Scalar Encoder with Buckets

Sahith Kumar Singari  
[sahith.singari@stud.fra-uas.de](mailto:sahith.singari@stud.fra-uas.de)

Anil Kumar Gadiraju  
[anil.gadiraju@stud.fra-uas.de](mailto:anil.gadiraju@stud.fra-uas.d)

Vinay Kumar Bandaru

[vinay.bandaru@stud.fra-uas.de](mailto:vinay.bandaru@stud.fra-uas.de)

*Abstract*— In real-life situations, Scalar encoders with buckets are useful in a variety of real-world scenarios where continuous quantities must be represented as discrete values. This is frequently the case in machine learning and data analysis, when you may want 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 with buckets. Encoders convert various data types into sparse distributed representations. They accept external inputs and convert them into a binary representation known by the CLA, similar to how the retina or cochlea convert external information into binary neuronal representations. This paper evaluates Investigate the implementation and performance of scalar encoding using buckets in AI and machine learning applications. We will examine the literature on best practices for determining the number and breadth of buckets, compare scalar encoding using buckets to other encoding approaches.

*Keywords- Hierarchical Temporal Memory (HTM), Sparse Distributed representations (SDRs), neocortex, numeric, array*

# **Introduction**

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.

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. These ML models do not match the real-world issues

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 [4] [5]. HTM has given promising results in pattern recognition, and This can learn the temporal sequences and spatial flow of sensory inputs as data.

# **LITERATURE** **SURVEY**

## **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.

SDR can be used in a wide range of applications using HTM systems. Consequently, the first step of using an HTM system is to convert a data source into an SDR using what we call an encoder. The encoder converts the native format of the data into an SDR that can be fed into an HTM system. The encoder is responsible for determining which output bits should be ones, and which should be zeros, for a given input value in such a way as to capture the important semantic characteristics of the data. Similar input values should produce highly overlapping SDRs.

Diagram

Description automatically generated

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

## **Neocortex**

The neocortex is the region of the cerebral cortex that is responsible for mental functioning in humans. There are also billions of cells and millions of meters. The cells are tiered, with distinct areas dedicated to vision, hearing, touch, movement, sensory balance, stimulation, and so on.

*Figure.2: Neocortex Layers* [6]

HTM is a working model that was created with the goal of simulating the biological neocortex's functions. Its role is to process the sensory input provided as input data. HTM replicates the neuron model using a variety of techniques until the functionality framework is established to receive the appropriate sensory data [7]. The research has also demonstrated that biological neurons carry out more intricate tasks.

## **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.

## **Hierarchical Temporal Memory (HTM)**

The HTM model learns the procedure that occurs in one layer 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 recurrent pattern.

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.

# **Methodology**

The project Scalar Encoder with Buckets 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

## **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. 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. To make sure that every column takes part in the training, the inactive columns are boosted. 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. To make sure that every column takes part in the training, the inactive columns are boosted. 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.

*Figure. 3: HTM Algorithm Flow*

## **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.

# **implementation**

To build an effective encoder that can result in similarity, you must comprehend the components of your data.

In the aforementioned 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 in 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.

## **Scalar Encoder 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. :**param maxval:** The maximum value of the input signal. (If "periodic Equals= True," input is strictly smaller) :**param 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.

**InitEncoder:** (Assistant function) There are three ways to think about the representation.

**GetFirstOnBit:** The bit offset of the first bit to be set in the encoder output is returned.

As the encoded output wraps around in periodic encoders, this can be a negative value.

**getTopDownMapping**: The internal \_topDownMappingM matrix needed to handle the **bucketInfo** and **topDownCompute** functions is returned. This is a matrix with one row for each category (bucket), with each row containing the encoded output for that category.

**GenerateRangeDescription**: create a description from a range's text description.

**GetBucketIndex**: 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.

**SetBucketValue:** Set the value of the bucket at the given index to the given value.

## **EncoderBase:**

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**: encodes input Data and writes the encoded value to the 1-D array of length .

param output: 1-D array of the same length .

getEncodedValues: 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 (timeOfDay, dayOfWeek, etc.)

The main difference between this method and ".get Scalars" is that it returns strings instead of scalars. :param input Data: The input data in the format it was received from the data source.

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.

inputData: The input data that needs to be encoded.

return: an array containing the inputData's encoded form.

**ClosenessScores:** Calculate ratings of proximity between the expected and actual scalar values (s). Typically, the real scalar values are those returned by the method "topDownCompute".

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.

## **Testcases Methods of Scalar Encoder**

**ScalarEncoderTests:** This is a unit test for the ScalarEncoder class' Encode function. It makes clear that this is a unit test that the testing framework may run by using the [TestMethod] attribute. When performing smaller sets of tests, it can be useful to categorize the test using the [TestCategory("UnitTest")] attribute.

Test data for the test method is provided by the [DataRow] attribute. The test data in this instance consists of a double value and an array of integers that represents the predicted outcome. The width, MinVal, MaxVal, and radius parameters should be used by the ScalarEncoder class to encrypt the double value, and it should then return an array of integers that contains the resulting value. The test then compares the outcome to the anticipated outcome to see if the test has passed or failed.

**ScalarEncoderUnitTestNonPeriodic:** A scalar input value and an array of integer values (the sparsedistributed representation) are the arguments for the "ScalarEncoderUnitTestNonPeriodic" function. A new instance of the "CortexNetworkContext" class is then created, and various encoder options are then set using a dictionary. A new instance of the "ScalarEncoder" class is then created and used to encode the input value. With the "Assert.IsTrue" method, the obtained sparse distributed representation is contrasted with the anticipated outcome.

An example of a data row used for parameterized testing is presented in the second technique. The encoder is tested using a scalar value and the anticipated output array as arguments. When the width is set to 11, the minimal value, it appears to test the behavior of the encoder, the minimum value is 1, the maximum value is 100, and the resolution is 0.15.

**SeasonEncoderTest:** A binary vector is created by the encoder from a scalar value that represents the day of the year. The test determines whether the encoder outputs the anticipated binary vector given a specific input value.

An expected binary vector and an input value of 1.0 (indicating January 1) are specified in the test's single data row. The length of the anticipated binary vector, which is equal to the number of bits utilized to encrypt the value, is 12. The value is on the edge of the range of values that the encoder can encode because the first and last bits are both set to 1, which indicates this. The value falls into the first group of encoded values because the second bit is also set to 1, signalling this. the rest of set to 0, indicating that the value is not in any of the other categories.

The test uses the Assert.IsTrue method to check if the expected binary vector is equal to the vector returned by the encoder for the given input value. If the vectors are not equal, the test will fail.

**TimeTickEncodingTest:** scalar encoder is used. The test is configured to encode a collection of numbers and generate a bitmap output, and it currently seems to be passing.

The code creates a scalar encoder that can encode values in the range of 0 to the number of days in the current year by using a DateTime object to obtain the current time. After encoding each value in the range using a loop, it ensures that the output has the right number of bits set to 1.

The aim of this test is not quite evident from the code, but it appears to be confirming that the scalar encoder can accurately encode a variety of time values.

**ScalarEncodingGetBucketIndexNonPeriodic:** 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.

The test appears to cover proper GetBucketIndex method operation as well as visual verification of the encoded bit pattern for non-periodic scalar encoder settings.

**ScalarEncodingGetBucketIndexPeriodic:** These methods appear to be testing the ScalarEncoder 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.

**TestGenerateRangeDescription:** The ScalarEncoder 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 ScalarEncoder 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 ScalarEncoder class ClosenessScores method is being tested in this test. The ScalarEncoder 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 ScalarEncoder is used by the ClosenessScores function to determine the closeness score between the expected and actual values.

**InitTest:** The setting-based initialization of the ScalarEncoderExperimental object.

By invoking the getDefaultSettings() method, it first sets the default encoder settings. The "Name" property is then changed to "hello" and a new property called "TestProp1" with the value "hello" is added.

Then, it calls the Initialize method on a newly created ScalarEncoderExperimental object to initialize it with the adjusted encoder settings. Also, it uses the indexer property to set a new property "abc" with the value "1".

Finally, it claims that the encoder object's "TestProp1," "Name," and "abc" attributes are set to the required values.

Overall, the ScalarEncoderExperimental object can be initialized appropriately with the given values, and properties may be set and accessed correctly, according to this test method.

**InitializeAllEncodersTest:** It initializes every encoder listed in the CortexNetworkContext object and verifies that the initialized encoder's properties match the encoderSettings dictionary's default values.

The CortexNetworkContext instance is first created, and the method confirms that it has at least one encoder and is not null. It then generates a fresh instance of the ScalarEncoderExperimental class using the encoderSettings dictionary for each encoder in the context. It then repeatedly checks that each setting's value corresponds to the appropriate value of the initialized encoder by iterating over the encoderSettings dictionary.

Overall, this function verifies that all encoders are appropriately initialized with the specified settings by the CortexNetworkContext's CreateEncoder method.

static Dictionary: A dictionary containing the default settings for a scalar encoder is the result of the getDefaultSettings function. These conditions consist of:

W: The "width" of the output signal, or the number of bits that are set to represent a single value. The default value for this integer is 11. Keep in mind that W must be odd to prevent centering issues.

N: how many bits are in the output. The default value for this integer is 0. N will be determined as the smallest power of 2 bigger than or equal to W if it is not specified.

MinVal: the input signal's lowest value. The default value for this double value is 1.

MaxVal: the input signal's maximum value. This value has two parts.

MaxVal: the input signal's maximum value. The default value for this double number is 100.

Radius: Non-overlapping representations exist when two inputs are separated by greater than the radius. In general, there will be at least some bit overlap between two inputs that are closer together than the radius. The default value for this double value is 0.

Resolution: It is assured that two inputs that are separated by a distance larger than or equal to the resolution will have different representations. The default value for this double number is 0.15.

Periodic: If this condition is true, the input value "wraps around" so that MinVal = MaxVal. If the input for a periodic value is not strictly less than MaxVal, MaxVal becomes a valid upper bound. A boolean value, true is the default for this one.

Non-periodic inputs that are smaller than MinVal or greater than MaxVal will be trimmed to MinVal/MaxVal if ClipInput is true. A boolean value, true is the default for this one.Name: a string that identifies the encoder. It is "TestScalarEncoder" by default.

The boolean flag IsRealCortexModel indicates whether or not the encoder is a component of a real cortex model. True is the default.

**ScalarEncodingEncodeIntoArray:** This test method examines the ScalarEncoder 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 ScalarEncoder 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.

**ScalarEncodingDecode:** This unit test is for the ScalarEncoder 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 ScalarEncoder 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 gray (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 ScalarEncoder class's EncodeIntoArray function returns the right arrays for a variety of input values.

# **TestCase with RESULTS**

In this project,

# **Conclusion**

Multi Sequence learning for Sequence of Numbers which uses Neocortex API is used as a reference model to develop a solution for Multi Sequence learning - Sequence of Alphabets and Multi Sequence learning- Image data sets. HTM Prediction Engine was modified with different parameters to match the respective training process. The Sequence of Alphabets (Anticancer Peptide Sequence) Stored as a CSV file was modified and stored as an encoded value in the dictionary using Scalar Encoder and SDR input for the Training process. A prediction algorithm was developed to predict the trained sequences where the similarity matrix generated is compared with each of the SDRs of the Sequence learned during the training phase and based on the accuracy and observation class (Label), the Sequence is predicted.

HTM Image Encoder was incorporated to develop a solution that could train multiple Image data sets and a prediction algorithm that could predict input images. The HTM Image Encoder binarizes the input image and stores as array elements of zeros and ones used as SDR Input for training. Similar to the Prediction algorithm for Sequence of Numbers and Alphabets, the Prediction of Image algorithm was developed, and the input image was predicted by comparing with the trained data sets and returning the prediction output based on accuracy and Observation class (Label).

We performed Multi Sequence Learning for a different sequence of data sets and could achieve up to 87.5% of accuracy in the Training Phase.

The experiments carried out helped us understand different types of encoders, such as scalar encoders and HTM Image encoders, how the Spatial pooler creates SDR inputs and computes the learning phase, and how the Homeostatic Plasticity controller helps in stabilizing the learning phase in NeoCortex API.

# **References**

|  |  |
| --- | --- |
| [1] | Keele, "“Sequence learning,” - B. A. C. G. J. D. S. W.," 1998. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/21227209.. |
| [2] | "D. V. B. Michael D Mauk, “Sequence learning,”," 2004. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/15217335. |
| [3] | "B. H. J. L. R. .Rabiner, “An introduction to hidden markov models,”," 1986. [Online]. Available: http://ai.stanford.edu/~pabbeel/depth\_qual/Rabiner\_Juang\_hmms.pdf. |
| [4] | Hawkins, "J. C. S. A. J. , “Continuous online sequence learning with an unsupervised neural network model,”," 2016. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/27626963. |
| [5] | 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. |
| [6] | "Neocortexlayersimageref," [Online]. Available: http://brainmind.com/neocortex.html. |
| [7] | K. J. Hole, "The htm learning algorithm," 2016. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-30070-2\_11 . |
| [8] | 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 . |
| [9] | M. L. Repository, "AntiCancer Peptides Data sets," 2019. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Anticancer+peptides.. |
| [10] | B. N. a. I. N. S. G. Eason, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," London, Phil. Trans. Roy. Soc, pp. vol. A247, pp. 529–551. |
| [11] | J. C. Maxwell, "A Treatise on Electricity and Magnetism," Oxford: Clarendon, 1892, pp. 3rd ed., vol. 2. pp.68–73. |
| [12] | I. S. J. a. C. P. Bean, "Fine particles, thin films and exchange anisotropy in Magnetism," G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. vol. III, 271–350. |