MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Optimization of Hierarchical Temporal Memory through Efficient Sparse Representations

Scientific Students' Association Report

Author
Csongor Pilinszki-Nagy

Supervisor
Bálint Gyires-Tóth, PhD

October 28, 2019

Contents

Kivonat

A Hierarchikus Temporális Memória (HTM) egy speciális neurális hálózat, ami nagyban különbözik a széles körben elterjedt mély neurális hálózatokhoz képest. Idősorok és szekvenciák hatékony tanulására képes, amit a hálózat változó hosszúságú szekvencia memóriával valósít meg. Ez elméletben egy rendkívül hatékony módja a szekvenciák modellezésének.

A hálózat négy elkülöníthető rétegből áll, mindegyik külön feladattal rendelkezik. Elsőként a kódoló réteg átalakít bármilyen bemenetet ritka bináris vektorokká. A Spatial Pooler normalizálja a kapott bemeneteket, majd a térbeli modellezést végzi el. A Temporális Memória végzi el a szekvencia tanulást a Spatial Poolertől kapott normalizált adatokon. Végül a dekódoló visszaalakítja a Temporális Memória kimenetét a cél értékre, ami a szekvencia következő eleme. Ez a felépítés a hálózat könnyű megértését eredményezi, így annak működése magyarázható. A hálózat rétegei között a kapcsolatok ritkák, amely hatékony implementálása az elméleti kihívások mellett körültekintő tervezést és optimalizálást igényel.

Eddig kevés kutatás zajlott a témában, a legígéretesebb eredményeket a Numenta csoport készítette. A korábbi kutatásom a Numenta NuPIC implementációjából indult ki, a jelenlegi munkám pedig az előző TDK dolgozatom eredményeire épít. A legutóbbi implementációt felhasználva annak változóit ritka mátrixokra és tenzorokra cseréltem, mivel ezek jobban kell, hogy illeszkedjenek a hálózat struktúrájához.

A kutatásom célja megvizsgálni, a ritka mátrix és tenzor implementáció számítási és memória komplexitásra gyakorolt hatását. A hálózatot szintetikus adatokon való túltanítással validálom, majd a szekvenciák hosszát növelni szeretném. Végül valós adatokon is össze szeretném mérni a hálózat teljesítményét a Long Short-Term Memory (LSTM) stuktúrával szemben.

Abstract

Hierarchical Temporal Memory (HTM) is a special type of neural network that differs from the widely used deep neural networks. It is suited to learn time series and sequences efficiently. The network implements a variable length sequence memory, which is a very powerful way to model series.

The network has four distinct layers in series, each having a specific task. First, the encoder layer translates any input into sparse binary vectors. The Spatial Pooler performs normalization and models the spatial features of the encoded input. The Temporal Memory does the sequence learning on the Spatial Pooler's normalized output. Finally, the decoder takes the Temporal Memory's outputs and translates it back to the target value, e.g., the next instance in the sequence. This structure enables a good understanding of the network and produces an explainable solution. The connections in the network are sparse, which requires prudent design and implementation.

There has been a limited amount of research on this topic. The most promising results came from Numenta. My previous work was based on their implementation, and my current work is the continuation of my previous Scientific Students' Association Report. Using my last implementation, I substituted the values to sparse matrices and tensors since these should fit better with the network's structure.

The goal of this paper is to examine the impact of sparse implementation to the network computational and memory complexity. I validate my network through first overfitting on synthetic data, then lengthen the training series. Finally, real world data is also considered as a way to be measured against the widely used Long Short-Term Memory (LSTM) solutions.

Chapter 1

Introduction

Today artificial intelligence is helping us solve more and more complex tasks. The complexity and capability of these networks are increasing rapidly. However, these networks are still functioning as approximations for nonlinear tasks.

Current neural networks are similar to the human brain, but there are fundamental differences[5], that need to be implemented in order to achieve artificial general intelligence, according to Numenta[10][9] (which is a nonprofit research group dedicated to developing the Hierarchical Temporal Memory theory and bring it to mainstream use). Some of the problems current neural networks suffer from are overfitting, vanishing gradients, false positives and negatives due to noise or just the general amount of computation needed to train trough many epochs.

The Hierarchical Temporal Memory (HTM) network is one of them and Numenta is certain that Artificial General Intelligence (AGI) can only be achieved by mimicking the neocortex and implementing those fundamental differences in a new model. Also there is need to produce explainable AI solutions, that can be understood for example for safety reasons, and understanding and modeling the human brain should deliver better understanding of the decisions.

Sequence learning is one domain of machine learning that aims to learn temporal patterns in sequences, temporal data and time-series. Through the years there were several attempts to solve sequence learning, currently the state of the art deep learning solutions use one dimensional convolutional networks or recurrent neural networks with LSTM cells.[13][6] Despite of the improvements over other solutions these algorithms still lack some of the preferable properties, that would make those ideal for sequence learning.[5]

The HTM network can only work on sequence learning tasks, since it is inherently temporal. Sequence learning means modeling temporal patterns instead of spatial ones. Currently the state of the art neural network for sequence learning is Long Short-Term Memory (LSTM).

Chapter 2

Previous works

The domain for this report is sequence learning. There are a lot of statistical sequence learning methods like Hidden Markov Models[4], Autoregressive Integrated Moving Average (ARIMA)[3]. However this paper is limited to neural network solutions and their performances, this way these architectures are easily comparable.

2.1 Deep learning based sequence modeling

Sequence learning is a way to model temporal data. The model should predict the values in the next time step using the previous history of values. Artificial neural networks evolved in the last decades , but were popularized thanks to the advances in parallel hardware performance recently only. The premise of all these networks is the same, build a network using neurons and synapses with weights connecting them together. Make predictions using the networks weights, and backpropagate the error to optimize weight values. This iterative method can achieve great results.[14][1][2]

Convolutional neural networks were created to recognize images better than the usual neural networks using the spatial relations of the image's pixels. Since then other types of one and three dimensional convolutions solved other tasks as well. This network works efficiently by using small kernels to execute convolutions on the pixels values recognizing small details is any part of the images including lines and edges. These kernels combined with pooling and normalization layers proved to be a powerful way to extract information layer by layer from images. However there are downsides too, convolutional neural networks can't distinguish the different layout of the image components, so a face with different layout is still a face recognized by the network.[8]

Recurrent neural networks were created to learn from sequences and predict some future value, so there is a time-based correlation between the data points in addition to the spatial features. Since time only goes forward, it is not advised to treat the time dimension like any other spatial dimension. The network deals with the time-based correlation by having a recurrent connection with itself, where the last state of the neural network is the input of the network in the current time step.

Recently the use of Recurrent Neural Networks dominates the field of the sequence learning methods because they deal with the time-based structure of the data more efficiently than a fully connected neural network. However, the problem with sequence learning with RNNs is that training those enables only shorter sequence inputs. This Long Short-Term Memory cell can store and retrieve it's inner state and can get rid of the vanishing gradient problem. Both of these solutions use the same basic concept, which rolls out the temporal part of the data into a multi-layered network.

The important task in sequence learning is getting an accurate representation of the context surrounding the actual input. The LSTM networks solves this by chaining together LSTM cells, which either retain of drop information about previous input data. The disadvantages of recurrent neural networks with or wihtout LSTM cells is the limited scope in time. Using too large of an input size can result in vanishing gradients for the first data points. An advanced solution using LSTMs is the Hierarchical Attention Network (HAN)[15]. This type of network contains multiple layers of LSTM cells, which model the data on different scopes.

2.2 Hierarchical Temporal Memory

HTM starts with the core assumption that everything the neocortex does is based on the memory and recall of sequences. These sequences are patterns of the Sparse Distributed Representation (SDR) input, which are translated into the sequences of cell activations in the network. This is an online training method, which doesn't need multiple epochs of training. Most of the necessary synapse connections are created during the first pass, so it can be viewed as a one-shot learning capability. The HTM network can recognize and predict sequences with such robustness, that it does not suffer from the usual problems hindering the training of conventional neural networks. HTM builds a predictive model of the world so every time it receives input, it is attempting to predict what is going to happen next.

HTM is a unique approach to artificial intelligence that starts from the neuroscience of the neocortex. The neocortex is involved in all that is considered intelligent behavior. The structure of the neocortex is homogeneous. The neocortex has a hierarchical structure where lower parts process the stimuli, and higher parts learn more general features. The neocortex consists of neurons, segments, and synapses. There are vertical connections that are the feedforward and feedback information and there are horizontal connections that are the context inputs. The neurons can connect to other nearby neurons through segments and synapses. The feedforward neural network somewhat resembles these principles, it is homogeneous, meaning every cell does the same computations and it has a hierarchy, every cell in higher layers learn something more abstract from the layers beneath. The HTM network can not only predict the future values of sequences but detect anomalies in sequences.

The network consists of the following components:

- The SDR Scalar Encoder, which is capable of representing multidimensional scalar data as a Sparse Distributed Representation (SDR). There are encoders for just about any type of data, but those are beyond the scope of this paper. SDR representation is a form of coding data into binary bit arrays so that it retains the semantic similarity between similar input values by overlapping bits.

- The Spatial pooler, which decides which columns should be activated given the SDR representation of the input. The spatial pooler acts as a normalization layer for the SDR input, which makes sure the number of columns and number of active columns stays fixed. This is crucial for the HTM network to work. The spatial pooler also acts as a convolutional layer of the SDR inputs, by only connecting to specific parts of the input. This however is different from the convolutional neural networks which have uniforms connections, this one is random. This randomness achieves high robustness against different noises.

- The Temporal Memory (in Figure 5.2) receives input from the spatial pooler and does the sequence learning, which is expressed in a set of active cells. Both the active columns and active cells are sparse representations of data just as the SDRs. These active cells not only represent the input data but provide a distinct representation about the context that came before the input.

- The Scalar Decoder takes the state of the Temporal memory and treating it as an SDR decodes it back to scalar values.

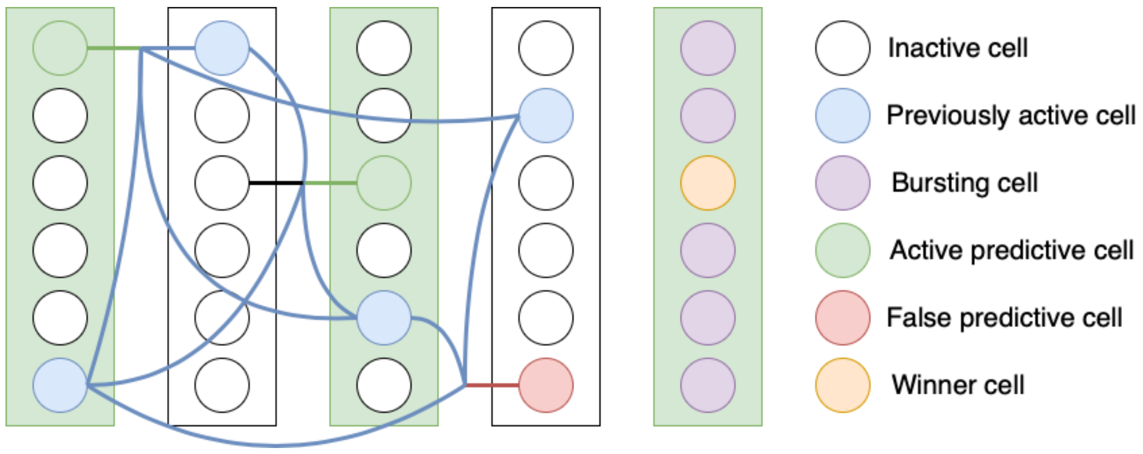

Figure 2.1: *Temporal Memory connections*

The HTM is the close model of the human neorcotex. It consists of two main parts accompanied by two additional wrapping layers. The connections in the neocortex are there to store and recall sequence patterns. This results in accurate predictions in known sequence cases. The connections in the neocortex are sparse.

2.2.1 Sparse distributed representation

HTM starts with the core assumption that everything the neocortex does is based on the memory and recall of sequences. These sequences are patterns of the SDR input, which are translated into the sequences of cell activations in the network. The capacity of a dense bit array in 2 to the power of the number of bits. This gives the bit array a large capacity but little resistance to noise.

Dense representation is a bit array where the ones and zeros are for example in a 50-50% ratio. On the other hand, a spare representation has only a 2% ratio; this enables favorable properties for use in the HTM network. In this sparse case, the capacity is much smaller. In a good representation, every bit in this bit array can have a specific meaning which represents a given object. A sparse bit array can be stored efficiently by only storing the indices of the ones.

In order to enable classification and regression there needs to be a way to decide whether or not two SDRs are matching, so if the HTM network encountered such an SDR before. This is what SDR comparison is for, which calculates the overlapping bits between SDRs. To decide whether or not an overlap can be considered a match there is a threshold value called. The accidental overlaps in SDRs are rare so the matching of two SDRs can be done with high precision. The rate of an SDR matching as a false positive is meager.

2.2.2 SDR scalar encoder

The language of the HTM network is an SDR. There needs to be an encoder for it so that it can be applied to real-world problems. The first and most crucial encoder for the HTM system is the scalar encoder, and I will be focusing on this in the rest of this paper. An encoder for the HTM must meet the following criteria.

The principles of SDR encoding:

- Semantically similar data should result in SDRs with overlapping bits. The higher the overlap, the more the similarity.
- The same input should always produce the same output, so it needs to be deterministic.
- The output should have the same dimensions for all inputs.
- The output should have similar sparsity for all inputs and should handle noise and subsampling.

2.2.3 Spatial pooler

The spatial pooler is the first layer of the HTM network which comes after the encoder. It takes the SDR input from the encoder and outputs a set of active columns. These columns represent the recognition of the input. The columns also have a normalizing effect. There are two tasks for the spatial pooler, maintain a fixed sparsity and maintain overlap properties of the output of the encoder. These properties can be looked at like the normalization in other neural networks which helps the learning process by constraining the neurons behavior.

The column in a spatial pooler means a set of cell that shares the same segment connecting to the encoders SDR input. Presented with the input of these cells that are in a column would like to be activated simultaneously, this means that the column is activated.

The spatial pooler has connections between the SDR input cells and the spatial pooler columns. Every synapse is a potential synapse which can be connected or not depending on its strength. At initialization, there are only some cells connected to one column with a potential synapse. The randomly initialized spatial pooler already satisfies the two criteria, but a learning spatial pooler can do an even better representation of the input SDRs.

It uses Hebbian learning between the input cells and the spatial pooling columns. After learning the columns should have better activation scores because the connected synapses better aligned with those active cells that the column represents.[7]

2.2.4 Temporal Memory

The Temporal Memory receives the active columns as input and outputs the active cells which represent the context of the input in those active columns. At any given time step the active columns tell what the network sees and the active cells tell in what context the network sees it.

The cells in the Temporal memory can be either active or inactive. Additionally the network's cells can be in a predictive state based on their connections, which mean an activation is anticipated in the next timestep for that cell. A cell is activated if it is in an active column and was in a predictive state in the previous time step.

The Temporal Memory should expect what is going to happen in the next time step, which columns will be activated. Going further into the column structure all the cells share the same segment input from the encoder. This means that for a given input cell in one column want to be active at the same time.

There are three scenarios for a columns cells activation:

- No cells were predicted inside the column
- Some cells were predicted inside the column
- Some cells were predicted inside a column that should not be activated in the first place

The connections in the Temporal Memory between cells are created during learning, not initialized like in the spatial pooler. When there is an unknown pattern in the previous cell activations, then new winning cells need to be chosen, and new segments formed to those winner cells before.

Bursting expresses the union of all possible context representation in a column, so expresses that the network does not know the context. To later recognize this pattern a winner cell is needed to choose to represent the new pattern the network encountered. The winner cells are chosen based on two factors, matching segments and least used cells.

- If there is a cell in the column that has a matching segment, it was almost activated. Therefore it should be the representation of this new context.
- If there is no cell in the column with a matching segment, the cell with the least segments should be the winner.

The training happens similarly to spatial pooler training. The difference is that one cell has many segments connected to it, and the synapses of these segment do not connect to the previous layer's output but other cells in the temporary memory. The learning also creates new segments and synapses to ensure that the unknown patterns get recognized the next time the network encounters them.

The synapse reinforcement would be the same as the spatial pooler if the cell were correctly predicted in the column. The segment that led to the prediction of the cell has updated the synapses that were connected to active cells will increase in permanence, and the ones that were not will be decreased. Also if there were not enough active synapses, the network grows new ones to previous active cells to ensure at least the desired amount of active synapses.

If the cell is a bursting cell, then the learning is different in this case. Since there were no segments that were activated by synapses, there needs to be a new segment that will recognize this unknown pattern. First, there is the question of which cell should be active next time the network encounters the same pattern. This is called a winner cell, and only these will take part in the learning process. Correctly predicted cells are automatically winner cells as well, so those always learn. The winner cell can be chosen in two ways. There could be equal segments attached to the cell, which means they have a slightly less activation to become activated but still pass the matching threshold. In this case, the segment almost matches the input so there only needs to be a slight change to the synapses in this matching segment, some new synapses to the previous winner cells.

In the case where there are no equal segments connected to the cell then there needs to be a new segment that will recognize this unknown pattern. The winner cells should be the one that has the least segments so that no cell would be responsible for detecting too many patterns. This is an equalizer of responsibilities across cells. The new segment will connect to some of the winner cells in the previous time step.

2.2.5 SDR scalar decoder

After the temporal memory activated the cells inside the network, we have a representation that encapsulates the given input in form of the column activations, while also encoding the context in which the network received the given input. This way in one state the whole previous sequence is encoded and a prediction can be made. The prediction is the task of the decoder, which takes an SDR input and outputs scalar values. This time the SDR input is the state of the network's cells in the temporal memory. This part of the network is not well documented, the only source is the implementation from the NuPIC package.

2.2.6 Segments and synapses

In the HTM network cells are connected through synapses and segments. Synapses start off as potential synapses. This means that a synapse is made to a cell, but not yet strong enough to propagate the activation of the cell. During learning this strength can change and above the threshold the potential synapse becomes connected. This type of learning is a Hebbian-like learning.

In the HTM network, cells are not connected directly through synapses with each other. A segment's activation is also binary, either active or not. A segment becomes active if enough of its synapses become active, this can be solved as a summation across the segments.

In the spatial pooler one cell has one segment connected to it, so this is just like in a normal neural network. In the temporal memory one cell has multiple segments connected to is. If one of these segments is activated, then the cell is activated too. This is like an or operation between the segments. One segment can be viewed as a recognizer for one SDR representation.

2.2.7 Hebbian learning

Training in the HTM network differs from other neural networks. In the network all neurons, segments and synapses have binary activations. Since this network is binary, the typical loss back propagation method will not work in this case. The learning suited for such a network is Hebbian-learning. It is a simpler but unsupervised learning method, where the learning only occurs between neighboring layers.

- Only those synapses that are connected to an active cell through a segment learn
- If one synapse is connected to an active cell, then it contributed right to the activation of that segment. Therefore its strength should be incremented
- If one synapse is connected to an inactive cell, then it did not contribute right to the activation of that segment. Therefore its strength should be decreased

2.2.8 Sparse matrices

Using sparse matrices enables to better scale the network compared to the dense matrix representation used for the previous implementation of this network.[11] Fortunately there is already a well optimized sparse package for python in the scipy.sparse package which is beneficial for the implementation. However there are some missing operators.

There are multiple ways of implementing a sparse matrix representation. There are different formats for different use-cases. There is the compressed sparse row representation or CSR. This is optimized for row based reading, for example in left multiplication. The pair of this format is the compressed sparse column format or CSC which is optimized for column reading, like in right multiplication. From these the linked list format is beneficial, because it enables the insertion of elements. In the other two cases insertion is a costly operation.

2.3 Dense HTM implementation

In my previous work I implemented a dense matrix implementation of the NuPIC HTM network.[12] This allows to treat this network the same as the other widely used and well optimized networks. While this step was necessary as a proof of concept, it is not suitable for large data, since the size of these networks is much bigger compared to the LSTM or CNN networks.

The network codes every input into SDR representations which are represented as binary vectors. The Spatial Pooler and Temporal Memory layers are connected and communicate through these vectors passing SDRs along.

The connections in this network are grouped into segments and synapses. The synapse connections have strengths, but are calculated in a binary fashion based on synapse thresholds. One segment's connections can be expressed as a binary vector, while more segments can be expressed as a binary matrix. Since in the Spatial Pooler only has one segment per columns, this representation is adequate. However in the case of the Temporal Memory every cell has multiple segments, this can only be expressed by multiple matrices, as a tensor for example.

Chapter 3

Objectives

I started my work at the beginning of 2018. I summarized the results until October 2018 in the previous Scientific Students' Conference. In that work I started to lay out my plans about rebuilding the HTM network using matrix and tensor operations using the math package for Python called NumPy. This document introduces the work done since November 2018.

My goal is to implement completely matrix based solution to observe the implications for computational and memory complexity. Due to its favourable learning properties of one-shot learning and robustness against noise, this type of network remins the focus of my research. The current dense implementation cannot yet handle vast amounts of data, but using sparse matrices should scale well with a proper solution.

Chapter 4

System design and implementation

4.1 Baseline LSTM network configuration

The baseline LSTM network consists of an LSTM layer with a 100 cells and a dense layer also with 100 cells. At the end a linear layer gives out the output. The input size is the last 100 timesteps and the network should predict the next element.

4.2 NuPIC HTM network configuration

The Nupic HTM network consists of four layers: an SDR Encoder, a Spatial Pooler, a Temporal Memory, and an SDR Decoder(/Classifier). These are configured as the default sizes by numenta as having 2048 columns, 128 cells per column and 128 maximum segments per cell in the Temporal Memory. The network receives the last timestep and should predict the next timesteps values.

4.3 Sparse HTM network design

4.3.1 Sparse Python package

The network uses the Scipy sparse package as the main package for matrix operations. This package is extended for further use in the HTM network and also to implement a Sparse tensor class. It implements all the common sparse matrix format, which are efficient in memory complexity and have little overhead in computational complexity. This computational complexity vanishes as the matrix becomes sparse enough.

4.3.2 SDR scalar encoder

The SDR encoder class has one function which return the SDR representation of a scalar input. To initialize the layer one needs to specify the following:

- Value range: This is the range where the network is capable of representing numbers. Values outside of this range will be capped.
- Number of buckets: This determines the number of buckets that the value rage is divided to. This is a measure of precision.

- Number of active cells: This is the number of consecutive active cells, that represent a value.

The size of the binary output vector is derived from the number of buckets and the number of active cells, so that the representation for the lowest value in the range gets the first bits activated and the greatest value in the range get the last bits activated. To get the SDR input for the spatial pooler only the input scalar value is needed.

Algorithm 1: The main function of the SDR scalar encoder, which turns a scalar value into an SDR representation

```
1  function getSdrInput
     ($value, numberOfBuckets, numberOfActiveCells, valueRange$);
   Input  : value: scalar input, numberOfBuckets: resolution of SDR,
            numberOfActiveCells: number of active bits in SDR, valueRange: tuple
            of the min and max accepted values
   Output: SDR scalar reprezentation as sparse row vector
2  minValue, maxValue = valueRange
3  size = numberOfBuckets + numberOfActiveCells - 1
4  widthOfBuckets = (maxValue - minValue) / numberOfBuckets
5  startIndex = floor((value - minValue) / widthOfBuckets)
6  startIndex = cap(startIndex, 0, size - numberOfActiveCells)
7  endIndex = startIndex + numberOfActiveCells
8  sdr = sparseMatrix(1, size)
9  sdr[0, startIndex:endIndex] = 1
10 return sdr
```

4.3.3 Spatial pooler

The spatial pooler receives the input from the SDR encoder and outputs the active columns as a binary vector. To initialize this class the following arguments are needed.

- Input size: the size of the SDR input vector.
- Column count: The number of columns in the spatial pooler.

Other values derived from the initializer values:

- Active column count: The number of active columns in the spatial pooler is typically the 2% of the number of columns.
- Potential synapse count: The number of potential synapses one cell has to the input cells. This is set at 50% of the input vector length.

There are also constants needed to execute the training on this module.

- Minimum synapse strength: This is the minimum value a potential synapses' strength can be, typically 0.

- Maximum synapse strength: This is the maximum value a potential synapses' strength can be, in our case it is 100.
- New synapse strength: This is the strength at which a new potential synapses' strength is initialized. It should be slightly under the synapse threshold value, in our case 40.
- Synapse threshold: If the synapse strength reaches this strengths the potential synapses is considered a connecting synapse. This value is set to 50.
- Synapse increment: This is the amount a synapses' strength is incremented if it is considered a good synapse, meaning it is a correct connection. This should be higher than the synapse decrement value, in our case it is 2.
- Synapse decrement: This is the amount a synapses' strength is decremented if it is considered a bad synapse, meaning it is a false connection. This should be less than the synapse increment value, in our case it is 1.

There are sparse matrices representing the synapse connection between the input cells and the columns, each having the appropriate sizes (input size, column count).

- Potential synapses: sparse binary matrix storing the potential synapses
- Synapse strengths: sparse matrix containing integers between 0 and 100 that corresponding to the potential synapse strengths.
- Column activity: Dense vector containing a moving average about the activity of each column.

Train

During learning all the previous steps of inference are done followed by the updating of the synapses. Since new potential synapses are not created that matrix is left unchanged. For the synapse strengths first the good and bad connections are needed to be queried.

Good synapses are potential synapses connected to active input cells and active columns. This can be translated to an inner product between the two vectors of input cells and active columns, followed by an element-wise multiplication to mask out the non-potential synapses. The bad synapses are the potential synapses that connect to an active column but aren't good synapses. So this can be executed by a simple subtraction between two binary sparse matrices.

After the good and bad synapses are determined, the synapse strength is increased by the increase amount for the good synapses and decreased by the decrease value for the bad synapses. After these the values are capped to stay within the range of synapse strengths.

Algorithm 2: The Spatial pooler's (SP) main function, which returns the active columns based on the input SDR and the synapses of the SP and also updates its synapse strenghts

```
1  function train (inputSDR);
   Input  : inputSDR: SDR representation as sparse row vector
   Output: Active columns as sparse row vector
2  connectedSynapses = synapseStrengths >= synapseTreshold
3  columnActivations = dot(inputSDR, connectedSynapses)
4  boostFactors = 1 / columnActivity
5  columnActivations = multiply(columnActivations, boostFactors)
6  activeColumnIndices = topNArgMax(columnActivations)
7  activeColumns = sparseMatrix(1, columnCount)
8  activeColumns[0, activeColumnIndices] = 1
9  columnActivity = 0.99 * columnActivity + activeColumns
10 goodSynapses = dot(transpose(inputSDR), activeColumns)
11 goodSynapses = multiply(goodSynapses, potentialSynapses)
12 badSynapses = potentialSynapses - goodSynapses
13 badSynapses = multiply(badSynapses, not(activeColumns))
14 synapseStrengths += goodLearningRate * goodSynapses
15 synapseStrengths -= badLearningRate * badSynapses
16 synapseStrenghts = cap(synapseStrengths, 0, maxSynapseStrength)
17 return activeColumns
```

Inference

The inference in the spatial pooler goes like the following. First the connected synapses are queried based on the synapse strengths being over the threshold. This results in a sparse binary matrix. Then the column activations are calculated by a matrix multiplication between the input vector and the connected synapses matrix. This results in the number of active synapses for every column. If boosting is enabled the the boosting factors are calculated, which is the inverse of the column activities capped to the 0.5 and 2 range. The activation values are then multiplied by the boosting values. From these boosted values the top n are selected, these will be the active columns.

4.3.4 Temporal Memory

The temporal memory receives input from the spatial pooler as a sparse vector of active columns. This input is still an SDR input but normalized by the spatial pooler. To initialize the class the following arguments are needed.

- Column count: The number of columns received by the spatial pooler.
- Column size: The number of cells in each column.
- Maximum segment count: This is the maximum number of segments a cell is allowed to have.

There are also derived values from the initialized values:

- Minimum synapse strength: The minimum value a synapse's strength can be, in our case 0.
- Maximum synapse strength: The maximum value a synapse's strength can be, in our case 100.
- Synapse threshold: If the value of the potential synapses strength reaches this value it is considered connected. In our case it is 50.
- New synapse strength: The value a newly grown synapse gets as strength. It should be slightly over the synapse threshold. In our case it is 60.
- Synapse increment: This is the value the good synapse strengths are incremented.
- Synapse decrement: This is the value the bad synapse strengths are decremented.
- Synapse punish: This is the value the synapses are decremented with, that were part of a falsely predictive cell's segment.
- Synapse count: The number of the desired number of the active synapses for a segment.
- Activation threshold: the number of active synapses for a segment to become active.
- Matching threshold: the number of active synapses for a segment to become matching, meaning a candidate for being part of the growing of new synapses.

The sparse matrices and tensors that represent the state of the network.

- Bursting winner cells: A sparse matrix with size (column count, column size) that represents the cells that are bursting. Bursting cells are cells that are in an active column that is not a predictive column, meaning it does not have an active segment. This means that this column was not expected to be active.
- Predictive cells before: The predictive cells in the previous time step in a form of sparse matrix with the same size as the bursting winner cells. Predictive cells are cells that have at least one active segment.
- Active cells before: The active cells in the previous time step in the same matrix form. These are the cells that were predictive and also in an active column, so they were correctly predicted.
- Winner cells before: The winner cells in the previous time step in the same matrix form. The winner cells are the cells that take part in the synapse growing phase of the training.
- Potential synapses: This is the sparse binary tensor that holds the values for the potential synapses in the shape of (max segment count, cell count, cell count). It is a stack of sparse matrices with size (cell count, cell count) which represent connection between all the cells of the temporal memory.

- Synapse strengths: This is sparse tensor with the same dimensions of the previous tensor holding the strength values for every segments every synapse between 0 and 100.

Train

The inference starts with determining the predictive cells. This starts with the synapses as well like in the spatial pooler. First the connected synapses are determined by the threshold, this results in a binary tensor. Using this tensor the segment activations can be determined by a dot product between the active cells before vector and the connected synapses tensor. This multiplication results in a matrix with size (max segment count, cell count) which contains all the activations for all the segments.

After this the active segments are selected based on the activation threshold, which results in a binary matrix with the same size as before. After this a reduction with the OR operator is needed to determine the cells which are predictive, because those have at least one active segment. This results in a vector with the size of cell count. These cells are predictive, because based on the previous active cells these cells are expected to be active by the network. This gives the temporal modeling capabilities of the network.

After the predictive cells are present the active cells are just an element-wise multiplication with the active columns. This cells that are in a column that is not active will burst all the cells in the column. This will be two matrices one with the active cells and one with the bursting cells. The active cells one is the output and the bursting cell one can be used to determine the ratio of the networks bursting rate, representing uncertainty in the network.

The next task is to get the bursting winner cells. These are the cells of the bursting ones (one per column) that will represent this new sequence that the network didn't expect and will become active the next time the same sequence is given to it. This can be split into two parts first the bursting cells that are winner because those have matching segments with the highest activation in the column, and there are cells that have no matching segment but have the least amount of segments. Using the precious activation values for segments, with reduction operations that sum across a given axis and element-wise multiplication for masking out only the bursting winner cell's segments, the bursting winner cells and those cell's segments can be determined.

Using the result of winner cells the update and growing of synapses can be executed. The good and bad synapses can be determined in a similar fashion like in the spatial pooler. The growing synapses are between the winner matching segments and the winner cells from the previous time step. This ensures that the cell will be predictive the next time the same sequence comes. This is also an inner product between the bursting winner cells' segments and the winner cells before. The synapses are not all grown to all the winner cells, it is only a random subset of those synapses, only so much that satisfy the minimum requirement for the active synapse count. Those synapses that are either good or newly grown are reinforced or added to the network, those that are bad or lead to false predictions are weakened.

After the synapses update the state of the network is saved. If the sequence is ended then these state variables are initialized empty again. This ensures a more deterministic operation for the network.

Algorithm 3: The main function of the Temporal memory, which returns the active cells based on the active columns in the spatial pooler and also chooses winner cells, updates the synapse strengths and grows new synapses

```
1  function train (activeColumns);
   Input  : activeColumns: sparse row vector
   Output: Active cells as a sparse row vector
2  connectedSynapses = synapseStrengths >= synapseThreshold
3  segmentActivation = dot(activeCellsBefore, connectedSynapses)
4  matchingSegments = segmentActivations >= matchinThreshold
5  activeSegments = segmentActivations >= activeThreshold
6  predictiveCells = reduceOR(activeSegments, axis=segments)
7  predictiveColumns = reduceOR(predictiveCells, axis=cells)
8  burstingColumns = activeColumns - predictiveColumns
9  burstingCells = repeat(burstingColumns, times=columnSize)
10 activeCells = multiply(activeColumns, predictiveCells)
11 activeCells |= burstingCells
12 winnerCells = activeCells
13 burstingSegments = repeat(burstingCells, times=numberOfSegments)
14 burstingSegmentActivations = multiply(segmentActivations, burstingSegments)
15 burstingCellActivations = max(burstingSegmentActivations, axis=cell)
16 maxSegmentActivationPerColumn = max(burstingCellActivations, axis=column)
17 winnerCells |= (burstingCellActivation == maxSegmentActivationPerColumn)
18 numberOfSegments = reduceSUM(reduceOR(connectedSynapses, axis=synapses),
     axis=cell)
19 minNumberOfSegmentPerColumn = reduceMIN(numberOfSegments,
     axis=column)
20 winnerCells |= (numberOfSegments == minNumberOfSegmentPerColumn)
21 goodSynapses = dot(transpose(winnerCellsBefore, activeCells)
22 goodSynapses = multiply(goodSynapses, potentialSynapses)
23 badSynapses = potentialSynapses - goodSynapses
24 badSynapses = multiply(badSynapses, not(activeCells))
25 synapseStrenghts += goodLearningRate * goodSynapses
26 synapseStrengths -= badLearningRate * badSynapses
27 synapseStrengths = cap(synapseStrenghts, 0, maxSynapseStrength)
28 growingSynapses = sample(dot(transpose(winnerCellsBefore), winnerCells))
29 synapseStrenths += newSynapseStrenght * growignSynapses
30 return activeCells
```

4.3.5 SDR scalar decoder

This part of the network is not well documented and the weakest part of the network compared to the other parts of the network. The other parts are either non adaptive like the decoder, or require two epochs like the temporal memory to minimize bursting and fully train on a pure input.

This is on the implementation level similar to a spatial pooler, only without boosting and only one column is allowed to become active. This one-hot encoding which has a length of number of buckets is then turned into a scalar value using the given value range and number of buckets. The training is similar to the spatial pooler as well, the synapses between the active inputs and outputs are reinforced and the ones connected to inactive inputs weakened. This part however needs more work, since it needs multiple iterations to produce results, which hinders the performance of the network.

Algorithm 4: The main function of the SDR decoder, which results in a scalar value predicting the next element in the sequence

```
1  function train (activeCells);
   Input  : activeCells: sparse matrix
   Output: Scalar value prediction
2  connectedSynapses = synapseStrengths >= synapseThreshold
3  outputActivations = dot(activeCells, connectedSynapses)
4  maxIndex = argmax(outputActivations)
5  oneHot[0, maxIndex] = 1
6  output = minValue + (maxIndex + 0.5) * bucketWidth
7  goodSynapses = dot(transpose(activeCells), oneHot)
8  goodSynapses = multiply(potentialSynapses)
9  badSynapses = potentialSynapses - goodSynapses
10 badSynapses = multiply(badSynapses, not(oneHot))
11 updateSynapses(goodSynapses, badSynapses)
12 return output
```

4.3.6 Segments and synapses

Since one cell can have multiple segments in the temporal memory, this can't be represented by a simple matrix, rather as a 3 dimensional tensor, where the first dimension represents the segments and the two other dimension are the regular input cells and output cells. This way the calculation of cell activation in the temporal memory can also be encapsulated into one simple operation between matrices and tensors.

4.3.7 Helper functions

SparseMatrix class

The sparse matrices class heavily relies on the scipy.sparse Python package which is a highly optimized implementation. There a few missing operations necessary to this network namely:

- Multiplication with a Sparse Tensor class
- Equal operator
- Add axis to get a SparseTensor class
- Repeat axis to get a SparseTensor class

- Reduce AND along an axis
- Reduce OR operator along an axis
- Reshape matrix to a 3 dimensional sparse tensor
- Reduce MAX operator along an axis
- Reduce MIN operator along an axis
- Cap matrix values to a given value range

The matrices can be stored in a number of different configurations sparsely, namely is compressed sparse row or column (CSR and CSC), linked list (LIL) etc. The different storage options are suited for different usage. The CSR representation is good for left multiplication, while the CSC for right multiplication. Both are equally efficient on other operators. When manipulating the matrix values independently the LIL matrix format is the best, however all the other operations are slower using this type. The conversion between these is efficient, so every time a matrix can be converted to best suit the given operation needed to execute.

SparseTensor class

Implementing the network using matrix operations raises some problems at the temporal memory level. There each cell can have multiple segments which have synapses that connect to other cells. This connection can not be expressed in a matrix only one segment per cell fits in a matrix representation just like in the spatial pooler. For this an additional dimension is needed, which results in a tensor-like structure. In our case it is a stack of sparse matrices where the stack in itself is also a sparse structure. For this a dictionary is perfect, where the first index of the tensor is the key and the reference to the sparse matrix is the value stored. This way a well optimized package can be extended to handle three dimensions. All this is wrapped in the class SparseTensor, which uses the stack of SparseMatrix objects previously implemented.

There is an implementation of HTM network by Numenta called NuPIC, but it is not an optimized code, more of a proof of concept prototype. The goal of my work is to apply massively parallel training possible in HTM networks. Thus, it will be more scaleable to more complex tasks. Furthermore, I investigate the sequence learning capabilities of an HTM network against an LSTM baseline network. The training data is a multidimensional superposed sinusoidal signal with added noise.

The SaprseTensor class is used to for the Temporal memory part of the network where one cell can have multiple segments. This require an additional dimension to the usual two dimension stored in a matrix. So in this case out tensor class is compiled from a stack of sparse matrices. To also enable sparsity in the additional dimension it is a dictionary, where the key is the first index of the matrix stored in the stack. This dictionary only stores those matrices which have at least one nonzero values, so the overhead is minimal. Once a value is added that correspond to an index which is nonexistent yet in the matrix a new matrix is added with the appropriate index. When a matrix has all of it's values zeroed out then the matrix is removed from the dictionary sparing resources in future operations. The operations can be for the most part easily propagated back to the SparseMatrix classes operations. The most typical implementation is to iterate over the sparse matrices of the sparse tensor and call the appropriate methods of the matrix class and then collect the results to a new tensor. Otherwise like when reshaping a tensor to a matrix aside from the iteration of the sparse matrices a reindexing and reorganizing of the elements are needed. The reduction operations like reduce or and reduce max are simple iterations when made on the first axis and matrix concatenation when the second or third axes are selected. Overall there are room for optimization still in these operations, the need for new matrix creation and the length of certain for loops can be shortened, but this solution shows that a fully sparse implementation is possible and scales well in terms of space complexity.

Chapter 5

Results

In this paper four network types were studied: the LSTM, the NuPIC HTM, the dense HTM and the sparse HTM networks. The three of them were studied in terms of measurements too. The results were measured in training time and by observing the predictions visualized. The training times are the following.

Configuration	Epochs	Timesynth	Hot Gym	Audio
LSTM	100 epochs	240s	310s	440s
NuPIC HTM	1 epoch	230s	250s	260s
Sparse HTM	1 epoch	500s	510s	530s

Table 5.1: *Training times*

The predictions look the following:

As it is seen on these images the LSTM does learn the basic patterns, but seems to not follow well enough the spikes. It has a rounding effect where it flattens out the sudden spikes in this real world data.

In the case of the HTM the sudden spikes are predicted much more, even if those are overshot. This shows a completely different behaviour, it it not a rounding effect, the network matches the current values with previous existing patterns and copies the predictions from there.

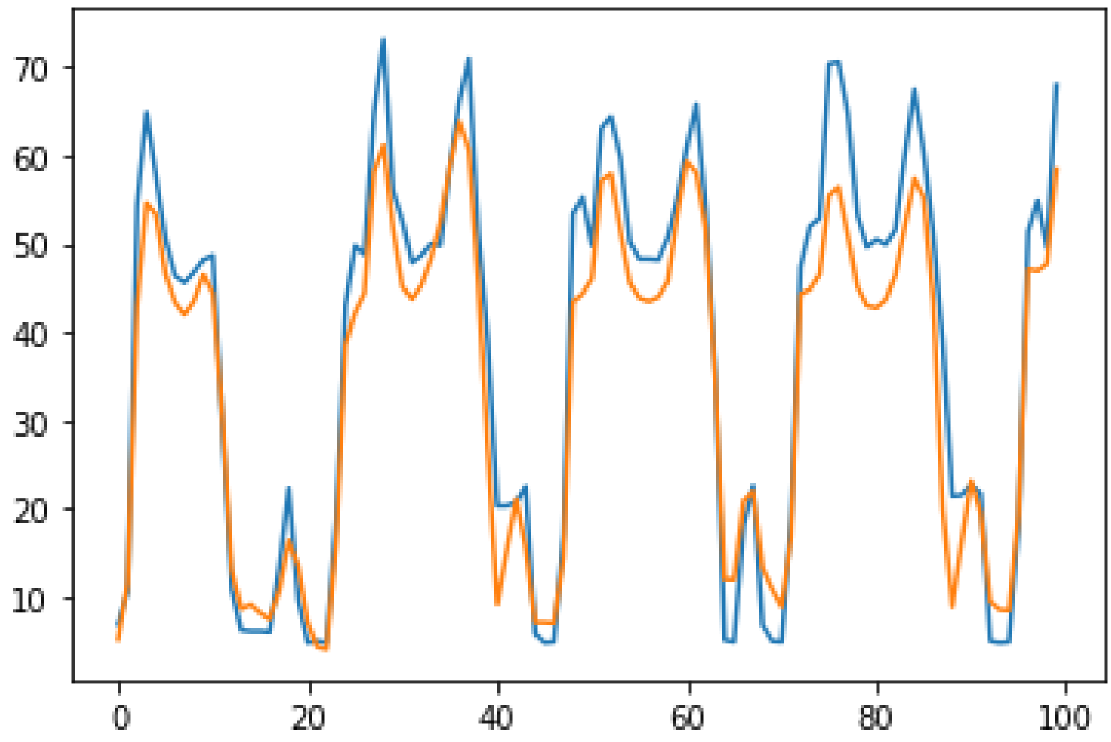
Figure 5.1: *Temporal Memory predictions for Hot Gym dataset (sample)*

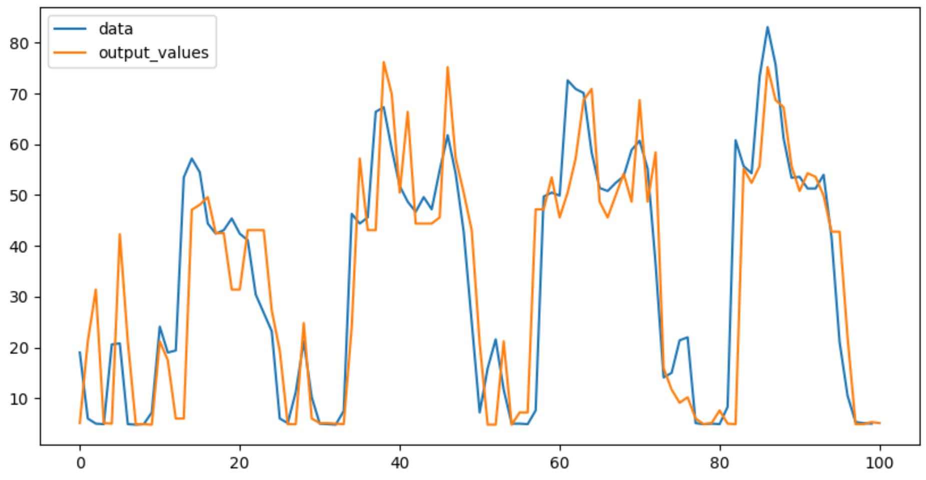

Figure 5.2: *HTM predictions for Hot Gym dataset (sample)*

Chapter 6

Summary

In conclusion I observed the solutions connected to sequence learning with deep learning neural network. I presented the baseline as the single layer LSTM network and compared it to the other baseline Numenta implementation and to my previous dense HTM implementation. During this paper I presented the methodology of turning the implementation of this network to a matrix operation based one. Finally I showed that the network is capable of producing one-shot learning and that this type of learning is on par in training time with the 100 epoch LSTM training time. In the future I would like to observe more variables about these network, but those are beyond the scope of this paper.

Acknowledgement

The research presented in this paper has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Abbreviations

- RNN: Recurrent Neural Networks
- LSTM: Long Short-Term Memory
- HTM: Hierarchical Temporal Memory
- SDR: Sparse Distributed Representation
- SP: Spatial Pooler
- TM: Temporal Memory

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] François Chollet et al. Keras. `https://keras.io`, 2015.

[3] Javier Contreras, Rosario Espinola, Francisco J Nogales, and Antonio J Conejo. Arima models to predict next-day electricity prices. *IEEE transactions on power systems*, 18(3):1014–1020, 2003.

[4] Sean R Eddy. Hidden markov models. *Current opinion in structural biology*, 6(3):361–365, 1996.

[5] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin. Biological and machine intelligence (bami). Initial online release 0.4, 2016.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Richard Kempter, Wulfram Gerstner, and J Leo Van Hemmen. Hebbian learning and spiking neurons. *Physical Review E*, 59(4):4498, 1999.

[8] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[9] Numenta. Htm school, 2018.

[10] Numenta. Numenta webpage, 2019.

[11] Pilinszki-Nagy. Accelerating learning with hierarchical temporal memory. 2018.

[12] Pilinszki-Nagy. Optimizing hierarchical temporal memory for sequence learning. 2018.

[13] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[14] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[15] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.