, either across layers or laterally, allowing for cells to change to predictive states when a familiar pattern is occuring.

3.3 Spatial Pooling

In order to transform data into the correct distributions pooling algorithms have been implemented. The first step in a region is to form an SDR. This is where the spatial pooling algorithm comes into play. The actions for the neurons and regions mentioned previously are accounted by the spatial pooler.

3.3.1 Goals

The amount columns that are active, the permanence values of synapses and the activation levels of columns are all manipulated by the spatial pooler. This behavior can be described by a series of overlapping goals [1].

Most of the tasks the spatial pooler fullfils are operations done on the columns in the region. A certain amount of columns are supposed to be active at any time, this is the SDR. At the same time the spatial pooler is set up to use all columns at one point of time, so columns which have not been activated have their activation levels boosted.

The mapping created for the input to the internal representations is expected to have a level of complexity. This is level of complexity is achieved by having the spatial pooler maintaining the number of synapses that are required for a column as threshold to activate. On the flip side the spatial pooler must prevent columns from forming to many synapses so that a column does not respond to too many different types of inputs.

3.3.2 Implementation

The spatial pooler performs its tasks divided into three phases and a initialization. Starting with a empty region of columns and eventually keeping track of the number of columns and active synapses.

Before the region starts with computing the input all the columns receive a number of potential synapses that are taken as random inputs from the input sets. These synapses receive random permanence values in close range of the activation threshold for cells allowing them to quickly become used or unused synapses.

The first phase starts with receiving the input which can be sensory data or the output of another region. Then an already set number of columns will have to activate. To do this first for each column a score is calculated based on the active synapses connected to the column and multiplied with boost (a weight to bias

```
1: for c in columns do
2:   c.overlap = 0
3:   for synapses in c.inputSynapses() do
4:     c.overlap = c.overlap + synapse.active()
5:     if c.overlap < minOverlap then
6:       c.overlap = 0
7:     else
8:       c.overlap = c.overlap * c.boost
9:     end if
10:  end for
11: end for
```

Algorithm 1: Overlap

the activation of certain columns). This phase is described by the psuedocode in Algorithm 1. MinOverlap represents the minimum score or inputs required for a column to go to phase two.

The second phase uses the overlap score calculated by phase one to activate the columns. The columns are chosen based on if the score is high enough compared to the inhibition from the other columns. If a column is among the top n highest scores, in comparison to its neighbors, it will activate. The value of n is determined by the parameter desiredLocalActivity. Algorithm 2 depicts the second phase.

```
1: for c in columns do
2:   localActivity = nthHighest(c.neighbors, desiredLocalActivity)
3:   if c.overlap > 0 and c.overlap > localActivity then
4:     activeColumns.append(c)
5:   end if
6: end for
```

Algorithm 2: Inhibition

The final phase is the learning phase which updates the dynamic variables used in the previous two phases. Firstly the permanence for the synapses of active columns is modified, see lines 3-8 of algorithm 3. After updating the permanence values the algoirthm will update the minDutyCycle or the desired fire rate, the average fire rate in line 14 and the times that the overlap score was higher than minOverlap in line 16. For columns that do not fire often enough the permanence values of their synapses will increase. The boost variable is updated for columns do not activate often enough.

3.4 Temporal Pooling

The second algorithm that manipulates the columns and cells within the HTM network is the temporal pooler. The idea behind the temporal pooler is that it leads to stable representations of sequences, making it possible for predictions to occur. These representations are formed by forming connections between cells that are often fired sequentially, creating a memory distributed among all the cells.

```
for c in activeColumns do
  for synapse in c.potentialSynapses do
    if synapse.active() then
      synapse.permanence += permanenceInc
      synapse.permanence = min(1.0, synapse.permanence)
    else
      synapse.permanence -= permanenceDec
      synapse.permanence = max(0.0, synapse.permanence)
    end if
  end for
end for
for c in columns do
  c.minDutyCycle = 0.01 * c.neighbors.maxDutyCycle
  c.updateActiveDutyCycle()
  c.updateBoost()
  c.updateOverlapDutyCycle()
  if c.overlapDutyCycle < c.minDutyCycle then
    increasePermanences(c, 0.1 * permanceThreshold)
  end if
end for
```

Algorithm 3: Updating

3.4.1 Implementation

The temporal pooler can be divided into three phases. The temporal pooler regulates the behavior that results from the basal (context) and apical (feedback) dendrites.

The first phase takes the feedforward input provided by the spatial pooler and does the following for all active columns:

1. Predictive cells will be activated.
2. If there are no predictive cells the whole column will be activated.
3. Set the learnstate on active for cells with a connection to cells with an active learnstate.
4. If no cell has received an active learnstate, activate the learnstate of the cell with the least connections.

The second phase checks for each cell, in the region, their lateral connections. If enough activation occurs from those lateral connections the cell will enter a predictive state. Furthermore, the connections that caused the activation will be strengthened.

The final phase is considered the learning phase. It does two things specifically:

- The cells with active learning states have all their connections reinforced.
- Cells that were in a predictive state but did not activate, will have the connections that caused the prediction to be diminished.

The temporal pooler, together with the spatial pooler, is the core processing that occurs in HTM. The spatial pooler ensures the internal representations and that all columns and cells are used evenly for that purpose. The temporal pooler uses those internal representations and adds context to them. However, the temporal pooler remains in a experimental phase and in the future its tasks, or the way it performs them, might still change.

3.5 Parameters

The HTM methods has an incredibly large amount of parameters which can be tuned. An example of a set of model parameters used for a single region within an HTM network can be found in appendix A. Seperately these parameters all have some small influence on the performance of an HTM in this section we are looking at the most significant variables.

The first parameter that stands out is a familiar one, which is the learning rate. The learning rate has a similar effect in an HTM network as in an ANN that it determines how fast the network is reacting to its mistakes. Too high of a learning rate can still lead to divergence and a too low learning rate leads to very slow convergence.

Next we have the encoder for every input a region receives it can encode the bits using numerous of cells. The encoder determines in what fashion the poolers choose the kind of cells to represent the input and how they are connected with each other. For instance an AdaptiveScalarEncoder leads to activation of cells for a certain bit string x, when another bit string y comes along the encoder will possible use some cells that were used for the bit string x. The number of cells that overlap between x and y is dependent on the similarity of the strings. A CategoryEncoder, when two different categories have been recognized, will try to encode those categories into two complete seperate groups of cells.

Both poolers have their own set of parameters, these parameters determine how quickly the poolers increase or decrease the permanence for synapses. There are also parameters restricting the amount synapses for cells and the activation threshold for cell activation; making these parameters in a way comparable to the learning rate. A too high decay will possibly lead to a very slow convergence.

Chapter 4

Empirical Investigation

This chapter describes the settings used for the empirical investigation. The performance of HTM has been measured in this thesis using regression use cases. Furthermore it was compared to specialized methods like SVM and RF, as well as CNN which has been originally created for image recognition datasets. To encompass variance in the datasets several vastly different datasets were used.

4.1 Specifications

```
Server Specs
  Intel Haswell E5-2630-v3 CPU configuration
    8 cores @2.4GHz
    64GB memory
  CentOS Linux release 7.2.1511

Software used
  NuPIC 0.5.4
  Commit: 2790cefcd0bfaea4af768bd79240bb471b99da6c
  scikit-learn (0.17.1) Python package
  theanets (0.8.0) Python package
  Theano (0.8.2) Python library
  ffnet (0.8.3) Python library
```

4.2 Datasets

Three types of datasets were used to measure regression performance. These datasets were all downloaded from the UCI Machine Learning Repository. The experiments where these datasets originate from were chosen at random.

First a small dataset, referred to as airfoil, consisting of 6 attributes including the predicted field and 1503 instances. This data came from a NASA experiment sound pressure from airfoils based on five variables. Downloaded from `https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise#` (visited on 10-08-2016).

The second dataset is the medium dataset, referred to as crime. This dataset has 147 attributes of which several were possible fields to predict. A few fields were not relevant for the predictions and a lot of records included missing values. This can test the capabilities of the machine learning techniques when handling noisy data. In the dataset 2215 communities are included with mostly attributes relevant and correlating to crime. Downloaded from `https://archive.ics.uci.edu/ml/datasets/Communities+and+Crime+Unnormalized` (visited on 10-08-2016).

Lastly we have the large dataset, referred to as histogram. The number of instances and attributes were 53500 and 386 respectively making it very hard to optimize for this data. The server was unable to even run the GP model for this data. This data consists of varables from CT slices, describing the bone structure and air inclusion in human bodies. The location of the image was supposed to be asserted based on those CT slices. Downloaded from `https://archive.ics.uci.edu/ml/datasets/Relative+location+of+CT+slices+on+axial+axis` (visited on 10-08-2016).

To reduce variance in the results 10-fold cross validation was used for training and testing. All three of these datasets were split into 10 equal sized subsamples and every sample was used once as a validation set, while the other 9 were the training sets.

4.3 Model Tuning

A core part of the performance of different machine learning techniques is the parameter tuning. Trying to find the optimal parameters for a model for a specific dataset is a challenge in itself. In this section the methods used for parameter optimization will be elaborated on. Unfortunately limited by time not every possible variable can be tested. The variables tested were in reasonable range when taking into account computational time.

4.3.1 Sklearn

The scikit-learn package was used for the implementations of the methods: GP, RF and SVM. The scikit-learn package has the useful functions randomsearch and gridsearch for hyperparameter optimization. Both functions were used for all three methods resulting in a reasonable estimation of the optimal hyperparameters. The methods form a search space out of the possible parameters and then move along the search space either randomly, or exhaustively along the finite set parameters representing the possible best parameters.

4.3.2 Swarming

Numenta employs its own search space methods for finding the best variables for HTM [11]. In its core, swarming is similar to gridsearch and randomsearch in the sense that models of parameters will be tested on the dataset. The best evaluated model will be returned. Swarming can be divided into two levels of processing.

Starting with the field search, this process determines which attributes to include in the model. This is done by building single attribute models for every attribute. The attribute that produced the best model will be chosen. Then continuing the sequence the process builds double attribute models using the previously found best attribute in every combination. This sequence continues until adding attributes does not improve model evaluations.

The best parameters for every attribute model are found by running mini-swarms (MS). Mini-swarms evaluate multiple models by changing the parameters using the Particle Swarm Optimization (PSO) algorithm. The basic idea behind this algorithm is that particles are put in a search space of possible parameters. These particles are evaluated and then receive velocities based on the locations of the best particle globally, and the best particle in their local range. Over time these particles should move to the most optimal positions.

4.4 Code

The code of the programs that were used can be found in appendix B. The code shown is written specifically for the airfoil dataset but can be modified for usage on other datasets. The programs require the installation of the specifications mentioned above. This code can easily be transformed to work for another dataset by adjusting the input file and reappointing the target value. Before running these programs a swarm or another search must be done in order to find the hyperparameters. The data files are expected to be in the same folder as the programs.

Chapter 5

Results

This section contains the results of the empirical comparison between the serveral machine learning methods mentioned in chapter 2. The results for the three datasets are summarized in the tables 5.1, 5.2 and 5.3. Extreme differences were found both in performance as in the the running times for almost all the methods. These differences in performance can be positioned mostly on the distinct natures of the machine learning methods.

Two metrics were used for the comparisons. The first method is mean squared error (MSE). Given a y that represents the true value and $\hat{y}$ the predicted value the MSE over n samples is defined as follows:

$$MSE(y, \hat{y}) = \frac{\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2}{n}$$

The second metric is R^2 this allowed for comparisons of methods over datasets see plot 5.2. For the same y, $\hat{y}$ and n as MSE, R^2 is defined as:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1}(\hat{y}_i - y_i)^2}$$

5.1 Speed

The time it took for methods to reach a certain level of accuracy varries significantly. The measured time only consists of the time that was required for fitting the model to the training set until it converges and running the models on the validation sets. This excludes the search for hyperparameters by swarming or grid search.

Outliers in speed are the SVM and FF methods which are extremely slow in comparison to the others. It must be noted that the average of the SVM methods is very slow in the plot 5.2, however, the speed for svm was significantly faster than most methods for two out of three datasets. The SVM speed for the third

dataset, can be attributed to the complexity of the kernel mapping, that was required on such a large dataset. The FF implementation was testing out various activation functions during execution which other models already had done before execution. Next, the HTM and CNN models showed medium speeds, which is to be expected when working with a larger set of layers, and thus more complexity in training. Finally the GP model showed a speed just slower than HTM and CNN but was not tested for the largest dataset. Giving it a more favorable position in graph 5.2.

5.2 Structure

The structure of a model has a great effect on its performance on certain datasets. This structure is formed by the implementation together with the user input. A lack of performance could be attributed to the nature of a model, or lack of optimization in the structure or user input.

As CNN was formed with image recognition in mind the same structure might not prove useful for regression. This could explain the resulting performance found. The CNN model is most effective when data points found in the same receptive field correlate well with each other. Which is often the case for images, pixels next to each other are usually part of the same object. This is not necessarily the case in regression tasks. In those cases the convolutions might have lead to significant losses of information.

The SVM models for all three datasets were using an RBF kernel function. Similarly the GP model was limited only one autocorrelation function. Both these choices showed best performance over all three datasets, representing the overall performance of such methods. Custom functions might have shown a better performance for singular datasets.

It has been noted before that the HTM theory and implementation are both works in progress [1]. The HTM model has not been build with necessarily regression in mind and no modifications were added to ensure better performance for regression. The structure of HTM was taken as is and optimized in performing the regression problem through only HTM encoders and regions. On the other hand, the CNN models start of with convolution and pooling layers but ends with an MLP, which can produce formidable results on its own.

Table 5.1: Showing the performance results of the methods on the airfoil dataset.

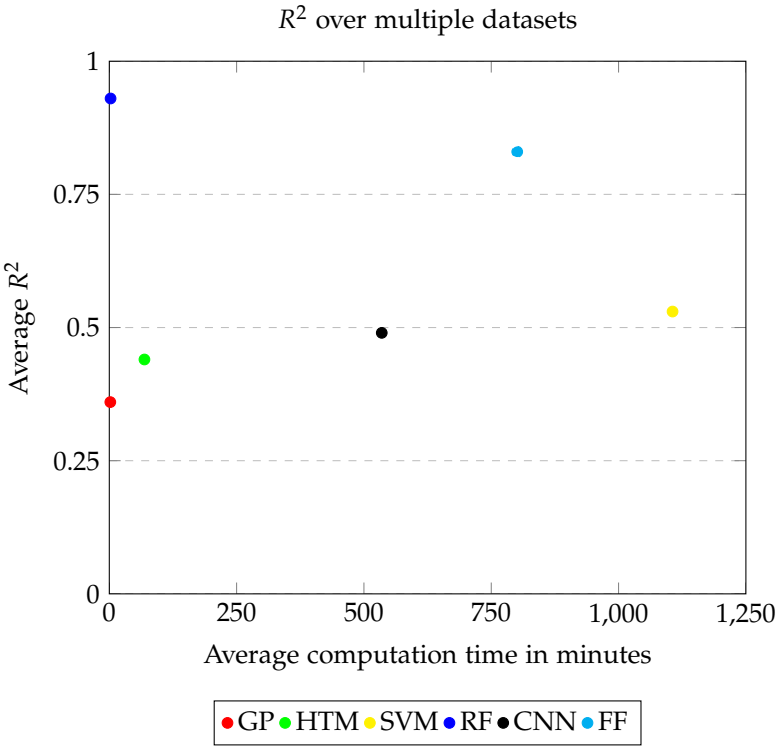
Method	MSE	R^2	Time
HTM	20.040	0.734	19m20s
GP	22.780	0.521	3m22s
SVM	19.627	0.587	2m31s
RF	4.174	0.912	0m2s
CNN	38.533	0.190	18m51s
Feedforward	6.564	0.862	8m46s

Table 5.2: Showing the performance results of the methods on the histogram dataset.

Method	MSE	R^2	Time
HTM	176.609	0.587	188m9s
SVM	0.936	0.998	3299m12s
RF	2.427	0.995	7m20s
CNN	0.245	0.999	633m43s
Feedforward	55.733	0.888	2223m14s

Table 5.3: Showing the performance results of the methods on the crime dataset.

Method	MSE	R^2	Time
HTM	218663.370	0.000	0m11s
GP	205969	0.191	0m40.71s
SVM	254831.102	-0.001	10s
RF	32521.822	0.872	0m11s
CNN	176966.373	0.305	919m4s
Feedforward	62966.357	0.753	169m50s



5.3 Discussion

The HTM model does not perform well in the regular regression use cases when compared to most other methods. For two out of three cases HTM performed among the worst. However, it is understandable why HTM performs so much for such a regression problem when taking into account the theory behind the

mehthods.

The methods in this thesis use the inputs received from a dataset to make a prediction of a target. The methods have their own way of doing this, RF uses multiple decision trees while FF use the activation functions of their neurons to process an output. CNN forms receptive fields from groups of neurons and then searches for correlations there. This is very effective for images, as is shown from the results of the histogram dataset. It might also lead to losses of information as can be seen in the airfoil set. The way the input is processed plays a significant role on the performance of a method. The HTM method uses the input as a means to form an SDR. The SDR is then only used as an element in a temporal sequence of SDRs. The HTM methods does not search for correlations between the inputs and the output, but instead looks at the input as a chain, continuing from the previous input. This basically means that the HTM model is not the right fit for regular regression datasets, however, regression sets with temporal attributes might produce different results.

Chapter 6

Conclusions

In this thesis a conceptual overview was given for the HTM model showing how it differs from some popular machine learning methods being used currently. An empirical investigation has been performed as well, comparing the HTM model to several other techniques, all of them specialized in regression problems with the exception of the CNN model.

The HTM model was created with the idea of a generalized intelligence supposedly being able to have one foundation framework for performing completely different tasks. From the results followed that HTM definitely does not compare to specialized methods in regression problems. Furthermore when looking at the performance of CNN, a model not specialized in regression, the HTM performance is not necessarily better for the datasets.

Two things have to be taken into account when looking at these results. Firstly, the HTM model is based purely on spatial-temporal sequence recognition and thus does not search for the connections, between the input and output, in the same matter as other methods do. It might be possible that HTM will perform better for regression where time is significant. Secondly, the HTM model is not completely developed yet theoretically and was not necessarily made with regression problems in mind. However, in the future the foundation framework might be more refined both theoretically and practically, allowing for HTM models to be able to respond more accurately and efficiently to any type of problem presented. It will be interesting to see how well HTM currently performs in other areas. An attempt has been made for this thesis to compare the HTM to CNN in image recognition cases. Unfortunately due to problems with the implementation of HTM no empirical comparison has been made, leaving it as a possibility for future work.

Appendix A

Model Parameters

```
'model': 'CLA',
'modelParams': {    'anomalyParams': {    u'anomalyCacheRecords': None,
                                          u'autoDetectThreshold': None,
                                          u'autoDetectWaitRecords': None},
                    'clParams': {    'alpha': 0.025971514856200795,
                                     'clVerbosity': 0,
                                     'regionName': 'CLAClassifierRegion',
                                     'steps': '1'},
                    'inferenceType': 'TemporalMultiStep',
                    'sensorParams': {    'encoders': {    u'1': {    'clipInput': True,
                                                                    'fieldname': '1',
                                                                    'n': 79,
                                                                    'name': '1',
                                                                    'type': 'AdaptiveScalarEncoder',
                                                                    'w': 21},
                                                         u'2': None,
                                                         u'3': None,
                                                         u'4': None,
                                                         u'5': None,
                                                         u'6': {    'clipInput': True,
                                                                    'fieldname': '6',
                                                                    'n': 142,
                                                                    'name': '6',
                                                                    'type': 'AdaptiveScalarEncoder',
                                                                    'w': 21},
                                                         '_classifierInput': {    'classifierOnly': True,
                                                                                  'clipInput': True,
                                                                                  'fieldname': '6',
                                                                                  'n': 95,
                                                                                  'name': '_classifierInput',
                                                                                  'type': 'AdaptiveScalarEncoder',
                                                                                  'w': 21}},
                                         'sensorAutoReset': None,
                                         'verbosity': 0},
                    'spEnable': True,
                    'spParams': {    'columnCount': 2048,
                                     'globalInhibition': 1,
                                     'inputWidth': 0,
                                     'maxBoost': 2.0,
                                     'numActiveColumnsPerInhArea': 40,
                                     'potentialPct': 0.8,
                                     'seed': 1956,
                                     'spVerbosity': 0,
                                     'spatialImp': 'cpp',
                                     'synPermActiveInc': 0.05,
                                     'synPermConnected': 0.1,
                                     'synPermInactiveDec': 0.0671043688429658},
```

```
48                          'tpEnable': True,
49                          'tpParams': {       'activationThreshold': 13,
50                                              'cellsPerColumn': 32,
51                                              'columnCount': 2048,
52                                              'globalDecay': 0.0,
53                                              'initialPerm': 0.21,
54                                              'inputWidth': 2048,
55                                              'maxAge': 0,
56                                              'maxSegmentsPerCell': 128,
57                                              'maxSynapsesPerSegment': 32,
58                                              'minThreshold': 10,
59                                              'newSynapseCount': 20,
60                                              'outputType': 'normal',
61                                              'pamLength': 2,
62                                              'permanenceDec': 0.1,
63                                              'permanenceInc': 0.1,
64                                              'seed': 1960,
65                                              'temporalImp': 'cpp',
66                                              'verbosity': 0},
67                          'trainSPNetOnlyIfRequested': False},
68      'predictAheadTime': None,
69      'version': 1}
```

Appendix B

Code

B.1 Sci-kit models

```
#!/usr/bin/env python

import numpy as np
from sklearn.gaussian_process import GaussianProcess
from sklearn.cross_validation import KFold

output = open("output", "wb")
completeX = np.genfromtxt('airfoil.csv', delimiter=',')

#Divide the target and inputs for the airfoil data
completey = completeX[:,5]
completeX = np.delete(completeX, 5, 1)


gp = GaussianProcess()
#svr_rbf = SVR() #SVM Method
#rf = rf.fit(X_train, y_train) #RF Method


kf = KFold(completeX.shape[0], n_folds=10, shuffle=True)

for train_index, test_index in kf:
  X_train, X_test = completeX[train_index], completeX[test_index]
  y_train, y_test = completey[train_index], completey[test_index]

  gp = gp.fit(X_train, y_train)
  prediction = gp.predict(X_test)

  #SVM
  #y_rbf = svr_rbf.fit(X_train, y_train)
  #prediction = y_rbf.predict(X_test)

  #RF
  #rf = rf.fit(X_train, y_train)
  #prediction = rf.predict(X_test)


  for i in range(0, prediction.size):
    print >> output, prediction[i], y_test[i]
```

B.2 Feedforward Net

```
#!/usr/bin/env python
import numpy as np
from ffnet import ffnet, mlgraph, readdata, savenet
from sklearn.cross_validation import KFold

output = open("output", "wb")

#LOAD training set
completeX = np.genfromtxt('airfoil.csv', delimiter=',')
completey = completeX[:,5]
completeX = np.delete(completeX, 5, 1)

kf = KFold(completeX.shape[0], n_folds=10, shuffle=True)
for train_index, test_index in kf:
 X_train, X_test = completeX[train_index], completeX[test_index]
 y_train, y_test = completey[train_index], completey[test_index]
 #number of inputs varries for datasets
 conec = mlgraph( (5,100,1) )
 net = ffnet(conec)
 net.train_tnc(X_train, y_train, maxfun = 100)
 prediction = net.test(X_test, y_test)
 for i in range(0, prediction[0].size):
  print >> output, prediction[0][i][0], y_test[i]
```

B.3 Convolutional Neural Network

```
#!/usr/bin/env python
import climate
import logging
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rng
import theanets
from sklearn.cross_validation import KFold

climate.enable_default_logging()

output = open("output", "wb")

#LOAD DATASET
completeX = np.genfromtxt('airfoil.csv', delimiter=',')
completey = completeX[:,5]
completeX = np.delete(completeX, 5, 1)
completey = completey.reshape(completey.shape[0],1)


ff = dict(name='feedforward', size=100)
c1 = dict(name='conv1', size=100, filter_size=1)
p1 = dict(name='pool1', size=100)
c2 = dict(name='conv1', size=100, filer_size=2)
p2 = dict(name='pool1', size=100)


kf = KFold(completeX.shape[0], n_folds=2, shuffle=True)

for train_index, test_index in kf:
        X_train, X_test = completeX[train_index], completeX[test_index]
        y_train, y_test = completey[train_index], completey[test_index]
        e = theanets.Regressor(
                layers=(5, c1, p1, c2, p2, (1000, 'tanh') ,100,  1))

        e.train([X_train, y_train])
        prediction = e.predict(X_test)
        for i in range(0, prediction.size):
```

```
39                 print >> output, prediction[i][0], y_test[i][0]
```

B.4 HTM

```
1  import importlib
2  import sys
3  import csv
4  import datetime
5
6  from collections import deque
7  import unittest2 as unittest
8  from nupic.data.file_record_stream import FileRecordStream
9  from nupic.data.inference_shifter import InferenceShifter
10 from nupic.frameworks.opf.metrics import MetricSpec
11 from nupic.frameworks.opf.modelfactory import ModelFactory
12 from nupic.frameworks.opf.predictionmetricsmanager import MetricsManager
13 from nupic.swarming import permutations_runner
14 import numpy as np
15 import logging
16 import pandas as pd
17
18 #import nupic_output
19 _LOGGER = logging.getLogger(__name__)
20
21
22 DESCRIPTION = (
23   "\n"
24 )
25 GYMNAME = "airfoil"
26 DATA_DIR = "."
27 MODEL_PARAMS_DIR = "./model_params"
28
29 _METRIC_SPECS = (
30     MetricSpec(field='6', metric='multiStep',
31                inferenceElement='multiStepBestPredictions',
32                params={'errorMetric': 'aae', 'window': 10, 'steps': 1}),
33     MetricSpec(field='6', metric='trivial',
34                inferenceElement='prediction',
35                params={'errorMetric': 'altMAPE', 'window': 10, 'steps': 1}),
36 )
37
38
39 def createModel(modelParams):
40   model = ModelFactory.create(modelParams)
41   #CHOOSE TARGET FIELLD
42   model.enableInference({"predictedField": "6"})
43   return model
44
45 def getModelParamsFromName(gymName):
46   importName = "model_params.%s_model_params" % (
47     gymName.replace(" ", "_").replace("-", "_")
48   )
49   print "Importing model params from %s" % importName
50   try:
51     importedModelParams = importlib.import_module(importName).MODEL_PARAMS
52   except ImportError:
53     raise Exception("No model params exist for '%s'. Run swarm first!"
54                     % gymName)
55   return importedModelParams
56
57
58
59 def runIoThroughNupic(inputData, model, gymName, plot):
60         rscore = 0
61         y = 0
62         metricsManager = MetricsManager(_METRIC_SPECS, model.getFieldInfo(),
63                                         model.getInferenceType())
```

```
64          inputFile = open(inputData, "rb")
65          shifter = InferenceShifter()
66          output = open("htm_airfoil_output", "wb")
67          #MANUAL FOLDS ARE REQUIRED
68          dataSource = FileRecordStream(streamID="airfoilfold1.csv")
69          counter = 0
70          df = pd.read_csv('./airfoilfold1.csv')
71
72          for i in range(2, df.shape[0]):
73                  nextrecord = dataSource.getNextRecordDict()
74                  result = model.run(nextrecord)
75                  result.metrics = metricsManager.update(result)
76                  if i > 1351: #Validation set
77                    prediction = result.inferences["multiStepBestPredictions"][1]
78                    output.write("%s, %s \n" % (df.loc[i][5], prediction))
79          inputFile.close()
80          output.close()
81
82
83
84
85
86  def runModel(gymName, plot=False):
87    print "Creating model from %s..." % gymName
88    model = createModel(getModelParamsFromName(gymName))
89    inputData = "%s/%s.csv" % (DATA_DIR, gymName.replace(" ", "_"))
90    runIoThroughNupic(inputData, model, gymName, plot)
91
92
93
94  if __name__ == "__main__":
95    print DESCRIPTION
96    plot = False
97    logging.basicConfig(level=logging.INFO)
98
99    args = sys.argv[1:]
100   if "--plot" in args:
101     plot = True
102   runModel(GYMNAME, plot=plot)
```

Bibliography

[1] Hierarchical Temporal Memory including HTM Cortical Learning Algorithms. Technical report, Numenta, Inc., 2011.

[2] Subutai Ahmad and Jeff Hawkins. Properties of Sparse Distributed Representations and their Application to Hierarchical Temporal Memory. *Preprint at http://arxiv.org/pdf/1503.07469.pdf*.

[3] Alberto J. Perea and José E. Meroño and María J. Aguilera . Application of Numenta® Hierarchical Temporal Memory for land-use classification. *South African Journal of Science*, 105:370–375, 2009.

[4] Yoshua Bengio and Yann LeCun. *Scaling Learning Algorithms towards AI*. MIT Press, Massachussetts, 2007.

[5] Leo Breiman. Random Forests. *Mach. Learn*, 45(1):5–32, 2001.

[6] Gavin C. Cawley and Nicola L. C. Talbot. On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation. *Journal of Machine Learning Research*, 11:2079–2107, 2010.

[7] Li Deng and Dong Yu. Deep learning methods and application. *Found. Trends Signal Process.*, 7:197–387, 2014.

[8] Michael Galetzka. Intelligent prediction: an empirical study of the Cortical Learning Algorithm. Master's thesis, Mannheim University of Applied Sciences, 2014.

[9] Jeff Hawkins and Subutai Ahmad. Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex. *Frontiers in Neural Circuits*, 10:23, 2016.

[10] Jeff Hawkins and Sandra Blakeslee. *On Intelligence*. Times Books, New York, 2004.

[11] Numenta Inc. Swarming Algorithm. `https://github.com/numenta/nupic/wiki/Swarming-Algorithm`, September 2015. [Online; accessed 10-08-2016].

[12] Kevin Jarrett, Koray Kavukcuoglu, Marc Ranzato, and Yann LeCunn. What is the Best Multi-Stage Architecture for Object Recognition? *IEEE Press*, pages 2146–2153, 2009.

[13] Ioannis Kostavelis and Antonios Gasteratos. On the Optimization of Hierarchical Temporal Memory. *Pattern Recognition Letters*, 33(5):670–676, 2012.

[14] Ritchie Lee and Mariam Rajabi. Assessing NuPIC and CLA in a Machine Learning Context using NASA Aviation Datasets. Carnegie Mellon University EE18799-SV, 2014.

[15] Gilles Louppe. *Understanding Random Forests: From Theory to Practice*. PhD thesis, University of Liège, 2014.

[16] Davide Maltoni. Pattern Recognition by Hierarchical Temporal Memory. Technical report, DEIS, 2011.

[17] Dandan Mo. A survey on deep learning: one small step toward AI. 2012.

[18] Michael A. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com, 2015.

[19] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Massachussetts, 2006.

[20] Carl Edward Rasmussen. Gaussian Processes in Machine Learning. Technical report, Max Planck Institute for Biological Cybernetics, Cambridge, MA, USA, 2006.

[21] Edward Lloyd Snelson. *Flexible and efficient Gaussian process models for machine learning*. PhD thesis, University of London, 2007.