Setting Up & Using the NACS

Table of Contents

A Brief Note

Before you get started using the NACS you should note that the Clarion Library provides two methods for interacting with the NACS. The simpler method (i.e., as a stand-alone mechanism) is outlined herein. The other method (i.e., integrated with and initiated by the ACS and/or MCS[1]) can be found in the "*Advanced ACS Setup*" tutorial.[2] However, keep in mind that you should still read this tutorial first before attempting to use the NACS via the integrated method. At the very least, this tutorial will teach you how to initialize the top and bottom levels of the NACS. Furthermore, you may find that the stand-alone method is very useful for testing whether the NACS is operating correctly before moving onto the somewhat more complicated matter of integrating the NACS with the other subsystems.

Setting Up & Performing Reasoning

In this section we will go over an example of how to set up and run a task using the NACS's reasoning mechanism. If you are interested in following along, the specific example through which we will be walking is called *"Reasoner – Simple.cs"* and it can be found in the *Advanced* section of the *Samples* folder.

The "simple reasoner" simulation sample was designed with the same objective in mind as the "simple hello world" task. That is, its primary purpose is to provide a

[1] As is specified by the Clarion theory

[2] In the *Advanced* section of the *Tutorials* folder

simple introduction to the NACS. The specifics of the task, themselves, are not particularly interesting, nor were they intended to be. Instead, this task is simply meant to clearly demonstrate how to correctly setup, train, and use the various aspects of the NACS's reasoning mechanism. So let's begin our walk-through:

A Walk-through of the "Simple Reasoner" Task

The first thing you need to know are the necessary namespaces. As is normally the case, the primary classes you will use are located in either the `Clarion` or `Clarion.Framework` namespaces:

```
using Clarion;
using Clarion.Framework;
```

With this point out of the way, let's move on to the `Main` method:

```
public static void Main()
{
    Agent reasoner = World.NewAgent();

    InitializeWorld(reasoner);

    foreach (DeclarativeChunk dc in chunks)
        reasoner.AddKnowledge(dc);

    HopfieldNetwork net = AgentInitializer.InitializeAssociativeMemoryNetwork
        (reasoner, HopfieldNetwork.Factory);

    net.Nodes.AddRange(dvs);

    reasoner.Commit(net);

    EncodeHopfieldNetwork(net);

    SetupRules(reasoner);

    reasoner.NACS.Parameters.REASONING_ITERATION_COUNT = 2;
    reasoner.NACS.Parameters.CONCLUSION_THRESHOLD = 1;

    DoReasoning(reasoner);

    reasoner.Die();

    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}
```

Most of the interesting details of this task are actually in other methods that are called by the `Main` method. However, you may notice a few unfamiliar things in the above code. First, note the following line:

```
InitializeWorld();
```

For our "simple reasoner" task, we begin by initializing the `World` with 30 dimension-value pairs and 5 unique declarative "pattern" chunks. These chunks are manually specified by the following:

```
static int [][] patterns =
{
    new int [] {1, 3, 5, 11, 13, 16, 19, 23, 27},
    new int [] {3, 6, 7, 8, 12, 15, 20, 21, 26},
    new int [] {2, 4, 8, 9, 11, 17, 18, 24, 30},
    new int [] {1, 4, 10, 12, 15, 17, 19, 22, 29},
    new int [] {3, 5, 8, 10, 14, 18, 20, 25, 28}
};
```

Each of the sub arrays (located in the 2nd dimension of the above 2-dimensional array) specifies a different activation pattern for the 30 dimension-value pairs. The "value" of each dimension-value pair is actually numbered, and the integers in the above patterns are associated with these values. As mentioned previously, we will also need to create a `DeclarativeChunk` for each of these patterns. The following `World` initialization code demonstrates how we can accomplish this:

```
static void InitializeWorld(Agent a)
{
    for (int i = 1; i <= nodeCount; i++)
    {
        dvs.Add(World.NewDistributedDimensionValuePair(a, i));
    }

    for (int i = 0; i < patterns.Length; i++)
    {
        DeclarativeChunk dc =
            World.NewDeclarativeChunk(i, addSemanticLabel:false);

        foreach (var dv in dvs)
        {
            if (patterns[i].Contains(dv.Value))
            {
                dc.Add(dv);
            }
        }

        chunks.Add(dc);
    }
}
```

Note that the "`dvs`" and "`chunks`" collections in the above code are simply used to track our dimension-value pairs and declarative chunks between the different "phases" (or methods) of the task. As has been discussed in previous tutorials, this is generally a useful thing to do within any simulating environment, as it saves the additional overhead of using the "`World.Get...`" methods.

Distributed Dimension-Value Pairs

In the previous code sample, you may have noticed that we called the `NewDistributedDimensionValuePair` method instead

NewDimensionValuePair. This has been done in order to introduce you to a new (as of 6.1.1) feature of the Clarion library: DistributedDimensionValuePair. The concept behind distributed dimension-value pairs was first introduced by Helie & Sun (2010). The idea here is that more semantic representations (such as chunks) can be translated into more "distributed" (i.e., neural-like) representations in the bottom level of Clarion. In other words, instead of having the nodes of a network be tied to dimension-value pair that has inherent semantic meaning, collections of somewhat arbitrarily defined nodes could instead be utilized to represent semantic concepts more in a more sub-symbolic way (see Helie & Sun, 2010 for more details). As a matter of implementation, this idea has been actualized in the form of DistributedDimensionValuePair.

Distributed dimension-value pairs are agent-specific and do not require a dimension be specified (just a value). Otherwise, once initialized, they can be utilized essentially like a normal dimension-value pair. When added to a chunk (and as nodes in an ImplicitComponent), these distributed dimension-value pairs provide a "sub-symbolic" featurized representation for the chunk itself.

Adding Knowledge to the GKS

Moving back to our discussion of the Main method, the next thing you may have noticed is the call to the AddKnowledge method (located in the Agent class):

```
foreach (DeclarativeChunk dc in chunks)
    reasoner.AddKnowledge(dc);
```

This method is used to add the declarative chunks[3] into the GKS of our agent. Be aware that **ALL** chunks **MUST** be added to the GKS if they are to be used as part of reasoning. Besides the obvious theoretical consideration, we also need to do this for implementation-specific purposes. In particular, various aspects of the GKS's backend are actually used to help facilitate the reasoning process.

You should also note here that chunks should **NEVER** be altered (say, by adding or removing a dimension-value pair) after they have been added to the GKS. Doing so will break the storage method that is used to store these chunks within the GKS. To relate this to a well-known concept from object-oriented programming, altering a chunk once it is in the GKS is essentially the same as changing the hash code of an object after it has been stored within a HashMap. In order words, **DON'T DO IT**!

Initializing Associative Memory Networks

Moving along with our walk through of the Main method, the next thing to notice is the following:

```
HopfieldNetwork net = AgentInitializer.InitializeAssociativeMemoryNetwork
    (reasoner, HopfieldNetwork.Factory);

net.Nodes.AddRange(dvs);
```

[3] Technically, any type of Chunk can be added as "knowledge" into the GKS.

```
reasoner.Commit(net);
```

These lines are used to initialize a HopfieldNetwork in the bottom level of the NACS of our agent. Note that the HopfieldNetwork is a so called "auto-encoder", and as such, is mainly used as an auto-associative memory network.[4] Initializing a HopfieldNetwork is slightly different than initializing your standard "feed-forward" network. In particular, since the HopfieldNetwork is conceptualized as asynchronous (meaning it technically doesn't have an input and output layer[5]), IWorldObject objects are actually just added to a general collection of "nodes" for this network instead of being specified as part of either the input or output layer.

Once our HopfieldNetwork is set up, we need to encode some knowledge into it. Auto-associative memory networks work by "reconstructing" encoded knowledge (or patterns) given a partial (or noisy) "input." For our current task, we will want to encode the 5 patterns (i.e., the declarative chunks) that were discussed previously.

The Encode method, in the ImplicitComponentInitializer, actually handles the majority of the encoding work.[6] The only thing we need to do to use this method is specify the "data sets" that are being encoded. Also, you may wish to perform a separate "test" run to ensure that the data sets are correctly recalled.[7] We can do this by simply calling the Encode method and specifying true for the "testOnly" parameter. Note that this would most often be done for cases where you wished to use a different TRANSMISSION_OPTION for the "encoding" and "testing" phases.

The following code, from the "simple reasoner" task, demonstrates how we might encode patterns into, and then "test" the recall accuracy of our HopfieldNetwork:

```
static void EncodeHopfieldNetwork(HopfieldNetwork net)
{
    double accuracy = 0;

    do
    {
        net.Parameters.TRANSMISSION_OPTION =
            HopfieldNetwork.TransmissionOptions.N_SPINS;

        List<ActivationCollection> sis = new List<ActivationCollection>();
        foreach (DeclarativeChunk dc in chunks)
        {
```

[4] The difference between auto-associative and hetero-associative networks is mainly conceptual. The bottom level of the NACS can actually store any combination of these two types of networks and both will function as expected according to their own purpose and capabilities.
[5] As a matter of implementation, the HopfieldNetwork actually uses the non-asynchronous methodology (i.e., with equivalently configured input and output layers). However, all interactions have been purposely designed so that the network can be initialized using either conceptualization.
[6] For more details on how to use this initializer, see the "*Useful Features*" tutorial (located in the "*Features & Plugins*" section of the "*Tutorials*" folder).
[7] Although the Encode method actually performs this step automatically, if the default UNTIL_ENCODED option is used.

```
            ActivationCollection si = ImplicitComponentInitializer.NewDataSet();

            si.AddRange(dc, 1);

            sis.Add(si);
        }

        ImplicitComponentInitializer.Encode(net, sis);

        net.Parameters.TRANSMISSION_OPTION =
            HopfieldNetwork.TransmissionOptions.LET_SETTLE;

        accuracy = ImplicitComponentInitializer.Encode(net, sis, testOnly: true);
    } while (accuracy < 1);
}
```

After we have encoded knowledge into the bottom level of the NACS, the next thing we need to do is generate and add associative rules to the top level.

Initializing Associative Rules

The process for initializing associative rules in the top level of the NACS is very similar to the process used to add action rules to the top level of the ACS. For our "simple reasoner" task, we want to set up 5 rules, with the following convention:

If pattern X, then conclude pattern X + 1

For example, if the input to the top level is the `DeclarativeChunk` representing pattern 1, then the top level should conclude the `DeclarativeChunk` representing pattern 2. The following code demonstrates how we would set up these sorts of associative rules in the top level of the NACS:

```
static void SetupRules(Agent reasoner)
{
    for (int i = 0; i < chunks.Count - 1; i++)
    {
        RefineableAssociativeRule ar =
            AgentInitializer.InitializeAssociativeRule(reasoner,
            RefineableAssociativeRule.Factory, chunks[i + 1]);

        ar.GeneralizedCondition.Add(chunks[i], true);

        reasoner.Commit(ar);
    }
}
```

Performing Reasoning

The last thing we may want to do before we initiate the reasoning process is to set any (optional) reasoning parameters. For our current task, we will need to set the following parameters:

```
reasoner.NACS.Parameters.REASONING_ITERATION_COUNT = 2;

reasoner.NACS.Parameters.CONCLUSION_THRESHOLD = 1;
```

The first parameter specifies that the NACS should perform 2 reasoning iterations before return its conclusions. The second parameter indicates that we only want those "fully activated" conclusions to be returned. There are many other reasoning parameters that can be set, and which will alter the behavior of the reasoning mechanism. For more information on them, see the "auto generated" documentation[8] for the `NonActionCenteredSubsystemParameters` class.

At this point, though, we should now be ready to start reasoning. Note that reasoning is currently only operational as a stand-alone mechanism. Future versions of the Clarion Library will provide a more natural integration into the overall system. However, as this integration is currently under development, to use the NACS's reasoning mechanism, you will need to call the `PerformReasoning` method (found in the NACS of an agent) and specify the "input" that is being used to initiate this reasoning:

```
var o = reasoner.NACS.PerformReasoning(si);
```

The `PerformReasoning` method will return the conclusion(s) from reasoning in the form of a collection `ChunkTuple` objects. The `ChunkTuple` is essentially just a "wrapper" for a conclusion `Chunk` and its associated activation (which specifies the "support" for that conclusion). For our "simple reasoner" task, we use a partial (noisy) reconstruction of each pattern as inputs (into 5 different rounds of reasoning). These "noisy" patterns are created by "zeroing-out" a percentage of each pattern. For example, with a noise value of .4, the final 40% of the input will have nothing but 0 activations.

The following code demonstrates how, for our current example, we might set up input patterns, initiate reasoning, and process the conclusions:

```
static void DoReasoning(Agent reasoner)
{
    int correct = 0;

    foreach (DeclarativeChunk dc in chunks)
    {
        ActivationCollection si = ImplicitComponentInitializer.NewDataSet();

        int count = 0;

        foreach (DimensionValuePair dv in dvs)
        {
            if (((double)count / (double)dc.Count < (1 - noise)))
            {
                if (dc.Contains(dv))
                {
```

[8] Located in the "Documentation" folder.

```
                    si.Add(dv, 1);
                    ++count;
                }
                else
                    si.Add(dv, 0);
            }
            else
                si.Add(dv, 0);
        }

        Console.WriteLine("Input to reasoner:\r\n" + si);

        Console.WriteLine("Output from reasoner:");

        var o = reasoner.NACS.PerformReasoning(si);

        foreach (var i in o)
        {
            Console.WriteLine(i.CHUNK);
            if (i.CHUNK == dc)
                correct++;
        }
    }
    Console.WriteLine("Retrieval Accuracy: " +
        (int)(((double)correct / (double)chunks.Count) * 100) + "%");
}
```

If everything is working correctly, we should see the following behavior:

- 1st iteration = the bottom level will complete the partial input pattern
- 2nd iteration = the top level will receive the conclusion associated with the "reconstructed pattern" from the bottom level and will conclude the following pattern
- Conclusions = the "conclusion chunks" from each reasoning iteration

For example, if the input is based on a "partial reconstruction" of pattern 1, the conclusions from reasoning should be the declarative chunks associated with patterns 1 and 2.

Finally, to complete our task, we will need to kill our agent (as always):

```
reasoner.Die();
```

This concludes our walk through to the "simple reasoner" task. At this point, you should have everything you need to get started on developing your own reasoning-specific tasks using the Clarion Library's NACS. If you are interested, you can learn more about how to integrate the NACS with the ACS in the "*Advanced ACS Setup*" tutorial located in the *Advanced* section of the *Tutorials* folder.

Setting Up & Using Episodic Memory

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Creating Episodes

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Initializing Associative Episodic Memory Networks

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Generating New Knowledge and Associative Rules

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Performing “Offline” Learning

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Remember, as always, if you have any questions, want to submit a bug, or make a feature request, please feel free to post on our message boards (http://www.clarioncognitivearchitecture.com) or email us at clarion.support@gmail.com and we will do our best to respond back to you as quickly as possible.