Useful Features

Table of Contents

Viewing an Agent's "Internals"

Reporting the outcome of a task usually involves at least some amount of investigation into what, exactly, the agent learned. We can retrieve any of the internal (functional) objects that are contained within an agent using the `GetInternals` method located in the `Agent` class. For example, suppose your task used bottom-up learning. In this case, we will likely want to know what rules were learned. The code below demonstrates how we could access the rules in the action rule store for our agent, John:

```
foreach (var i in John.GetInternals(Agent.Internals.ACTION_RULES))
    Console.WriteLine(i);
```

The `GetInternals` method takes, as its input, the `InternalContainers` enumerator. This enumerator lists all of the functional internal objects that are available for retrieval from within an `Agent`. This includes:

- Drives
- Action rules
- Implicit decision networks
- Associative rules
- Associative memory networks
- Associative episodic memory networks

- Meta cognitive modules[1]

The previous example (above) iterated through the rules that were returned by the `GetInternals` method and wrote them out to the console. Below is an example of a possible output from that code[2]:

```
Condition:
        (Dimension = GoalChunk, Value = Salute), Setting = True
        (Dimension = GoalChunk, Value = Bid Farewell), Setting = True
        (Dimension = Salutation, Value = Hello), Setting = True
        (Dimension = Salutation, Value = Goodbye), Setting = False
Action:
        ExternalActionChunk Hello:
                DimensionValuePairs:
                        (Dimension = SemanticLabel, Value = Hello)
```

The `GetInternals` method is not only useful for outputting the agent's internal (functional) objects. It also provides an easy way to retrieve these objects from an agent for the sake of performing other tasks (such as pre-training or "offline" training, parameter tuning, etc.).

While you will likely find this feature to be very useful, you will probably also find that you want to know more about the inter-workings of the internal processes being performed by your agent (for the sake of tracing or debugging your simulation). The next section (below) covers the logging features of the Clarion Library.

Logging (using `Trace`)

Often times, when in the process of building and tuning a simulation, you may find it useful to view the internal processes of your agent(s). For example, by adjusting the generalization and specialization thresholds, you can increase/decrease the rate at which your agent performs either type of refinement. However, you obviously need some way to determine these metrics in order to appropriately tune the parameters.

To address those situations where you may want to trace the internal processes of the system, the Clarion Library provides several different levels of logging by leveraging C#'s tracing mechanisms. By default, the logging level is set to "`Off`" (i.e., logging is not performed), however, you can specify a logging level by setting the

---

[1] Meta-cognitive modules have the `MetaCognitiveDecisionNetworks` and `MetaCogntiiveActionRules` properties that you can use to view the internal components for those modules

[2] The rule output was taken from a run of the "HelloWorld-Full.cs" simulation, which is located in the "Beginner" folder under the "Samples" folder in the Clarion Library package

`LoggingLevel` property located in the `World` singleton. The following code demonstrates how this might be done:

```
World.LoggingLevel = TraceLevel.Warning;

//Elided additional initialization of the simulating environment and agent(s)
```

The Clarion Library uses the default tracing levels that are defined by the `TraceLevel` enumerator[3]. Below is a breakdown of the kinds of things that are logged by the system at the various trace levels:

- `Off` – No logging is performed
- `Error` – Only a few, "abnormal", system behaviors are logged
- `Warning` – The system will "warn" you when certain, basic, events occur (e.g., the agent chooses an action, the goal structure or working memory are updated, rules are added or deleted from the top level of the ACS, etc.)
- `Info` – Similar to the `Warning` level, except it provides more detailed information. For example, it will inform you of when:
    - Certain mechanisms (such as the drive strengths, goal structure, or working memory updating threads) begin/end their processes
    - The ACS (or MCS) determines that a rule should be extracted/generalized/specialized/deleted
    - Components are eligible and/or used during decision-making (in the ACS or MCS)
    - A certain rule/action type is targeted during decision-making
    - Etc.
- `Verbose` – The most detailed logging level. The system will provide very detailed information about all of the internal mechanisms within the system (e.g., the state of all processes, i.e., threads; all events; anything specified by the lower logging levels, etc.)

The Implicit Component Initializer

One of the most difficult parts of initializing a Clarion-based agent is the process of setting up and pre-training implicit components such as neural networks. To address this issue, the `ImplicitComponentInitializer` has been provided to assist you with this process. The initializer can be used with any functional object that extends from the `ImplicitComponent` class.

In the sections that follow, we will go over the various features that are at your disposal when using the `ImplicitComponentInitializer`.

[3] Located in the *System.Diagnostics* namespace

Pre-Training

When in the course of developing a task, you may occasionally find that you need to "pre-ordain" your agent (or part of your agent) with a capability that is either not appropriate for, or simply can not be learned using standard "online" learning techniques (such as reinforcement learning). While this sort of "online" learning may not be a necessary aspect of your task, you may still want the bottom level of one or more subsystems to be "imbued" with some sort of proceduralized or "automated" functionality (e.g., capabilities or knowledge that are the consequence of an evolutionary process or the result of past experiences). For these types of instances, the Clarion Library provides a very useful feature known as the `ImplicitComponentInitializer`.

Before we begin, in order to pre-train an `ImplicitComponent`, we will first need two things:

1. A target that is to be trained[4]
2. A trainer to provide the correct (or "desired") output to the target

The trainer can really be any type of `ImplicitComponent` (e.g., an equation, a table lookup, another previously trained `ImplicitComponent`, etc.). However, the one thing you **MUST** be sure of is that it can provide the **correct** output(s) for all of the training data sets that are being using to train the target.

Note that, by default, the Clarion Library provides several built-in "extension" components[5] that can very be easily be designated as trainers (i.e., without needing any training themselves). The simplest of these extensions is the `GenericEquation`[6]. In the demonstration that follows, this component will be used in the role of trainer for a `BPNetwork`, that will act as the target. By using the `Train` method of the `ImplicitComponentInitializer`, the target component will learn how to report the value of a simple linear equation (i.e., $Y = X$), as specified by the `GenericEquation`.

To begin, we need to setup and initialize both our target and trainer. Let's assume that we want to use this target in the bottom level of the ACS. The steps needed in order to accomplish this have already been discussed elsewhere[7], so we will not go into those details here. Instead, let's move on to our trainer.

Initializing a trainer works slightly differently than the standard method for initializing an agent's internal functional objects. Specifically, trainer components are not required to exists within an agent. You can, of course, use a component that is within an agent if you so choose. However, as is more typically the case, you will likely just want to initialize your trainer externally for the sole purpose of training your target. To accomplish this, we use the `InitializeTrainer` method of the

---

[4] The target component **MUST** implement the `ITrainable` interface
[5] See the *Clarion.Framework.Extensions* namespace
[6] See the *Basic Customization* tutorial (located in the *Customizations* section of the *Tutorials* folder) for details about this component
[7] See the *Setting Up and Using the ACS* tutorial (located in the *Basics Tutorials* section)

ImplicitComponentInitializer. This method is essentially the same as the Initialize methods in the AgentInitializer, except that it does not "tie" the initialized component to an agent. The following code demonstrates how the InitializeTrainer method might be used to initialize a GenericEquation:

```
GenericEquation eq = ImplicitComponentInitializer.InitializeTrainer
    (GenericEquation.Factory, (Equation)LinearEquation);
```

A few things should be noted at this point. First, to initialize a GenericEquation, we must specify a delegate method (e.g., LinearEquation), which conforms to the Equation signature. This method is used by the component in order to calculate the activations for the "nodes" on the output layer.[8] Second, an ImplicitComponent, initialized using this method, can **ONLY** be used as a trainer. In other words, it is not possible to later use this component as an internal functional object within an agent. Third, like any component that is initialized using the AgentInitializer, components initialized using the ImplicitComponentInitializer **MUST** be "committed" before they can be used. However, unlike how an agent's internals are committed, to commit a trainer, we will need to call that component's own Commit() method.

The following code demonstrates how we might initialize both a BPNetwork (target), in the bottom level of the ACS of our agent, John, as well as initialize a GenericEquation (trainer):

```
DimensionValuePair x = World.NewDimensionValuePair("Variables", "X");
DimensionValuePair y = World.NewDimensionValuePair("Variables", "Y");
Agent John = World.NewAgent("John");

//Elided Agent Initialization

BPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
    (John, BPNetwork.Factory);

GenericEquation eq = ImplicitComponentInitializer.InitializeTrainer
    (GenericEquation.Factory, (Equation)LinearEquation);

net.Input.Add(x);
eq.Input.Add(x);

net.Output.Add(y);
eq.Output.Add(y);

John.Commit(net);
eq.Commit();
```

After we have initialized our trainer and target, the next thing that we need to do is setup training data sets. Each training data set should specify a different configuration for the activations of input layer of our trainer and target components. There are two options, however, for specifying training data sets. The most

[8] See the *Basic Customization* tutorial for more details

straightforward method is to simply create a bunch of data sets using fixed input activation patterns. This is the preferred method when you are working with a very specific, known, set of training data. The other method that is available to you is to define a **range** for each input node (or a subset thereof), between which training should occur (and at a specified increment).

To define a **range**, for a given node on the input layer, we will use the `AddRange` method in the `ImplicitComponentInitializer`. As part of calling this method, we will need to specify the following:

- The `IWorldObject` associated with the input node
- The upper and lower bounds for the range
- The increment at which the range should be traversed (optional)[9]

Below is an example of how we would define a range (between 0 and 1, with an increment of 0.1) for our variable "X":

```
ImplicitComponentInitializer.AddRange(x, 0, 1, .1);
```

Note that if a range has been specified for a particular input node, it will be used for that node, irrespective of if a data set specifies a fixed value for that node. Keep this in mind in case you are using the `ImplicitComponentInitializer` to train multiple networks as you may want to remove one or more ranges (by calling the `RemoveRange` method) between training operations.

You **MUST** create at least one training data set, even if you have defined **ranges** for all of your input nodes. The `NewDataSet` method in the `ImplicitComponentInitializer` can be used to generate new data sets. Each data set is represented as an `ActivationCollection`. You will notice very quickly, while creating and using data sets, that they work essentially the same as `SensoryInformation`. This is because `ActivationCollection` is actually the base class for `SensoryInformation`. The following example demonstrates how to setup a single data set for our variable "X":

```
List<ActivationCollection> dataSets = new List<ActivationCollection>();

dataSets.Add(ImplicitComponentInitializer.NewDataSet());
dataSets[0].Add(x);
```

Recall that we specified a range for our variable "X", so we do not need to worry about specifying an activation. However, if we hadn't specified a range for "X", then specifying an activation for "X" would be **exactly** the same process as is normally done for `SensoryInformation` objects.

At this point, we are now ready to begin training. The `Train` method takes the following as inputs:

- The target

[9] The default increment is 0.01

- The trainer
- The data sets (as a collection of `ActivationCollection` objects)
- A termination condition (optional, `FIXED` or `SUM_SQ_ERROR`)
- The number of times over which the data sets should be iterated (optional, if the `FIXED` termination condition is used)
- The threshold under which the sum of squared error must fall (optional, if the `SUM_SQ_ERROR` termination condition is used)
- The selection temperature (optional, if the target is `IReinforcementTrainable`)
- Whether the data sets should be traversed in random order (optional)
- Whether the call to the method is intended **only** to **test** the performance of the target given the data set (optional)

In order to initiate training on our target `BPNetwork`, using the `GenericEquation` trainer (with `SUM_SQ_ERROR` termination condition and the default threshold), we need to do the following:

```
ImplicitComponentInitializer.Train(net, eq, dataSets,
    ImplicitComponentInitializer.TrainingTerminationConditions.SUM_SQ_ERROR);
```

When the above code returns, the `BPNetwork` will be fully trained to report the outputs (as specified by the `GenericEquation`) for the training data set (i.e., the range of values that we defined for "X").

As a final note, you can follow the status of the training operation simply by enabling the Clarion Library's built-in logging feature (see the section above). In addition, the `ImplicitComponentInitializer` can also be serialized, thus allowing you to save and reload your range specifications.

Auto-Encoding

One key feature of Clarion is the use of implicit "auto-encoder" components (e.g., Hopfield networks[10]) in the bottom level of the NACS to help facilitate associative reasoning. These networks can be used to enable some of Clarion's more unique reasoning capabilities (especially with regard to the synergy of rule-based and associative reasoning processes).

With the above being said, encoding knowledge into these sorts of components can sometimes be a bit tricky. Therefore, in order to assist you with this process, the `ImplicitComponentInitializer` also provides an `Encode` method, which is specifically designed to "train" implicit components that implement the `IAutoEncoder` interface.

Currently, the `HopfieldNetwork` class is the primary "auto-encoder" that comes pre-packaged in the Clarion Library. Therefore, it will be used for our demonstrations on how to `Encode` using the `ImplicitComponentInitializer`.

[10] See the Clarion-H addendum to the technical specification document (located here)

The steps for initializing a target `ImplicitComponent` for encoding is actually very similar to what was described in the previous section for training. In fact, in many ways, the two processes are essentially the same. However, the main place in which they differ is that encoding does not require a "trainer." Instead, we simply need to specify the data sets that are being encoded into the auto-encoder, and the `ImplicitComponentInitializer` will handle the rest.

Similar to the `Train` method, the `Encode` method takes the following inputs:

- The target auto-encoder
- The data sets
- A termination condition (optional, `FIXED` or `UNTIL_ENCODED`)
- The number of times over which the data sets should be iterated (optional, if the `FIXED` termination condition is used)
- Whether the data sets should be traversed in random order (optional)
- Whether the call to the method is intended **only** to **test** the retrieval accuracy of the target given the data set (optional)

When the `Encode` method returns, the target will be fully encoded and will be able to rebuild any of the patterns in the data sets. In the code example below, we demonstrate how you could setup a `HopfieldNetwork` (in the bottom level of the NACS) with 10 nodes and use the `ImplicitComponentInitializer` to encode 3 activation patterns for those nodes (using the default encoding options):

```
//Initialize the 10 nodes
DimensionValuePair n1 = World.NewDimensionValuePair("Node", 1);
DimensionValuePair n2 = World.NewDimensionValuePair("Node", 2);
DimensionValuePair n3 = World.NewDimensionValuePair("Node", 3);
DimensionValuePair n4 = World.NewDimensionValuePair("Node", 4);
DimensionValuePair n5 = World.NewDimensionValuePair("Node", 5);
DimensionValuePair n6 = World.NewDimensionValuePair("Node", 6);
DimensionValuePair n7 = World.NewDimensionValuePair("Node", 7);
DimensionValuePair n8 = World.NewDimensionValuePair("Node", 8);
DimensionValuePair n9 = World.NewDimensionValuePair("Node", 9);
DimensionValuePair n10 = World.NewDimensionValuePair("Node", 10);

Agent John = World.NewAgent("John");

//Elided other agent initializations

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

//Add the 10 nodes to the Hopfield network
net.Nodes.Add(n1);
net.Nodes.Add(n2);
net.Nodes.Add(n3);
net.Nodes.Add(n4);
net.Nodes.Add(n5);
net.Nodes.Add(n6);
net.Nodes.Add(n7);
net.Nodes.Add(n8);
net.Nodes.Add(n9);
```

```
net.Nodes.Add(n10);

//Don't forget to commit the network!
John.Commit(net);

ActivationCollection[] patterns = new ActivationCollection[3];

//Pattern 1
patterns[0] = ImplicitComponentInitializer.NewDataSet();
patterns[0].Add(n1, 1);
patterns[0].Add(n2, 0);
patterns[0].Add(n3, 1);
patterns[0].Add(n4, 0);
patterns[0].Add(n5, 1);
patterns[0].Add(n6, 0);
patterns[0].Add(n7, 1);
patterns[0].Add(n8, 0);
patterns[0].Add(n9, 1);
patterns[0].Add(n10, 0);

//Pattern 2
patterns[1] = ImplicitComponentInitializer.NewDataSet();
patterns[1].Add(n1, 1);
patterns[1].Add(n2, 1);
patterns[1].Add(n3, 0);
patterns[1].Add(n4, 0);
patterns[1].Add(n5, 1);
patterns[1].Add(n6, 1);
patterns[1].Add(n7, 0);
patterns[1].Add(n8, 0);
patterns[1].Add(n9, 1);
patterns[1].Add(n10, 1);

//Pattern 3
patterns[2] = ImplicitComponentInitializer.NewDataSet();
patterns[2].Add(n1, 0);
patterns[2].Add(n2, 0);
patterns[2].Add(n3, 0);
patterns[2].Add(n4, 1);
patterns[2].Add(n5, 1);
patterns[2].Add(n6, 1);
patterns[2].Add(n7, 0);
patterns[2].Add(n8, 0);
patterns[2].Add(n9, 0);
patterns[2].Add(n10, 1);

ImplicitComponentInitializer.Encode(net, patterns);
```

Populating the Input and Output Layers of an Implicit Component

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).

Timing

You may have noticed by this point that timing is an important feature in the Clarion library. The system uses time stamps to track everything from the interactions between the various subsystems and modules to response times. In general these time stamps can be accessed using a (somewhat ubiquitous) property called `TimeStamp`. In the sections that follow, we will look at two particular features that use this timing and that may be useful for your Clarion-based project.

Response Time

Many tasks, especially those which aim to explore human cognitive phenomenon, place importance on the role of response times as a measurement for performance. Therefore, as part of the process of action decision-making, the Clarion library also keeps track of various timings. These timings include:

- Perception time
    - Includes drive strength updating operations in the MS and any "pre-action selection" operations (e.g., goal setting) in the MCS
- Decision time
    - Varies based on the level selection method that is used in the ACS
- Actuation time
- Reasoning time

Each of the above timings are tunable via parameters. These parameters can be located as follows:

```
//Perception Time
Agent.GlobalParameters.PERCEPTION_TIME
John.Parameters.PERCEPTION_TIME

//Actuation Time
Agent.GlobalParameters.ACTUATION_TIME
John.Parameters.ACTUATION_TIME

//Decision Time
ActionCenteredSubsystem.GlobalParameters.TOP_LEVEL_DECISION_TIME;
John.ACS.Parameters.TOP_LEVEL_DECISION_TIME

ActionCenteredSubsystem.GlobalParameters.BOTTOM_LEVEL_DECISION_TIME;
John.ACS.Parameters.BOTTOM_LEVEL_DECISION_TIME

//Reasoning Time
NonActionCenteredSubsystem.GlobalParameters.REASONING_ITERATION_TIME;
John.NACS.Parameters.REASONING_ITERATION_TIME
```

Every time an agent perceives, the action that results from that perception will also be accompanied by a response time. In general, this response time is determined by:

$$ResponseTime = PerceptionTime + DecisionTime + ActuationTime$$

If your simulation is being run in "asynchronous mode", this response time will be provided to you by default as part of the parameters passed into the `ProcessChosenExternalAction` method. Otherwise, to get the response time related to a particular perception you need to do the following:

```
long rt = John.GetResponseTime(si);
```

Note that the system will only hold onto the response time data for so long (as determined by the `PREVIOUS_RT_CAPACITY` and `LOCAL_EPISODIC_MEMORY_RETENTION_THRESHOLD` parameters, located in the `Agent` and `ActionCenteredSubsystem` classes respectively), so this method should typically be called immediately after the `GetChosenExternalAction` method is called.

The response time represents the time that it took the agent to perform a particular action (which was initiated by perceiving a sensory information object). Conceptually, the response time .

In fact, the system already uses these response times in a variety of ways:

- The `MAX_RESPONSE_TIME` is used by default to increment the time stamp when a new `SensoryInformation` object is created using `World.NewSensoryInformation...`
    - Note that the actual response time can also (optionally) be used
- To determine when an action is delivered to the simulating environment
    - By adding the response time to the time stamp of the affiliated perception

By default, response times can tend to look fairly one-dimensional since the calculation of these times are based on parameters that don't vary (except, of course, for decision times). In reality, however, human response times fluctuate a bit, even when tasks are highly learned and proceduralized. Therefore, in the Clarion library, a set of parameters have been added to allow for variability in both the perception and actuation times. This variability can be added by simply changing the following parameters:

```
Agent.GlobalParameters.PERCEPTION_TIME_VARIABILITY_THRESHOLD
John.Parameters.PERCEPTION_TIME_VARIABILITY_THRESHOLD

Agent.GlobalParameters.ACTUATION_TIME_VARIABILITY_THRESHOLD
John.Parameters.ACTUATION_TIME_VARIABILITY_THRESHOLD
```

Once set, the perception and actuation times will vary $\pm threshold$. The distribution of this variation is normalized by default. However, if you prefer to use a different

distribution, you can also specify your own custom variability calculator using the following delegate:

```
delegate long ResponseTimeVariabilityCalculator(long defaultTime, double
threshold);
```

Methods written using this delegate signature can be used by an agent to calculate either the perception or the actuation time variability. Feel free to use the same method for either timing, as the `defaultTime` and `threshold` parameters will be specific to each timing. However, note that you can also specify different distributions for these timings if you so desire.

If you do choose to write your own variability calculator, be aware that you must set it for the agent in whom it is to be used. This is done by calling either of the following properties:

```
John.PerceptionTimeVariabilityCalculator =
     (ResponseTimeVariabilityCalculator)SomeCalculatorMethod;
John.ActuationTimeVariabilityCalculator =
     (ResponseTimeVariabilityCalculator)SomeCalculatorMethod;
```

"Real-time" Mode

As was mentioned earlier, timing in the Clarion library is everything. However, the system keeps track its own time stamps, which are not actually correlated to the speed at which the system runs. In other words, it has no basis on the time within the real world. By default, the system will simply run as quickly as it possibly can. This is likely preferred for most tasks, as it would be very laborious if you had to wait for more than 1 second each time your agent perceived something.

That being said, you may find cases where you will want your agent to operate in "real-time" (e.g., when actually interacting with the real world). In these cases, agents can be forced to take the amount of time they are supposed to take when performing various operations. For example, if perception time takes 200 milliseconds, then an agent can be made to will wait that amount of time before it starts performing decision-making; if actuation time takes 500 milliseconds, then an agent will wait that long before delivering an action to the outside world; and so on and so forth.

Turning on "real-time" mode is as simple as flipping a switch. Below is an example of how we can specify that our agent, `John`, should run in real-time mode:

```
John.Parameters.IN_REAL_TIME = true;
```

Asynchronous Operation

As we have mentioned a couple of times throughout these tutorials, the Clarion Library is an asynchronous (i.e., multi-threaded) system that leverages an advanced publisher/subscriber event model in order to facilitate the interactions between all

of the internal mechanisms within an agent. Because of this fact, the Clarion Library can also be setup to interact asynchronously with a simulating environment.

Setting up the simulating environment to interact asynchronously with agents is actually a fairly simple process. All we need to do is extend the abstract `AsynchronousSimulatingEnvironment`[11] class:

```
public class SomeSimulatingEnvironment : AsynchronousSimulatingEnvironment
{
... //Elided simulation code
}
```

After we have extended this class, there are only two other things that need to be done:

1. Override the abstract `ProcessChosenExternalAction` method (which is defined by the `AsynchronousSimulatingEnvironment` class)
2. Register the asynchronous simulating environment with the agent (by calling the agent's `RegisterAsynchronousSimulatingEnvironment` method)

The follow code sample demonstrates how we might accomplish the above for our agent, `John`:

```
// Register the simulating environment in the initialization section
John.RegisterAsynchronousSimulatingEnvironment(this);

...

protected override void ProcessChosenExternalAction(Agent actor,
    ExternalActionChunk chosenAction, SensoryInformation relatedSI,
    Dictionary<ActionChunk, double> finalActionActivations, long performedAt,
    long responseTime)
{
    ... //Elided code to process the agent's chosen action and deliver feedback
}
```

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.

[11] Located in the *Clarion.Plugins* namespace