Information Technology Course
Module Software Engineering
by Damir Dobric / Andreas Pech

Scalar Encoder with Buckets

Sahith Kumar Singari
sahith.singari@stud.fra-uas.de

Anil Kumar Gadiraju
anil.gadiraju@stud.fra-uas.de

Vinay Kumar Bandaru
vinay.bandaru@stud.fra-uas.de

***Abstract*— Scalar encoders with buckets are useful in a variety of real-world scenarios where continuous values must be represented as discrete values. This is frequently used in case machine learning and data analysis, like to convert a continuous characteristic such as age, temperature, or height into a categorical feature that may be used in a model or algorithm. Sensor data analysis is one common application for scalar encoders. Without Buckets concept it is less versatile, Limited Precision and Lack of adaptability. Which means there will be limited accuracy of the scalar encoder results from the mapping of continuous input values to a discrete set of data and less adaptable with more difficult to modify to different datasets because it requires the user to set parameters like the number of min/max values, and radius. Moreover ,Non-periodic encoding is a limitation of the Scalar Encoder that may be problematic for particular Input data's. So, this paper evaluates the limitations of the Scalar Encoder and how they can be addressed by combining the Scalar Encoder with Buckets.**

I. Introduction

Scientists have gained insights by working on the cortex that sequence learning has large invariant changing series of inputs. The exact neural mechanism of sequence memory is still unknown, but models that give a reading of the neurons are used to study. These models show significant capabilities to recollect and recognize the sequence of inputs using rules.

Hierarchical Temporal Memory (HTM) is a Biomimetics model based on the principles of memory predictions developed by scientists to capture the architectural and algorithmic features of the neocortex [1] [5]. HTM has given promising results in pattern recognition, and This can learn the temporal sequences and spatial flow of sensory inputs as data.

Using a scalar encoder with buckets, continuous data can be transformed into a set of discrete values that can then be utilized for analysis, modeling, or machine learning. With this method, the range of continuous values is divided into a number of discrete intervals, or "buckets," and each value is then assigned to the bucket that it belongs in. The encoding procedure can be accomplished in a variety of ways and customized to the unique requirements of the application, for as by employing a binary or multi-level encoding scheme. In applications including sensor data analysis, natural language processing, and picture classification, scalar encoder with buckets are frequently employed and have shown to be effective tools for transforming continuous data into discrete data.

II. Literature Survey

A. *Hierarchical Temporal Memory (HTM)*

The HTM model learns the procedure that occurs in one cortex of the brain. HTM works on continuous streams of input patterns, attempting to construct rare and constant representations of input sequences based on the input stream's recursive pattern. [1]

HTM's capacity to forecast future patterns based on previously trained data patterns. After a few cycles, HTM receives a unique pattern that compares the prior patterns to the current pattern. Input patterns should not repeat, and the uniqueness should be maintained.

To depict patterns in the incoming data, HTM uses SDRs. Each input pattern is first transformed into an SDR, and the HTM network is then fed this SDR.

SDRs enable the efficient and effective encoding and processing of complicated and high-dimensional data patterns, which is why they are fundamental to the HTM algorithm.

B. *Sparse Distributed representations (SDRs)*

Sparse Distributed representations (SDRs) of input patterns are used in HTM's language. With a set amount of active bits, it produces SDRs internally. These bits have semantic value. As a result, two inputs with equivalent semantic meaning must have equal active bit representation in SDR, which plays an important role in HTM learning.

Hierarchical Temporal Memory (HTM) technique is based on the concept of SDRs, which are high-dimensional binary vectors with only a small fraction of the bits set to 1. SDRs are a natural way for the brain to represent patterns because they allow for the efficient storage and processing of large amounts of information.

The encoding process is similar to the operations of human and other animal sensory organs.

The cochlea, for example, is a specialized mechanism that translates the frequencies and amplitudes of external sounds into a sparse set of activated neurons. The underlying mechanism for this process (Fig. 1) consists of a row of inner hair cells that are responsive to different frequencies. When a specific frequency of sound is heard, the hair cells excite neurons, which send the signal to the brain. The set of neurons that are stimulated in this manner form the Sparse Distributed Representation of the sound.

An encoder in an HTM system initially turns a data source into an SDR.To capture the main semantic properties of the material, the

encoder selects which output bits should be ones and which should be zeros. SDRs with similar input values should have a high degree of overlap. [3]

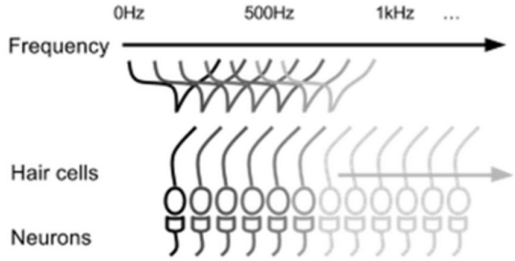

Figure.1: Cochlear hair cells excite a group of neurons based on the frequency of the sound.

C. Encoders

Encoders should generate SDRs with a constant number of bits 'N' and a fixed number of active (1's) bits 'W', regardless of what they represent. What do you know about the ideal values for N and W?
To preserve the features of sparsity, we cannot be a large proportion of N. But, if W is too small, we lose the features of a distributed representation.
While encoding data, there are several unique aspects to consider:
1. Semantically related data can trigger SDRs with overlapping active bits.
2. The same input should always provide the same SDR output.
3. The result will have the same dimensionality as all of the inputs (total number of bits).
Deterministic encoders should produce the same result from the same input each time. Without this attribute, the sequence learnt in HTM will be redundant because there is a shift in values with encoded representations. Put an end to creating adaptive or random element encoders.
The output of an encoder must produce the exact same number of bits for each of its inputs. SDRs are compared and handled so that a bit with a specific "value" is always at the same location using a bit-by-bit assumption. If the encoders offered various SDR bit lengths, comparisons and other operations would not be possible. [4]

III. Methodology

The project Scalar Encoder with Buckets is developed using C# .Net Core in Microsoft Visual Studio 2022 IDE (Integrated Development Environment) is used as a reference model to understand the functioning of Scalar Encoder which uses.

Scalar encoder maps continuous scalar data onto a defined range of integers to encode them into sparse distributed representations (SDRs). To calculate the SDR's bit count and bucket width, the encoder must first calculate a scaling factor and a resolution. The scalar value is then encoded as an SDR using a one-hot encoding technique, rounded to the nearest integer. A scalar encoder that translates values into buckets that are evenly spaced apart and whose widths are fixed is known as one with fixed bucket width.

Scalar encoder with buckets is another type of scalar encoder that transfers continuous scalar data onto buckets of different lengths. In contrast to a scalar encoder with set bucket width, each bucket's width is determined by the data distribution. A clustering technique is used by a scalar encoder with buckets to classify similar values into buckets of the same size. If the data distribution is unequal, this encoding method may be more effective than a scalar encoder with a set bucket width.

In the context of data encoding, fixed width refers to a sort of scalar encoding in which the range of input values is divided into a fixed number of identically sized buckets or bins. Each bin represents a specific range of values, and the input value is assigned to the bin corresponding to the range in which it falls.

The benefit of using a fixed width scalar encoder is that it is straightforward and easy to construct, and the output representation of the input values is consistent and predictable.

The Scalar Encoder contributes to the creation of a structured representation of the input data that can be efficiently processed by the Spatial Pooler, resulting in the formation of sparse distributed representations that are noise-resistant and generalizable to new inputs.

A. Spatial Pooler

The active columns' cells are mapped during the creation of SDR input by Spatial Pooler. Each column connects to the following section of input bits via a network of synapses. While many columns would have the same appearance, these columns are distinct from one another. Varying patterns result in varied levels of activation, and stronger activation limits weaker activation of the columns. Columns may cover a little portion of the space or the entire surface. Implementing the inhibitory mechanism results in a constrained representation of the input. Similar patterns result in similar activation columns. HTM learns from the input and breaks down cell connections. Learning results by updating synapse persistence (Fig 2). If we look into the Active bits in the active columns enhance the persistence value, while the other columns make it smaller. Inactive columns do not pick up new information.. The spatial pooler implies collections or clusters of spatially related data. During the spatial pooler's learning process, every pattern that manifests at the input is compared to the database of other patterns. The other columns make it smaller. Inactive columns do not pick up new information. [3]

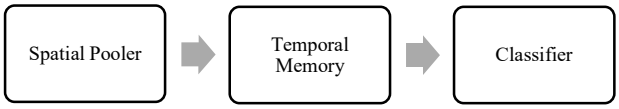

Figure. 2: HTM Algorithm Flow

B. Sparse Distributed Representation

In the HTM, SDR is an effective information organization system. Sparse means that only a tiny percentage of the big, interconnected cells are active at any given time. "Distributed" denotes that active cell are dispersed throughout the region and will be used to depict the region's activity. Because the binary representation is more biologically reasonable and highly computationally efficient, HTM considers the binary SDR converted from a specific encoder. Even though the number of possible inputs exceeds the number of possible representations, the binary SDR does not result in a functional loss of information due to the following critical features of the SDR. [1]

C. Scalar Encoder

Scalar Encoder is a type of encoding method used in Hierarchical Temporal Memory (HTM) systems. It is designed to convert a continuous input value into a binary representation that can be easily processed by the HTM.

The scalar encoder without buckets works by dividing the input range into a set of equally spaced intervals or "steps". Each interval is represented by a binary value, with a 1 indicating that the input value falls within that interval and a 0 indicating that it does not.

The number of steps or intervals used to encode the input value can be adjusted based on the desired level of granularity. A higher number of steps will result in a more fine-grained representation, while a lower number of steps will result in a more coarse-grained representation.

Scalar Encoder without buckets is useful for encoding continuous scalar values such as temperature, pressure, or speed into binary representations that can be processed by the HTM. It is a simple and efficient way to transform real-world data into a format that can be analyzed by the HTM. However, it does not consider the distribution of the input values and may not work well in situations where the input values are heavily skewed or non-uniform. It is the reason we introduce the buckets in it. over that kind of situation.

D. Scalar Encoder with Bucket Concept

Buckets, a predetermined number of uniformly sized intervals or "buckets" are used to partition the range of continuous values. The width of the interval, which is equal to the difference between the maximum and minimum values divided by the number of buckets, establishes the size of each bucket.

For instance, if we wanted to generate 10 fixed-width buckets and we had a range of continuous values from 0 to 100, we would first determine the width of each bucket: [4]

Bucket width = (100 - 0) / 10 = 10 copy the code
This implies that each bucket will have a width of 10 units. Here is how the 10 buckets would be described [0, 10], [10, 20], [20], [30], [40], [40], [50], [60], [60], [70], [80], [80, 90], [90, 100]
The interval boundaries are inclusive at the lower end and exclusive at the top end, as you can see. A square bracket [indicates that the endpoint is included while a parenthesis] indicates that the endpoint is excluded. This is a widely used convention in interval notation.

Each continuous value is assigned to the appropriate bucket based on its value after the buckets have been defined. For instance, as it lies between 40 and 50, the bucket [40, 50] would be given a value of 42.

Fixed-width bucketing is a straightforward and easy-to-use scalar encoding technique that works well with a variety of data formats. However, given that some buckets may include data that is not evenly distributed, it might not be the best option for greater than others in terms of values. Adaptive or variable-width bucketing may be a better option in some circumstances.

IV. IMPLEMENTATION

To build an effective encoder that can result in similarity, you must comprehend the components of your data.
In the afore mentioned hearing frequency example, the encoder was designed to have noises with a comparable pitch but did not account for how noisy the sounds were, which would call for a different strategy. The first step in designing an encoder is to choose whatever data element you want to capture.

Pitch and amplitude may be two of a sound's primary characteristics, whereas the weekend status of a date may be one. When two inputs have the same values for one or more of the selected data attributes, the encoder will produce overlapping representations of those inputs.

Moreover, there must be enough one-bits to account for subsampling and noise. Having one bit of at least 20 to 25 is a good general rule of thumb. Because of the noise and non-determinism in HTM systems, encoders with representations less than 20 one-bits will perform poorly and be more prone to errors.

We first divide the range of values into buckets when building an encoder implementation, and then we map the buckets into a group of active cells.
1) Decide on the value range (MinVal and MaxVal).
Range =(MaxVal - MinVal)* 2.
3) The "width" of the output signal "W"—the amount of bits that are set to encode a single value—should be determined.
4) The output's total number of bits, "N," should be chosen as the representational bit count.
5) A periodic or non-periodic parameter should be used to calculate the resolution.
6) Starting with N unset bits and setting them one at a time results in the encoded representation.

A. Scalar Encoder with Bucket Implementation

A scalar encoder converts a numeric (floating point) value into a bit array. Except for a contiguous block of 1's, the output is all 0's. The placement of this contiguous block changes in real time as the input value changes.
Linear encoding is used. If you want a nonlinear encoding, simply alter the scalar before encoding (e.g., with a logarithm function).
Binding the data as a pre-processing step, such as "1" = $0 - $.20, "2" = $.21-$0.80, "3" = $.81-$1.20, etc., is not advised because it removes a lot of information and prevents neighboring values from overlapping in the result. Instead, employ a continuous transformation that scales the data (a piecewise transformation is fine)...otherwise, "MaxVal " is a valid upper bound.
param w: The number of bits assigned to encode a single value - the "width" of the output signal restriction: w must be odd to avoid centering difficulties. **MaxVal :** The maximum value of the input signal. (If "periodic Equals= True," input is strictly smaller) **periodic**: If true, the input value "wraps around" so that "MinVal" = "MaxVal ". The input for a periodic value must be strictly less than "MaxVal ," otherwise "MaxVal " is a valid upper bound.
The number of bits in the output is specified by the parameter n. "**w**" must be more than or equal to the representations of two inputs separated by more than the radius are non-overlapping. In general, two inputs separated by less than the radius will overlap in at least some of their bits. This can be thought of as the radius of the input. param resolution: Two inputs separated by larger than or equal to the resolution will always have different representations. name: an optional string that will be included in the description: If clip Input is set to true, non-periodic inputs less than MinVal or greater than MaxVal are trimmed to MinVal/MaxVal .
If true, skip several safety tests (for compatibility reasons), otherwise false. Please keep in mind that "radius" and "resolution" are specified in relation to the input, not the output. "w" is given in relation to the output.
GenerateRangeDescription: The GenerateRangeDescription method generates a description of the bucket ranges used to encode the input data, which can help users understand the encoding scheme and adjust the parameters as needed. This increases the flexibility of the Scalar Encoder with Buckets and makes it easier to adapt to different datasets.

GetBucketIndex: The Scalar Encoder with Buckets provides an improved encoding scheme compared to the Scalar Encoder by using continuous ranges of buckets instead of individual buckets.
The GetBucketIndex method returns the index of the bucket range that an input value belongs to, allowing for a more efficient encoding of data.

The GetBucketIndex method returns the index of the bucket range that an input value belongs to, allowing for a more efficient encoding of data. Subclasses must override this.
returns a list of things, one for every bucket that this encoder has specified.
Each item represents the value assigned to that bucket; it has the same structure as the input that would be returned by the GetBucketInfo method.
If all you need are the bucket data, this call is quicker than calling: meth:'. GetBucketInfo' on each bucket separately.
return: a list of things, each item corresponding to a bucket's value.
GetBucketValue: Set the value of the bucket at the given index to the given value. A scalar encoder with buckets contains a GetBucketValues method that receives an input value and returns the bottom and upper bounds of the bucket in which it falls. The approach looks for edge cases first, including NaN and infinity values as well as numbers outside the encoder's range.
The method then sets the bucket count to 100 and determines each bucket's width by dividing the encoder's range of values by the bucket count. When it detects any invalid bucket widths, such as infinity, NaN, or negative values, it throws an exception.

By subtracting the minimum value of the encoder from the input value, dividing the result by the bucket width, and rounding to the closest integer, the algorithm then determines the index of the bucket into which the input value belongs.

Then, it determines the bucket's lower and upper bounds by multiplying the bucket index by the bucket width, adding the encoder's lowest value to determine the lower bound, and adding the width to get the upper bound.

The procedure then produces an array with double values representing the bucket's lower and upper boundaries.

GetBucketInfo: The information about the associated bucket in the scalar encoder is included in an int array that is returned by the GetBucketInfo method, which accepts a double input value.

By determining if the input value is more than the maximum value or less than the minimum value, the input value is first clipped to the range of the scalar encoder.

After dividing the encoder's range into N buckets, the algorithm determines which bucket the input value belongs to before calculating the index of that bucket. As the bucket index is determined by dividing by integers, it will always be an integer.

The algorithm then determines the center of the bucket by multiplying the bucket's initial value by half its width. Moreover, it determines the bucket's beginning and ending values.

In the case of a periodic encoder, the approach wraps the bucket index if it exceeds N buckets. Then, accounting for the periodicity of the encoder, it determines the distances to the closest bucket edges. The center of the bucket is determined by the method using the nearest edge.

A rounded bucket center value, rounded bucket starting value, rounded bucket ending value, and an int array with the bucket index are all included in the method's final output. These values can be used to represent the input value in the scalar encoder as an array of binary digits.

An encoder is a program that converts a value to a sparse distributed representation.
This is the foundation class for encoders that are OPF compatible. For use in locations like the SDR Classifier, the OPF requires that values be represented as a scalar value.

EncodeIntoArray: The method EncodeIntoArray in the Scalar Encoder with Buckets maps input values to continuous ranges of buckets instead of individual buckets, allowing for better precision in encoding data. It encodes input Data and writes the encoded value to the 1-D array of length .
param output: 1-D array of the same length.
get encoded Values: gives the input back in the same format as the method ".topDownCompute". This is the same as the input data for the majority of encoder types. For instance, this corresponds to the string and numeric values from the inputs for the scalar and category types, respectively. This gives the list of scalars for each of the sub-fields for datetime encoders (timecode, dayOfWeek, etc.)
return: a list of values with the same structure and arrangement as the values produced by the "topDownCompute" method.
returns an array containing the input Data's input sub-fields' bucket indices. To acquire the related field names for each of the buckets.
param input Data: The data from the source. Usually, this is a member of an object. return: array of bucket indices. encodeIntoArray"
Given that a new array is allocated with each call, this might be less efficient.
input Data: The input data that needs to be encoded.
return: an array containing the input Data's encoded form.

ClosenessScores: Calculate ratings of proximity between the expected and actual scalar values.
For each value in expValues (or actValues, which must be the same length), this method provides a single closeness score. The proximity score runs from 0 to 1.0, with 1.0 representing the best possible match and 0 representing the poorest one.
If this encoder is a straightforward one-field encoder, it will only accept one item in each of the "actValues" and "expValues" arrays. One item per sub-encoder is what multi-encoders anticipate.
Each type of encoder has its own proximity metric that can be defined. A category encoder, for instance, might return either 1 or 0.

B. Testcases Methods of Scalar Encoder

ScalarEncodingGetBucketIndexPeriodic: These methods appear to be testing the Scalar Encoder class's GetBucketIndex method, which returns the bucket index for a specified scalar value. With a non-periodic encoder, the first method, ScalarEncodingGetBucketIndexNonPeriodic, generates a bitmap image for a set of scalar values and their corresponding bucket indices. For a periodic encoder, the second method, ScalarEncodingGetBucketIndexPeriodic, accomplishes the same task. These techniques appear to be helpful for observing how the bucket indices alter depending on the encoder parameters and scalar values. **Nonperiodic:** Using a scalar encoder with non-periodic settings, a scalar value is encoded, and for each encoded scalar value, the encoder's GetBucketIndex method is used to retrieve the bucket index. The encoded output is then displayed as a 2D array bitmap, with green pixels designating active bits. The bitmap is then saved to a file with a name derived from the scalar value.

TestGenerateRangeDescription:

The Scalar Encoder class's GenerateRangeDescription method, which accepts a list of tuples representing ranges of values and produces a textual description of those ranges, is tested by the TestGenerateRangeDescription method.

First, the function constructs a Scalar Encoder class instance with a range of 0 to 100 with periodicity set to true. Asserting that the

created string descriptions match the anticipated values, it then generates three separate lists of value ranges.

Two ranges are specified in the first test case: 1.0 to 3.0 and 7.0 to 10.0. The two ranges are separated by a comma and are each represented by its lower and higher boundaries, formatted to two decimal places, in the expected string description of "1.00-3.00, 7.00-10.00."

In the second test scenario, a single value is specified as a range of 2.5 to 2.5. The string description that should be used is "2.50," which merely denotes a single value with two decimal places.

Two ranges are specified in the third test case: 1.0 to 1.0 and 5.0 to 6.0. The two ranges are separated by a comma, and the intended string description is "1.00, 5.00-6.00," which represents the first range as a single value and the second range as its lower and upper bounds formatted to two decimal places.

ClosenessScorestest: The Scalar Encoder class ClosenessScores method is being tested in this test. The Scalar Encoder is created with a set of parameters, two arrays of expected and actual values are defined, fractional is set to true, and the expected closeness score is set to 0.99. With these parameters, it then invokes the ClosenessScores function and verifies that the outcome is within 0.01 of the predicted closeness score. This test case verifies that the Scalar Encoder is used by the ClosenessScores function to determine the closeness score between the expected and actual values.

ScalarEncodingEncodeIntoArray: This test method examines the Scalar Encoder class EncodeIntoArray function. The method requires a Boolean indication indicating whether or not the encoder should learn, an output array, the length of the output array, and an integer input value. The test method calls the EncodeIntoArray method for a range of input values after creating an instance of the Scalar Encoder class with certain default settings.

The method checks each index of the array for each input value to ensure that the encoded input value has been successfully populated into the output array. Moreover, a bitmap of the encoded value is created and saved to disk. The method checks sure the EncodeIntoArray function returns zero before concluding.
This test method effectively covers the EncodeIntoArray method and validates that input values are appropriately encoded into an output array. Additionally, it confirms that the procedure produces the desired outcome. To make sure that the approach is effective for a variety of inputs, extra edge cases and input values could be tested as part of the test procedure.

Scalar Encoding Decode: This unit test is for the Scalar Encoder class decode function. The decode method extracts the original scalar value from an array of integers that represent a scalar value that has been encoded with the encoder.

The test cases use the decode method to obtain the original scalar value and contain 8 different sets of encoded values (output1 through output8). The scalar's lowest and maximum values, the number of bits utilized for encoding, the size of each bit, and whether or not the encoding is periodic are all test parameters.
Each of the eight test scenarios is iterated through in the foreach loop, which prints the encoded output and the decoded input for each case. This makes it possible to manually check that the decode method is working correctly.
ScalarEncoder_EncodeIntoArray_RangeOfInputValues_ReturnsCorrectArrays: This code looks to be testing the Scalar Encoder class EncodeIntoArray function. With an increment of 0.1, it loops through a set of input values from MinValue to MaxValue. It uses the EncodeIntoArray function to retrieve the intended array for each input value, then compares it to the actual array it gets using the same method.

It makes a bitmap image with the expected array in green and the actual array in grey (or red if they are not equal) and saves it to a folder if the expected and actual arrays are not identical. The intended and actual arrays, as well as the input value, are all indicated in the image's accompanying text. This test checks that the Scalar Encoder class's EncodeIntoArray function returns the right arrays for a variety of input values.

GetTopDownMapping:

In Periodic Encoding the Scalar Encoder with Buckets supports periodic encoding of values, allowing for better handling of cyclical data. The GetTopDownMapping method generates a mapping of the bucket ranges to a hierarchy of levels, which can be useful for representing cyclical data such as time of day or day of the week.

The top-down mapping of the input value to the encoder's buckets is represented via an integer array in this code by the public method _getTopDownMapping.

The three arguments required by the method are input, a Boolean flag indicating whether the encoder is periodic or not, Periodic, and NumBuckets, which denotes the number of buckets in the encoder.

If Periodic is true, the method first determines the bucket index that the input belongs in before determining the bucket width as 1/NumBuckets. After that, it loops through each bucket, measures the distance between the input and the one being used, and either sets the appropriate mapping value to 1 (if the distance is less than or equal to half the bucket width) or 0 (if it is greater than that distance).

If Periodic is false, the method first checks whether the radius parameter is set. If it's not, it calculates the radius as half the bucket width. It then calculates the index of the bucket that the input falls into, and loops through all the buckets. For each bucket, it calculates the distance between the input and the start of the bucket, and sets the corresponding mapping value to 1 if the distance is less than or equal to half the bucket width, or 0 otherwise.

Finally, the method returns the mapping array representing the top-down mapping of the input value to the encoder's buckets.

GetBucketInfo:

The code is a unit test for the Scalar Encoder class GetBucketInfo function. Using a number of binary bits, the Scalar Encoder is used to encode scalar values into a sparse distributed representation (SDR). A scalar value is passed into the GetBucketInfo method, which returns details about the bucket into which the data belongs.

Using a set of parameters that specify the encoding process, the first test, TestGetBucketInfoNonPeriodic, produces an instance of the Scalar Encoder. The test then looks for a set of scalar values in the bucket information given by the GetBucketInfo function. For values close to the bucket boundaries, outside of the range of valid values, and in the middle of the range of valid values, the test verifies that the bucket information is as expected.

The second test, TestGetBucketInfoPeriodic, is similar to the first test but with the Periodic parameter set to true. This means that the encoding process treats the range of scalar values as a periodic range, where values at the upper and lower bounds of the range are

considered adjacent. The test checks the bucket information returned by the GetBucketInfo method for a set of scalar values, including values outside of the range of valid values.

The Console. WriteLine statements are used to print the bucket information for each value to the console for debugging purposes. The CollectionAssert.AreEqual statements are used to compare the expected bucket information with the actual bucket information returned by the GetBucketInfo method, and throw an exception if they do not match.

V. TestCase with Results

A. Testcase-GetBucketValues:

The code is a unit test for checking the functionality of the Scalar Encoder class GetBucketValues() method. A class called Scalar Encoder converts scalar values into a binary representation that HTM (Hierarchical Temporal Memory) networks can use. This test checks that the GetBucketValues() method throws an exception if the input value is invalid and returns the desired result for the provided input value.

The test creates an instance of a Scalar Encoder with the following configuration settings:
W: 21,N: 1024,Radius: -1.0,MinVal: 0.0,MaxVal: 100.0
Periodic: false, ClipInput: false, NumBuckets: 100
The MinVal and MaxVal parameters specify the minimum and maximum input values that the encoder can handle, the Periodic parameter determines whether the encoder should wrap around at the boundaries, the Name parameter specifies a name for the encoder, the ClipInput parameter specifies the radius of the neighborhood around each bucket, the W parameter specifies the width of each bucket, the N parameter specifies the number of bits in the output encoding, the Radius parameter specifies the radius of the neighborhood around each bucket, the Name parameter specified.

The test then calls the GetBucketValues () function, confirms that it throws an Argument Exception, and calls it with the invalid input value of -10.0.
Using 47.5 as a valid input number, the test then calls the GetBucketValues() function, stores the output in the bucket Values variable, prints the actual and expected bucket values to the console for debugging, and checks to see if the output matches what was anticipated by [47, 48].

Overall, this unit test offers a thorough technique to ensure that the Scalar Encoder class GetBucketValues() method operates as anticipated given a range of input values and setup options. The obtained details of testcase are shown in Fig 3.

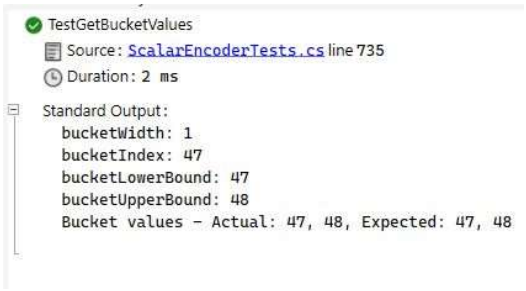

Fig3: Expected output &Actual Output of the Test Bucket Values

B. Testcase-ScalarEncodingGetBucketIndexNonPeriodic:

This code is an experiment to encode a set of numbers and produce bitmap output. It encodes a decimal number using a Scalar Encoder object with non-periodic settings. The encoder has the following settings:
"W": 21 - The width of the output vector.
"N": 1024 - The number of bits to use to encode the input range.
“Radius": -1.0 - The radius of the output vector.
"MinVal": 0.0 - The minimum value of the input range.
"MaxVal ": 100.0 - The maximum value of the input range.
"Periodic": false - Whether the encoder should wrap values around the ends of the input range. “ClipInput": false - Whether the input values outside the input range should be clipped to the minimum or maximum value. The code then loops through a range of decimal numbers, encodes each number using the encoder, and gets the bucket index of the number using the GetBucketIndex() method of the encoder. The result is then displayed as a bitmap image, with the value of the input and its bucket index as text. The bitmap image is saved in a folder named "ScalarEncodingGetBucketIndexNonPeriodic". The obtained details of testcase are shown in Fig 4.

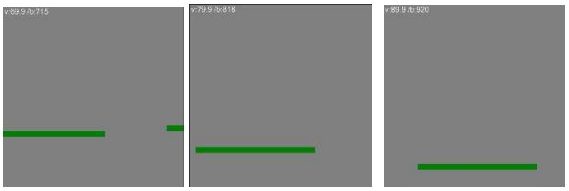

Fig4:The Actual output of the Testcase GetBucketIndexNonPeriodic.

C. TestCase- ScalarEncodingGetBucketIndexPeriodic

This code is an experiment to encode a set of numbers and produce bitmap output. It encodes a decimal number using a Scalar Encoder object with periodic settings. The encoder has the following settings:
"W": 21 - The width of the output vector.
"N": 1024 - The number of bits to use to encode the input range.
"Radius": -1.0 - The radius of the output vector.
"MinVal": 0.0 - The minimum value of the input range.
"MaxVal ": 100.0 - The maximum value of the input range.
"Periodic": true - Whether the encoder should wrap values around the ends of the input range.
"ClipInput": false - Whether the input values outside the input range should be clipped to the minimum or maximum value. The code then loops through a range of decimal numbers, encodes each number using the encoder, and gets the bucket index of the number using the GetBucketIndex () method of the encoder. The result is then displayed as a bitmap image, with the value of the input and its bucket index as text. The bitmap image is saved in a folder named "ScalarEncodingGetBucketIndexPeriodic". The obtained details of testcase are shown in Fig 5.

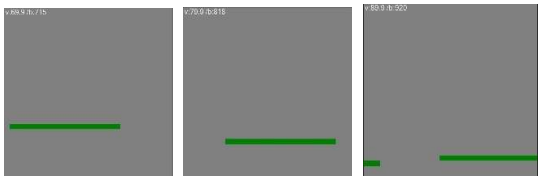

Fig5: The Actual output of the testcase GetBucketIndexPeriodic.

D. Testcase-GenerateRangeDescription:

The Scalar Encoder class GenerateRangeDescription () function. The method takes a list of ranges, where each range is represented by a tuple of two doubles and produces a string with a human-readable description of those ranges.
In the test, the GenerateRangeDescription () method receives three different sets of ranges as input. The Assert.AreEqual() method compares the method's actual output to the predicted output for each combination of ranges.
The tuples (1.0, 3.0) and are included in the first set of ranges (7.0, 10.0). The desired results are "1.00-3.00, 7.00-10.00."
The tuple is the single item in the second set of ranges (2.5, 2.5). The anticipated result is "2.50."The tuples (1.0, 1.0) and are included in the third set of ranges (5.0, 6.0) if all the expected outputs match the actual outputs, then the test passes. The obtained details of testcase are shown in Fig 6.

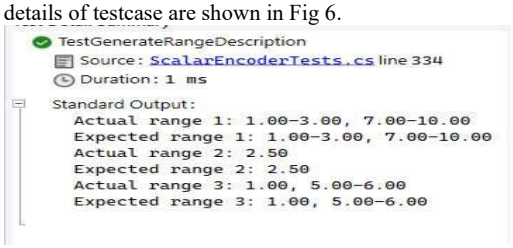

Fig6:Expected Output & Actual Output of Generate Range Description Testcase

E. Testcases- ClosenessScores:

This is a unit test method that tests the ClosenessScores method of the Scalar Encoder class.
The test arranges a new instance of Scalar Encoder with specific parameters. Then it sets up expected and actual value arrays with one value each. It then sets a Boolean flag to true to indicate that the values are fractional. The expected closeness score is set to 0.99.
Finally, the test asserts that the closeness score of the expected and actual value arrays is equal to the expected value with a tolerance of 0.01.
Params:
“expValues”: an array of expected values, set to { 50 }
“actValues”: an array of actual values, set to { 51 }
fractional: a Boolean flag indicating whether the values are fractional, set to true expected. Closeness: the expected closeness score, set to 0.99

The four test methods are testing the ClosenessScores method of the Scalar Encoder class with different input values and encoder parameters.
In ClosenessScorestest1, a periodic encoder is used with a target value of 50 and an actual value of 51. The expected closeness is 0.99, meaning that the actual value is very close to the target value.
In ClosenessScorestest2, a non-periodic encoder is used with a target value of 50 and an actual value of 50.3. The expected closeness is also 0.99 with a maximum difference of 0.01.
In ClosenessScorestest3, a periodic encoder is used with a target value of 50 and an actual value of 70. The expected closeness is 0.8, meaning that the actual value is farther from the target value compared to the previous two tests.
In ClosenessScoresTest4, a non-periodic encoder is used with a target value of 25 and an actual value of 75. The expected closeness is 0.5, meaning that the actual value is very far from the target value.
The main difference between the tests is the encoder parameters used, which affect the encoding of input values and the closeness scores calculation. The obtained details of testcase are shown in Fig 7.

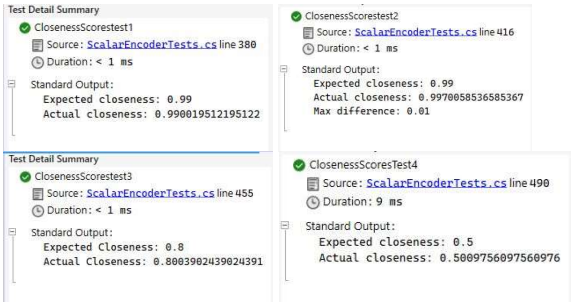

Fig 7:Expected Output &Actual output of ClosenessScores

F. Testcase-EncodeIntoArray:

This program is a unit test method for the Scalar Encoder class's EncodeIntoArray method. A collection of input values are encoded into a bitmap, and the output array is checked to make sure the input value was encoded correctly.
The following variables are used in the constructor of the Scalar Encoder:
"W": 21, the "width" of the output signal, which is the number of bits set to encode a single value.
The output's "N": 1024 bit count must be larger than or equal to W.
"Radius": 1.0, non-overlapping representations between two inputs separated by greater than the radius.
The input signal's minimal value, "MinVal," is 0.0.
The input signal's upper bound, "MaxVal ," is 100.0.
If "periodic" is true, the input value is determined.
"wraps around" such that MinVal = MaxVal . For a periodic value, the input must be strictly less than MaxVal , otherwise MaxVal is a true upper bound.
"Name": "scalar_periodic", the name of the encoder.
"ClipInput": false, if true, non-periodic inputs smaller than MinVal or greater than MaxVal will be clipped to MinVal/MaxVal ..The inputs array contains the input values to be encoded: { 1, 25, 60, 75, 99 }.
The method loops over each input value, calls EncodeIntoArray, verifies the output array, creates a bitmap of the encoded value, and saves it to disk.
The bitmap is created using the NeoCortexUtils.DrawBitmap method with the following parameters:
twoDimArray: the 2D array containing the encoded value.
1024: the width of the bitmap.
1024: the height of the bitmap.
${outFolder}\\encoded_{input}.png: the filename of the bitmap to be saved.
Color.Gray: the color of the background.
Color.Green: the color of the foreground.
text: a string containing the input value and the encoded value.
Finally, the method verifies that EncodeIntoArray returns 0 for each input value.
The obtained details of testcase are shown in Fig 8.

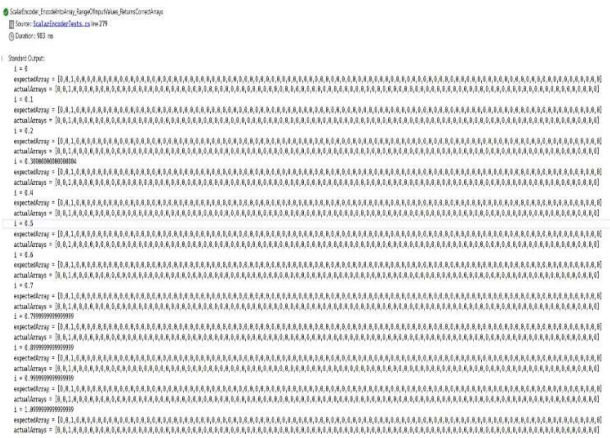

Fig8 :Expected output & Actual Output of Encode Into Array

G. Testcase-ScalarEncodingDecode:

This unit test method examines the Scalar Encoder class decode method. The method accepts some decoding-related parameters as well as encoded input values in the form of an array of integers (output1, output2, etc.).
The variables employed are:
MinVal: the input range's minimal value (0)
MaxVal: The input range's highest value (100)
The encoding's bit count is n. (14)
w: each encoding bin's width (3.0)
If the input range is periodic, a Boolean value called periodic will be present (true)
The procedure goes over the test cases and applies the decode technique to each one in turn. The console is then printed with the decoded input values for validation. The obtained details of testcase are shown in Fig 9.

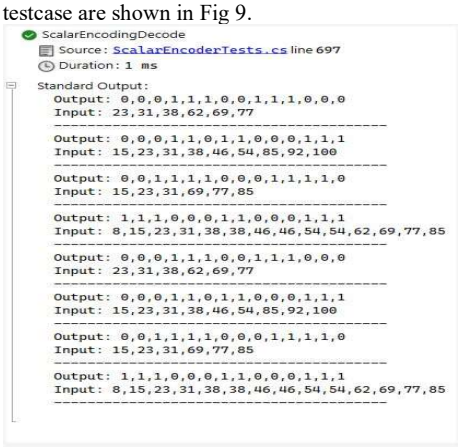

Fig 9:The Input & Output of the Decode Testcase.

H. Testcase- TestGetBucketInfo:

The code is a unit test in C# that tests the functionality of the GetBucketInfo method of the Scalar Encoder class. The Scalar Encoder is a class in the Hierarchical Temporal Memory (HTM) library that encodes scalar values into sparse distributed representations suitable for use as inputs to HTM algorithms.

The first test, TestGetBucketInfoNonPeriodic, tests the method with a non-periodic encoder. The encoder is initialized with the following parameters:
W: The number of bits in the output representation of each input value. In this case, it is set to 21.
N: The number of bits in the output representation of the entire encoder. In this case, it is set to 100.
Radius: The radius of the input space. A value of -1.0 means that the radius is automatically determined based on the range of the input space.
MinVal: The minimum value of the input space. In this case, it is set to 0.0.
MaxVal: The maximum value of the input space. In this case, it is set to 100.0.
Periodic: Whether the input space is periodic or not. In this case, it is set to false, indicating that the input space is not periodic.
Name: The name of the encoder. In this case, it is set to "scalar nonperiodic".
ClipInput: Whether to clip input values to the range of the input space or not. In this case, it is set to false, indicating that input values outside the range of the input space will be encoded.
NumBuckets: The number of buckets used to encode the input space. In this case, it is set to 100.
The test cases involve passing different input values to the GetBucketInfo method and verifying that the returned bucket information is correct. For example, the first test case passes the value 49.0 to the method and expects the bucket information to be [49, 50, 49, 50]. This means that the input value is in the bucket with index 49, its center is at bucket index 50, its start is at bucket index 49, and its end is at bucket index 50.
The second test, TestGetBucketInfoPeriodic, tests the method with a periodic encoder. The encoder is initialized with similar parameters as before, except that the Periodic parameter is set to true, indicating that the input space is periodic.

The test cases are similar to the non-periodic case, except that the periodicity of the input space affects the bucket information. For example, the first test case passes the value 49.0 to the method and expects the bucket information to be [49, 49, 49, 50]. This means that the input value is in the bucket with index 49, its center is at bucket index 49, its start is at bucket index 49, and its end is at bucket index 50.The output is shown in (Fig 10).

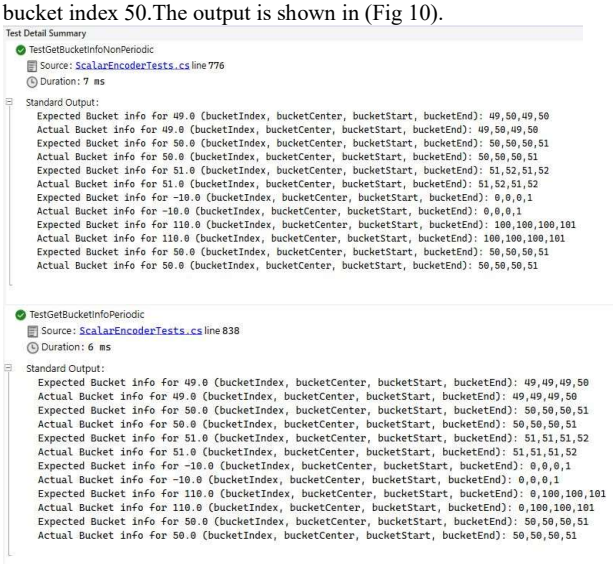

Fig10: The Actual & Expected output of GetBucketInfo Testcase with Periodic &Non Periodic.

I. Testcase-GetTopDownMapping:

This test method examines the operation of the Scalar Encoder class _getTopDownMapping function for a periodic encoder setup. It then confirms the right output of the _getTopDownMapping method by comparing the actual output with the desired output and outputting the arrays for debugging purposes after setting up a Scalar Encoder instance with particular setup parameters and a test input value.
These are the configuration parameters that were used:

W (int): The encoder's operating space's size in terms of input. The value is 21.
N (int): The encoder's output space has N bits total. The value is 100.
Radius (double): The size of the Gaussian distribution used to produce the encoder output overlap. a result of -1.00
A radius that is automatically determined depending on the size of the input space and the number of bits in the output space has a value of -1.0. It has a value of 1.0.
MinVal (double): The input space's lowest value. It has a value of 0.0.
MaxVal (double): The input space's maximum value. Its value is 1.0.
If the input space is periodic is indicated by the Boolean value periodic (bool). The value is true.
Name (string): The encoder's name. The value is "scalar nonperiodic."
A Boolean value indicating whether or not the input values should be clipped to the input space bounds is called ClipInput (bool).
The number of output bits that must be set is indicated by NumBuckets (int). The number of output bits that should be set to 1 for each input value is specified by NumBuckets (int). The setting is 4.
0.25 is the test input value.
The output array that is anticipated is 0 1, 0 0.
Using the _getTopDownMapping method of the encoder with the input value, periodicity flag, and number of buckets as parameters yields the actual output array.
For debugging reasons, the Console. WriteLine function prints the actual and anticipated output arrays.
Lastly, the test method uses the CollectionAssert.AreEqual method to confirm that the actual output array matches the anticipated output array. The output is shown in (Fig 11).

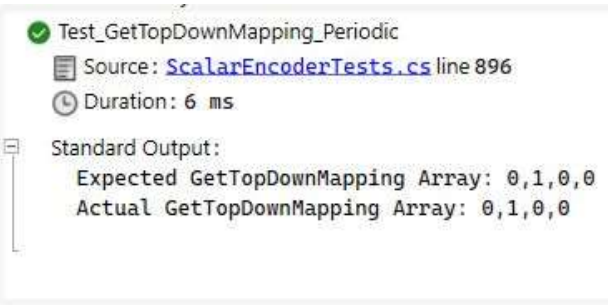

Fig11: Expected Output & Actual Output of the GetTopDownMapping

VI. Applications of Scalar Encoder with Buckets

A scalar encoder with buckets is often employed in applications that require continuous data to be encoded into sparse dispersed representations. Here are a couple such examples:

Time Series Data: A scalar encoder with buckets can be used to encode time series data, which transforms continuous values of time into binary representation. This model can then be applied to tasks such as anomaly detection, prediction, and classification.

Sensor Data: Sensor data, such as temperature, humidity, and pressure, can be encoded using a scalar encoder with buckets. Data that has been encoded can subsequently be used in a variety of applications such as environmental monitoring, home automation, and industrial control.

Speech Recognition: A scalar encoder with buckets can be used to encode speech signals, in which the continuous values of an audio stream are turned into a binary value. This representation can then be utilized to recognize spoken words in speech recognition tasks.

Image Processing: A scalar encoder with buckets can be used to encode image data, which transforms an image's continuous pixel values into a binary representation. This representation can then be applied to a variety of image processing applications, such as object detection, image segmentation, and image classification.

Overall, the scalar encoder with buckets is a versatile technique that may be used in a variety of applications requiring the encoding of continuous data into sparse distributed representations.

VII. Conclusion

In conclusion, the Scalar Encoder with Buckets is an improved version of the Scalar Encoder that overcomes some of its limitations. By mapping input values to continuous ranges of buckets, the Scalar Encoder with Buckets provides better precision in encoding data compared to the Scalar Encoder. The automatic setting of parameters such as the number of buckets and bucket size based on the input data, as well as the generation of a bucket range description, increases the flexibility of the Scalar Encoder with Buckets and makes it easier to adapt to different datasets. Furthermore, by supporting periodic encoding of values, the Scalar Encoder with Buckets can handle cyclical data more effectively.

The methods used in the Scalar Encoder with Buckets, such as ClosenessScores, EncodeIntoArray, decode, GetBucketIndex, GetTopDownMapping, GenerateRangeDescription, and GetBucketValues, all contribute to the improved performance of the encoder. The use of continuous ranges of buckets and the improved encoding scheme enable better precision in encoding data, while the automatic setting of parameters and the bucket range description increase the flexibility of the encoder. The support for periodic encoding of values provides a more effective way of handling cyclical data. Overall, the Scalar Encoder with Buckets is a powerful tool for encoding data and provides a more flexible and precise alternative to the Scalar Encoder.

Overall, the Scalar Encoder with Buckets encodes continuous data in a more flexible and accurate manner, making it a superior choice for many real-world applications.

VIII. References

[1] S. &. H. .. Ahmad, " "Properties of sparse distributed representations and their application to hierarchical temporal memory.,"," 2011. [Online]. Available: doi: 10.1371/journal.pone.0022149.

[2] S. A. J. Hawkins, ""Why neurons have thousands of synapses, a theory of sequence memory in neocortex,"," 2016. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fncir.2016.00023/full.

[3] S. Purdy, "Encoding Data for HTM Systems," [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1602/1602.05925.pdf.

[4] "Scalar Encoders," [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1602/1602.05925.pdf.

[5] S. P. S. Ahmad, "Spatial Pooling with Hierarchical Temporal Memory," [Online]. Available: https://arxiv.org/abs/1705.05363.