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

UnitTests: HomeostaticPlasticityController

Shubham Pramod Suryawanshi
shubham.suryawanshi@stud.fra-uas.de

Abstract—Homeostatic Plasticity Controller (HPC) plays a vital role as an extension of Hierarchical Temporal Memory (HTM) – Spatial Pooler (SP) learning algorithm. HPC and SP have been modelled and implemented as part of the .NET Core Library: NeoCortex API [1]. The HPC class is designed to work in tandem with the SP algorithm. By controlling the boosting and inhibition mechanism during Spare Distributed Representation (SDR) generation process, the HPC is fundamentally important to the HTM SP algorithm. To make sure methods in the modelled HPC class are operating accurately, rigorous Unit Tests have been designed and implemented in the project UnitTestsProject of the NeoCortex API.

Keywords—Homeostatic Plasticity Controller (HPC), Hierarchical Temporal Memory (HTM), Sparse Distributed Representation (SDR), Spatial Pooler (SP), Unit Testing, Code Coverage Metric.

I. Introduction

Inspired by the natural functioning of human brain, the Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) uses spatial pattern recognition and temporal sequence learning (Hawkins, Subutai and Cui, 2017) for intelligent operations such as classification.

The brain tissue if spread out like a cloth consists of many column-like structures which embody many neurons, such columns contain many mini-columns which can be activated and deactivated as whole. The mini-columns consist of cells that can be singularly activated if required. This model of human brain can be implemented and used to perform many intelligent tasks. To train such a model HTM CLA can be utilized, HTM CLA consist of two algorithms:

Spatial Pooler and Temporal Memory.

The Spatial Pooler operates on mini-columns as explained in the previous section connected to sensory inputs (Yuwei, Subutai and Hawkins, 2017) . It is responsible to learn spatial patterns by encoding the pattern into the sparse distributed representation (SDR). The Spatial Pooler algorithm implements a boosting of columns inspired by homeostatic plasticity mechanism. Unlike neurons in a deep neural network, in NeoCortex the cells have 3 stable states: Active, Inactive and Predictive. HTM SP algorithm and Temporal Memory sets cells to inactive or predictive state. SP provides two basic configuration algorithms: Global or Local Inhibition. HPC takes boosting and inhibition one step further, increasing the efficiency on a cellular level in NeoCortex.

II. HPC Workflow

Homeostatic Plasticity Controller controls the boosting and inhibition of Mini-Columns in the NeoCortex. In the following diagram, the role of HPC is clearly demonstrated. During the process of Sparse Distributed Representation generation, HTM SP algorithm along HPC helps in increasing efficiency as it encodes information in form of SDRs. This intelligent use of memory helps in efficiently using the available computing power.

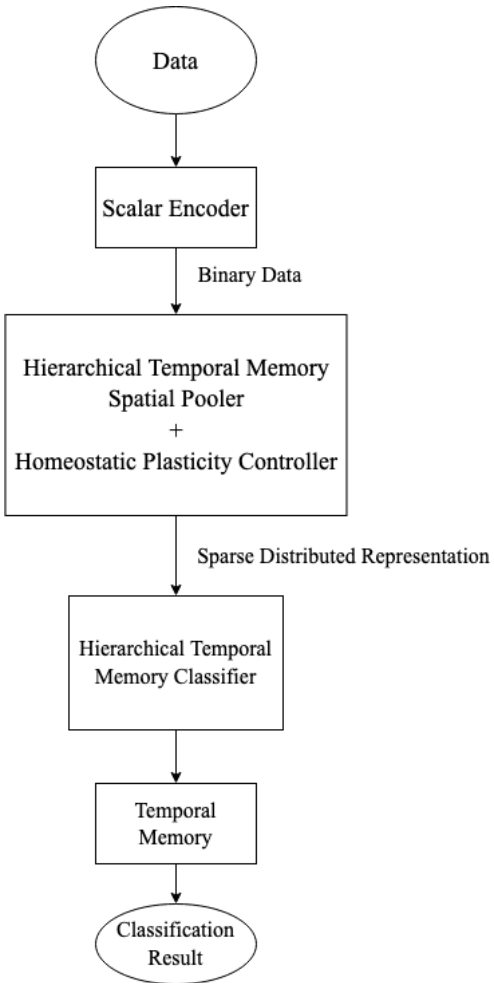

Figure 1: Flowchart describing a general pipeline for data during classification using HTM SP Algorithm. Source [2]

III. Unit Test Design

Unit Testing is a common practice in modern software development cycle. There are numerous benefits to implement Unit Tests in a project; they facilitate good design and provide a concrete frame to ensure solid system design. A well-designed Unit Test assists in debugging errors and could possibly help prevent an error to development stage.

Modularity is an important principle in modern Software Development, tightly coupled coded is a bad practice as it can be difficult to unit test. Writing Unit Tests highlights the coupling which may be less apparent. Thus, the practice of implementing Unit Tests naturally decouples code and provides modularity.

The quality of Unit Tests dictates its significance in the software project. It is not an uncommon sight for mature projects to have thousands of Unit Tests. A good Unit Test should consume minimal resource and run as fast as possible. Each Unit Test should be designed to be a standalone test which can be run in isolation, dependencies on file systems, databases, etc. will make the test vulnerable. Unit Tests are designed to be repeatable, thus producing a consistent result and highlighting the discrepancy in the code base with precision. These properties of a Unit Test should be kept in mind when engineering a Unit Test.

Dependencies on infrastructure when writing unit tests should be avoided. These would make the tests slow and brittle, to avoid this it should be ensured that unit test project does not have any references to or dependencies on infrastructure packages. Integration Tests should be used for Dependency testing. Use of Explicit Dependencies Principle and Dependency Injection can be used in Unit Tests to avoid direct dependency. You can also keep your unit tests in a separate project from your integration tests.

Mature projects have complicated Classes and their objects with a high number of properties associated with them. While designing Unit Tests the parameters to be tested should be in-place and other parameters can be mocked. A Mock object or a Stub object can be used to achieve this. A mock starts out as an empty object until it's asserted against a specific condition. After assertion a mock object can manifest itself to check whether a unit test has passed or failed. A stub is a controllable replacement for an existing dependency (or collaborator) in the system. By using a stub, you can test your code without dealing with the dependency directly. By default, a stub starts out as an empty object.

Apart from testing code Unit Tests also play a fundamental role in documentation for the project. A high-quality Unit Test exposes the inner operations of the function being tested and thus helps the user understand the mechanics of the function. Naming standards of Unit Tests are important because they explicitly express the intent of the test. Just by looking at the suite of unit tests, user should be able to infer the behavior of your code. Moreover, when a Unit Test fails, user can see explicitly which scenario does not meet the expectations.

Each Unit Test should strive to test just one feature of the method being tested. By avoiding multiple acts with single Unit Test, in case of Unit Test failure, the user can swiftly point to the code structure that is failing. Thus, improving Unit Test quality and minimizing efforts for bug-fixes.

For a class as important and fundamental as HPC, rigorous Unit Test should be maintained in-place to validate upcoming and existing features. The Unit Tests for HPC class are placed in a TestClass inside UnitTestsProject: HomeostaticPlasticityControllerTests. A total of 14 Unit Tests are designed to meticulously test the operations of each method of HPC class.

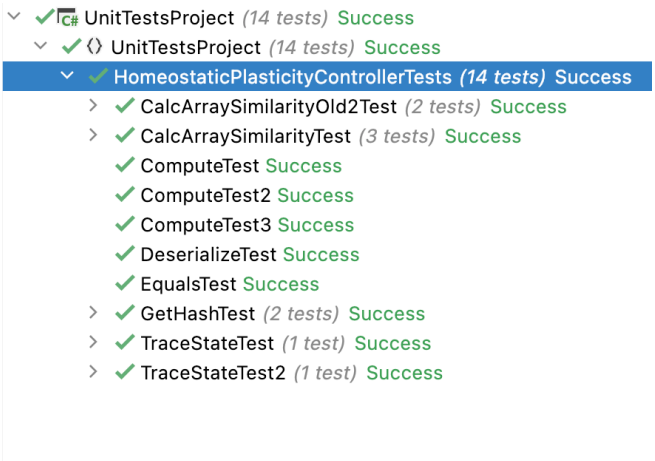

Figure 2: Run report for the Unit Tests designed for HomeostaticPlasticityController class

In certain Unit Tests where the arguments are defined at compile-time, an attribute called DataRow can be used to run multiple test cases with variety of data all of which test the same underlying code structure under different scenarios. This feature allows more rigorous testing without compromising readability of Unit Tests. Using this attribute, the same Unit Test is executed multiple times, each times using different test parameters.

```
[TestMethod]
[Description("Check CalcArraySimilarity method in different scenarios")]
[DataRow(-1.0, new int[] {}, new int[] {})]
[DataRow(0.6, new int[] {1,2,3,4,5}, new int[] {3,4,5,6,7})]
[DataRow(0.0, new int[] {1,2,3}, new int[] {4,5,6})]
```

Figure 3: Using the DataRow attribute to run multiple tests

Each of the following sections describe in greater detail the design process and execution cycle involved in each Unit Test designed for HPC class, these Unit Tests are placed in the HomeostaticPlasticityControllerTests Unit Test Class of the NeoCortex API.

1. CalcArraySimilarityTest

The method CalcArraySimilarity is designed to compare two arrays and calculate similarity among those arrays on a scale of 0.0 to 1.0, in case the arrays are identical a return value of 1.0 is expected.

The Unit Test designed to test this method has 3 DataRow attributes, each one tests a unique case. In the first case a Null array is passed to the method and the check implemented in the method to directly return -1 value in case of null arguments is tested. In second and third case, two arrays of varying similarity are tested.

2. GetHashTest

GetHash method is designed to take in an array of Integer values, iterate through the values and converting each value into array of Bytes. Further this new array is fed to cryptographic hash function SHA256 to get the encoded version of the array. The GetHashTest TestMethod tests this method for 2 different arrays of varying length.

3. TraceStateTest

TraceState method of HPC class can be utilized to keep track of the no. of stable cycles achieved during the training phase.

```
[0 - stable cycles: 29,len = 0]
MinKey=7G[illegible], min stable states=29
```

Figure 4: A sample output expected from TraceState method

Unit Tests for this method validate the behaviour of the method TraceState under different circumstances. Both cases are tested, once when stability is achieved and once when it's not achieved.

4. CalcArraySimilarityOld2Test

CalcArraySimilarityOld2 is designed along the similar lines as the methods CalcArraySimilarityTest. As the name suggests this method is deprecated and soon to be discontinued. Nonetheless while is still present two Unit Tests have been designed for this method. These tests compare arrays with differing similarities.

5. DeserializeTest

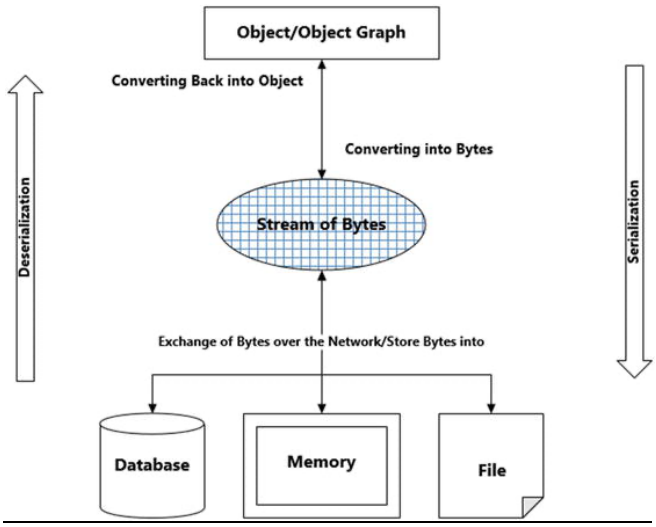

Figure 5: General operations occurring during Serialization and Deserialization process, Source [3]

Performing complex workflows, it is customary to store the results or data objects that are used in the workflow. So that user can carry on with the job and save the progress made so far. Other intention to Serialize a complex object would be to share it with fellow users and work on it as a group.

To store and share these data objects, user can serialize them share the serialized version among other users who can then deserialize the objects and then use it in other program or other workflows.

The Serialization and Deserialization methods in the HPC class use the industry standard classes from the Namespace System.IO, these are built for this specialized function: StreamWriter and StreamReader. HPC class provides both the functions, Serialization and Deserialization which wrap the mentioned System.IO Classes with additional functionality augmented specially for HPC objects. Using these functions, user can Serialize and Deserialize the HPC objects with ease.

DeserializeTest is a Unit Test design to ensure there is no data lost between the Serialization and Deserialization processes. The operation of this Unit Test involves creating a HPC object using a Connections object and then proceeding to serialize the object using the Serialization method provided in the HPC class. To ensure that no information is lost during this process, the serialized object is then deserialized using the Deserialization method provided in the HPC class and then the serialized and deserialized objects are then compared for equality. If the objects are identical, it is ensured that no information loss has occurred during the serialization and deserialization of HPC objects.

6. ComputeTest

Compute method of HPC class is the most significant method of the class. This method is called when compute method of SpatialPooler class is run. This method returns a Boolean value indicating if stable state is achieved in that particular compute cycle. There are 3 different Unit Tests designed for this method, each one of which tests different parameters in variety of conditions. The first test case is where a stable state is never achieved. In the second and third cases a stable state is achieved at least once. Functions of the internal private variables of the HPC class are validated in the third Unit Test for the method.

7. EqualsTest

Equals methods in the HPC class was designed to compare two HPC objects and return a Boolean value indicating if the two object were identical. This method is now deprecated and soon to be discontinued. A new method IsEqual is to be used to compare HPC objects for equality. As Equals method is still present in the codebase, there is a chance of it being used, hence it must be thoroughly tested. A total of 6 assertions are tested in the TestMethod EqualsTest. Each assertion tests a different scenario, covering every possible case inside the Equals method. Thus, each decision branch inside Equals method is executed at least once when Unit Test EqualsTest is run.

IV. Conclusion

As part of this project, Unit Test for HPC class of NeoCortex API were designed and place inside a new Unit Test class: HomeostaticPlasticityControllerTests. All of the tests that have been implemented successfully passed, indicating all the methods of the HPC class are performing as expected.

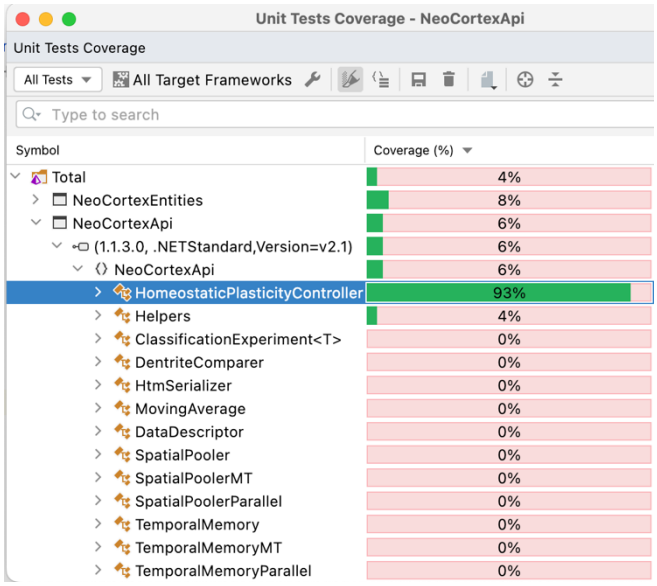

Figure 6: Code coverage metric for Unit Tests for HPC Class

Unit tests help to ensure functionality and provide a means of verification for refactoring efforts. Code coverage is an important metric that measures the amount of code that is executed when by Unit Tests are run [4]. As an example, if you have a simple application with only three conditional branches of code (branch I, II, III), a unit test that executes conditional branch I will report code coverage of 33%.

Code coverage allows the developer to judge the amount of code that has been tested and thus it guides how the regression can be further improved. As can observed from Figure 6, an impressive code coverage of 93% ensures that majority of the code base in the HPC class has been executed and validated at-least once when the tests inside Unit Test Class HomeostaticPlasticityControllerTests is run.

During the development of this project, an IDE named Rider by JetBrains has been used due to its high cross-platform compatibility and additional features which also include a tool to intuitively calculate code coverage without installing any additional third-party plugins. While using Microsoft Visual Studio, there are is a wide variety of tools available in .NET environment to calculate the metric code coverage, most popular among them are two open-source projects: Coverlet and ReportGenerator.

V. References

[1] D. Dobric, "Github," [Online]. Available: https://github.com/ddobric/neocortexapi-classification.

[2] A Neocortical Algorithm for Online Sparse Distributed Coding https://www.frontiersin.org/articles/10.3389/fncom.2017.00111/full

[3] The C# Programmer's Study Guide, Chapter: Serialization and Deserialization: https://link.springer.com/chapter/10.1007/978-1-4842-2860-9_11

[4] Unit testing best practices with .NET Core https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices