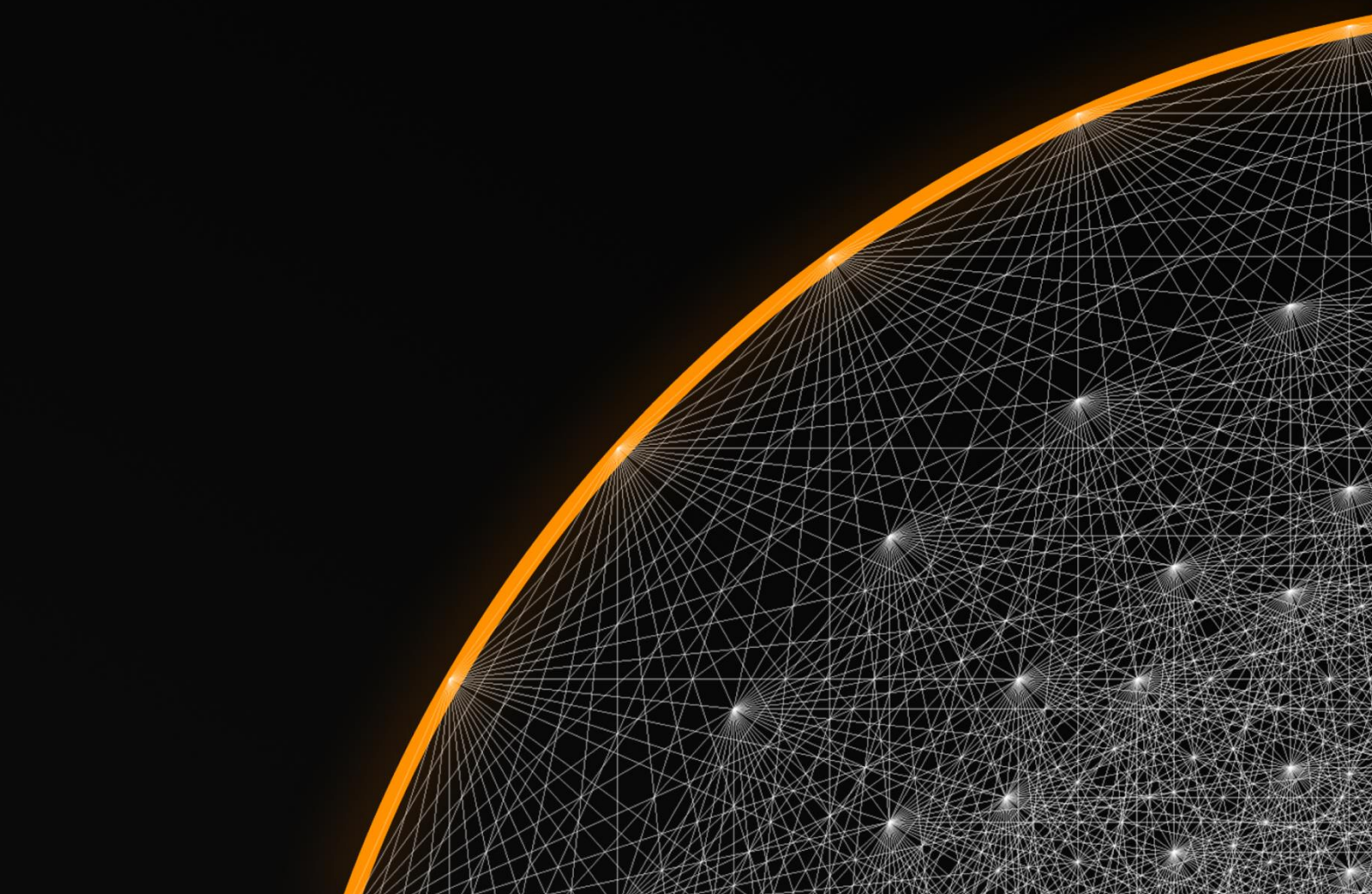
coAdjoint

Action Editor For Unity

User Guide



Contents

System Requirements

- Unity 4.x, Unity 4.x Pro or Unity 5.x
- Graphics card with DirectX 9 level (shader model 2.0) capabilities

Quick Start

To get up and running with coAdjoint Action in 20 minutes:

1) Open Unity. Make sure that the currently loaded project is the one you wish to install coAdjoint Action in.

2) Open the Asset Store by clicking **Window** on the toolbar, then click **Asset Store**.

3) In the Asset Store tab, focus the search bar and type 'coAdjoint Action'.

4) In the search results, select coAdjoint Action and click **Buy** and complete the purchase process. Once payment has been accepted by Unity, coAdjoint Action will start downloading to your hard-drive.

5) Once the download is complete, the **Importing Package** window will open. Press **Import** in the bottom right of the window. This will import coAdjoint Action into your current project.

6) In the project tab, click **Create**, then select **Action Skill Library**. In the pop-up dialog box, name your new skill library. A new Action skill library asset will appear in the **Libraries** folder of your project with the name you have just chosen.

7) In the Project view, navigate to this skill library in the **Libraries** folder and highlight it. In the Inspector, click **Edit Library**. Two new dockable windows will open titled **Library Explorer** and **Network Editor**.

8) In the Library Explorer, click **Add** and type a name in the dialog window. This will add a new Action skill network to your library.

9) Highlight the network in the Library Explorer, focus the Network Editor and click the **Add Skill to Network** button in the top left of the screen. Fill in the name of the skill and the Q, B and v fields (for

information on what these fields mean see the 'Step by Step Guide') and press **Ok**. A skill will now be anchored to your mouse.

10) Click anywhere in the Network Explorer to dock the skill to the **canvas**. You can reposition it by highlighting the skill and clicking and dragging.

11) Repeat this process to create several new skills. You can also create a new skill with the keyboard shortcut 'Ctrl+F'.

12) Let's now connect our skills together. Shift-click on two skills to highlight them. Press the **Create Connection between Skills** button located to the right of the Create New Skill button, or press 'Ctrl+D' on your keyboard.

13) Assign values to the strengths of the connections between skills. The values can be 0 but cannot be negative. It is recommended that you use values less than $1/n$, where n is the number of skills in the network.

14) Repeat this process to create several more skills. Try pressing the **Create Connection Between Skills** button with only one skill selected and see what happens.

15) Now add another new network to your library by repeating steps 9 to 13.

16) To export your library to use it in a game, highlight your skill library in the project tab. In the Inspector, click **Generate Runtime Library from Asset**; a .NET assembly will be added to the **Builds** folder of your project. Once your purchase is registered, you will be able to instantiate Action networks in scripts.

17) Create a new scene in your current project and add an **empty game object** to the scene by clicking **GameObject->Create Empty.**

18) Create a new UnityScript or C# script. To access the networks in your library, include the name of the new assembly. For example, if the assembly is called 'NewLibraryBuild.dll', type '`using`

`NewLibraryBuild;'` if you are using C#, or `'import NewLibraryBuild;'` if you are using UnityScript.

19) To instantiate an Action network from your library, use the following code:

```
<NETWORKNAME> myNewNetwork = new <NETWORKNAME>();
```

where `<NETWORKNAME>` is the name of a network in your skill library.

20) To level up a skill in your network, use the `myNewNetwork.Levelup` method. For example, if your network contained a skill called MyBestSkill, type:

C#:

```
void Start()
{
    Debug.Log("Initial level was"  +
          myNewNetwork.GetLevelAsDouble(<NETWORKNAME>.
               skills.MyBestSkill));

    myNetwork.LevelUp(<NETWORKNAME>.Skills.MyBestSkill,2);
}

void Update()
{
    if(Input.GetKeyDown(Keycode.A))
    {
          Debug.Log("Now the level is" +
               myNewNetwork.GetLevelAsDouble(<NETWORKNAME>.
                    skills.MyBestSkill));
    }
}
```

UnityScript:

```
function Start()
{
     Debug.Log("Initial level was"  +
          MyNewNetwork.GetLevelAsDouble(<NETWORKNAME>.
               skills.MyBestSkill));

     myNetwork.LevelUp(NETWORKNAME.Skills.MyBestSkill,2);
}
```

```
function Update()
{
    if (Input.GetKeyDown(Keycode.A))
    {
        Debug.Log("Now the level is"  +
          myNewNetwork.GetLevelAsDouble(<NETWORKNAME>.
               skills.MyBestSkill));
    }
}
```

21) Play the scene and press the 'A' key on the keyboard. If the level of your network has increased (as demonstrated in the Console), you have successfully implemented coAdjoint Action in a game!

Introduction

coAdjoint Action uses advanced mathematical techniques to create the most intricate and dynamic character progression system ever seen in video games. Despite this, the complexity of the model is completely hidden by an intuitive graphical editor and incredibly simple Application Programming Interface (API), placing control entirely in the hands of designers and developers.

This User Guide will answer important questions about coAdjoint Action, explain the Action Editor for Unity in great detail and also provide an API reference for scripting in Unity with Action.

We now answer some frequently asked questions about coAdjoint Action.

What's the point of coAdjoint Action?

However popular the current character development systems (level up systems, experience points, skill trees, etc.) are, they place severe restrictions upon the creativity and freedom of the gamer. We have always felt a deep dissatisfaction with level up systems, where mindless grinding is often rewarded with predefined progression or overly simplistic choices. These unnatural, often out of place systems are made redundant by coAdjoint Action, freeing designers and gamers to create characters and worlds that adapt and react to the actions and style of the user.

What makes it different?

coAdjoint Action differs from other skillpoint systems in two ways:

1) Skills are related to one another: just as in real life, skills can now have positive influences upon each other. For example, if a designer so wishes, using coAdjoint Action a strong character will become a faster sprinter than a weak character.

2) Updates to skills are designed to occur in real-time: coAdjoint Action is highly optimised and capable of performing updates whilst the game is being played. As such, every decision that a user makes can change the profile of their character, creating continuous, dynamic gameplay evolution.

These facts allow you to consider character progression in novel ways, opening up new possibilities in tactical game play and gamification. There are many exciting applications for coAdjoint Action, such as educational software, next generation RPGs, fitness apps and adaptive first person shooters. Many of the options that coAdjoint Action creates would be extremely difficult to implement without coAdjoint Action and its uncomplicated editor.

How does it work?

To explain how coAdjoint Action works, we must first define four key terms that we have already mentioned:

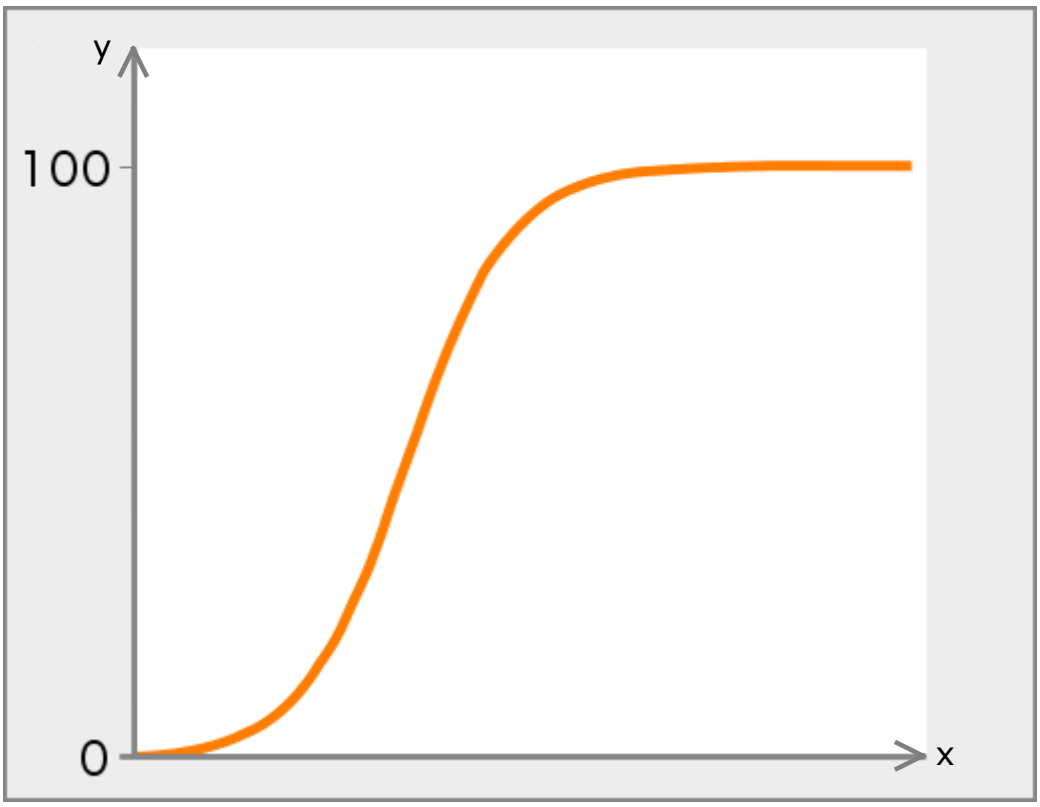
A **character** is an object in a game representing an intelligent entity (such as an NPC dog, Gordan Freeman in the Half-Life games, or even the human using the game themselves).

A **skill** is a task, role or ability that a character may perform or possess. A character can improve their ability in a skill by training that skill.

A skill X is **connected** to a skill Y if an improvement in the ability of X causes an improvement in the ability of Y (note: X being connected to Y does not imply that Y is connected to X).

A **network** is a set of connected skills.

coAdjoint Action models a character as a network. Skills are initially modelled as generalised logistic curves, as shown overleaf.



The x-axis of the graph represents the **experience** of the skill. Experience (or training) is a measure of the number of tasks a character has performed that contribute towards a skill. Different tasks may have different experience values: for example, if skill A is improved by performing task S and task T, task S may contribute 5 experience points, whilst task T may only contribute 2.5 experience points.

The y-axis of the graph represents the **level** of the skill. A skill's level is a measure of a character's ability in a particular skill: the higher the level, the greater the character's ability at using that skill. It is important to note that during the mathematical process of calculating skill levels, the values are normalised between 0 and 100, and that the level of a skill can **never reach 100**, but only asymptote towards it.

Each curve starts with 0 experience and 0 level. Changes in the network (i.e. improvements in skills) perform transformations on these skill curves as determined by the EPDiff equations, an area of mathematics studying the evolution of functions. These transformations shift the profile of the skill curves so that when you increase the experience of a skill, the level of the skill will increase at a faster rate than if no change in the network had occurred.

For example, consider a network of two skills, skill A and skill B, with skill A connected to skill B (A affects B). An improvement in skill A will have a carry-through effect to skill B, allowing skill B to improve at a faster rate

than if no improvement had occurred in skill A. A network of many skills works in exactly the same manner.

How does the licensing system work?

coAdjoint are currently using a 2-tier licensing system for the Action Computational Engine. The version of the Software Bundle currently available on the Unity Asset Store entitles you to the Light Distribution Licence. This license is designed for sharing demos or small projects. Once the executable game has started running, coAdjoint Action will only continue to work for 6 hours. Restarting the demo will allow for another 6 hour period of game-play using coAdjoint Action.
The Full Distribution license is designed for full releases of your game. This license gives you unrestricted use of coAdjoint Action on any computer for any period of time. Upgrading to the full license requires the acceptance of a new End User Licence Agreement and is free for companies with less than 10 employees. Please email support@coadjoint.co.uk for more information, or to request a full distribution license.

What is coAdjoint Orbit and how is it related to coAdjoint Action?

coAdjoint Orbit is a Unity editor extension currently available on the Unity Asset Store. coAdjoint Orbit uses Action networks to create a powerful adaptive Artificial Intelligence (AI) tool. coAdjoint Orbit is able to classify the skill levels of a character, and process that information to change AI behaviour based upon the calculated classification. For example, if a character in a first person shooter using coAdjoint Orbit was acting stealthily, the AI could adapt to this knowledge by utilising more search lights and being more aware of noises and disturbances. On the other hand, if the character was using a rocket launcher and wreaking havoc, the AI could utilise more defensive tactics and call in heavily armoured reinforcements. As with coAdjoint Action, coAdjoint Orbit is fully integrated into Unity and uses an intuitive user interface to control the powerful mathematics behind the code.

Step by step guide

In this section, we explain the process of using coAdjoint Action, from purchasing the software on the Unity Asset Store to utilisation of coAdjoint Action in your games. For a quick introduction to using the Action Editor for Unity and scripting with coAdjoint Action, please refer to the quick start guide at the beginning of this manual.

Purchasing coAdjoint Action

This section will explain how to download the coAdjoint Action Editor for Unity and to register your purchase of coAdjoint Action.

Note: coAdjoint Action, in its current incarnation, has been highly customised for use in Unity. If you are interested in using coAdjoint Action outside of Unity (for example on different platforms or in different programming environments) then please contact coAdjoint at enquiries@coadjoint.co.uk.

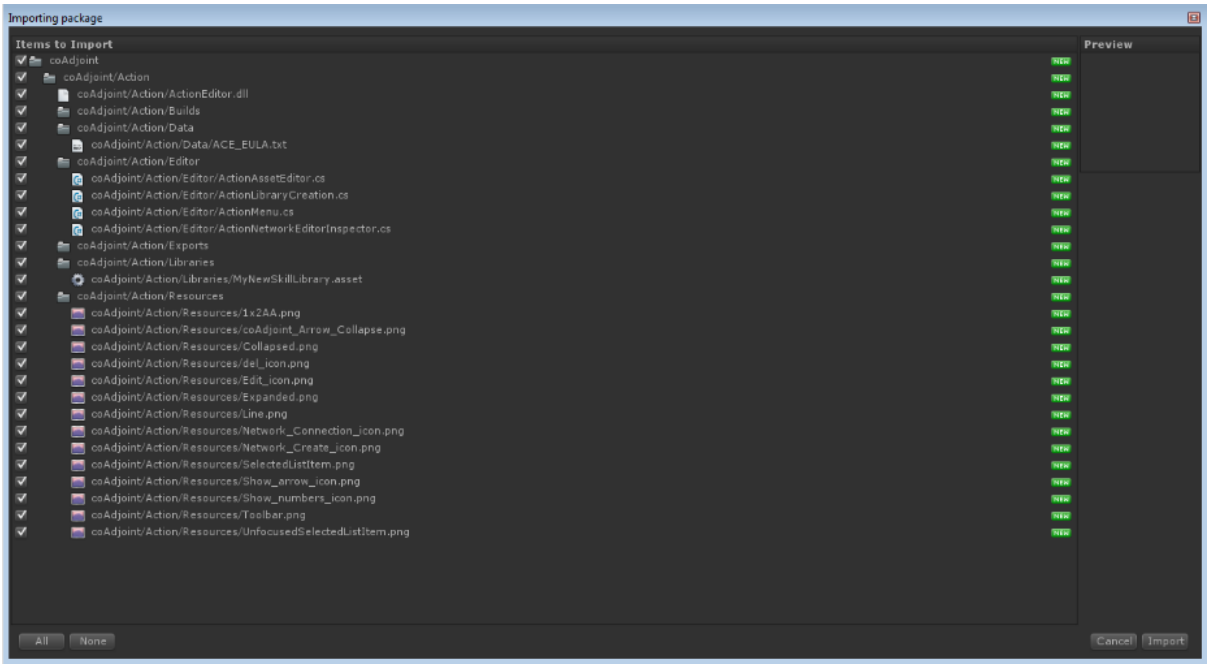
The purchase process for coAdjoint Action is incredibly simple.

- Open Unity for Windows. Make sure that the currently loaded project is the one you wish to import coAdjoint Action into.
- Open the **Asset Store** by clicking **Window->Asset Store**.
- In the Asset Store tab, find the search bar and type 'coAdjoint Action'.
- In the search results, select coAdjoint Action and click **Buy.**
- Enter in your payment details.

This will begin the download of coAdjoint Action onto your computer.

Note: The version of coAdjoint Action on the Unity Asset Store entitles you to register the standard Developer Version. For more information about the different versions of coAdjoint Action, please see 'How does the licensing system work?' in the Introduction).

Once download is complete, an **Importing Package** window will open. To import coAdjoint Action, press Import in the bottom right of the window.



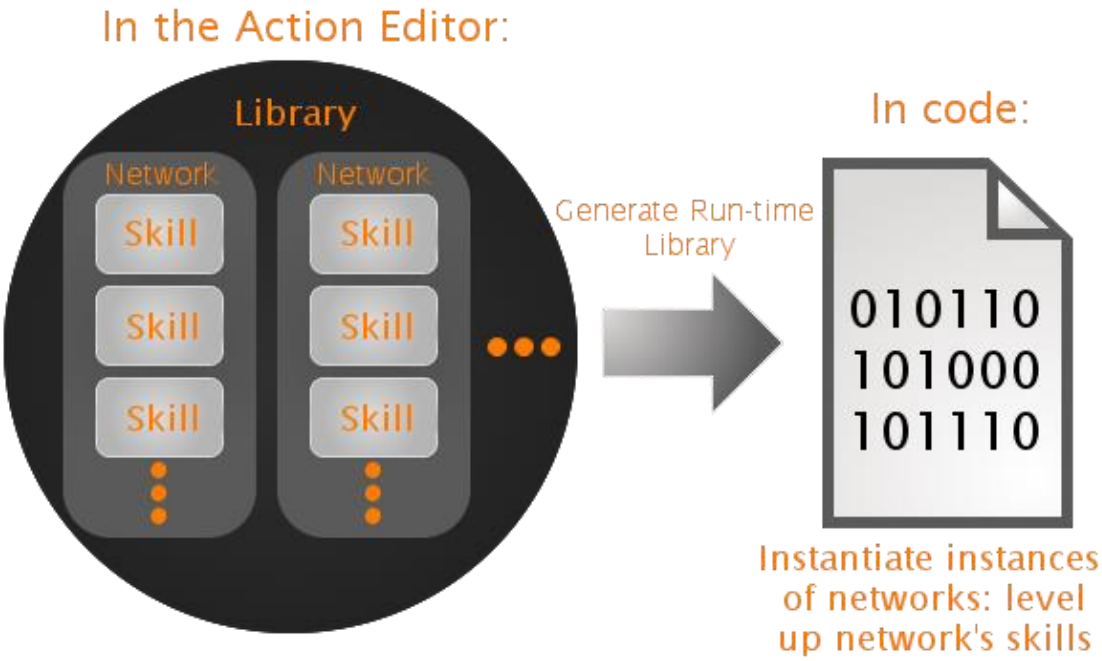
This will extract the **ActionEditor.dll** assembly into your project, along with several editor scripts and folders. We recommend that you do not attempt to edit these scripts. We also recommend that you register your purchase straight away to minimise waiting times between creating Action Networks in the Editor and using them in your games.

Let us now begin working in the coAdjoint Action Editor.

The Action Editor

There is a simple hierarchy to working with coAdjoint Action (see image below).

In the **Editor**, the first step is to create an **Action library**. A library is a placeholder for all the **Action networks** you want to include in a project, such as a demo or a game. Each library can hold as many networks as you want, and as you will see the creation of networks in the Network Editor is simple.



As explained in the introduction, each network contains **Action skills**: a network can contain as many skills as you want, and each of these skills can be **connected** to any number of other skills in a network (it should be noted that skills in different networks cannot interact).

Note: although not necessary, we recommend that you have one project for all your Action libraries and export completed libraries to your games, rather than import coAdjoint Action into many projects. Doing so will keep both your Action libraries and game projects organised.

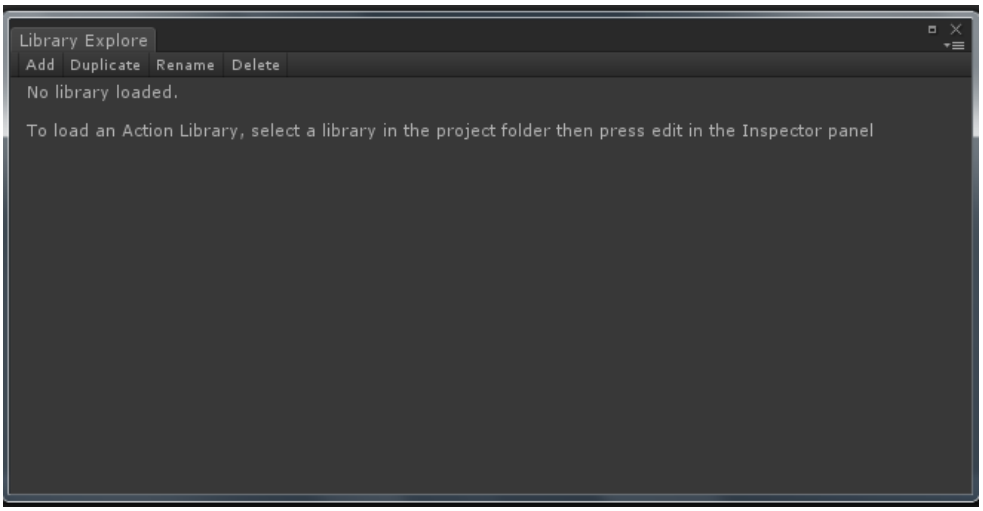
When you create a library, it does not exist outside the Action Editor for Unity. This means that you cannot use the networks you have created in an Action library in scripts in a game. To use networks, you have to generate a **runtime library** from your Action library. This creates a .NET assembly file that contains all the metadata about your Action library, but in a format that is useable in code. By using a generated runtime library you can instantiate networks and add experience to skills inside a network whilst your game is running.

For example, for a new puzzle game **BusyBots** you decide to use two networks: one that will be used by both the player and an enemy and one that will only be used by a different enemy to provide variation for the player. To implement this scenario you would create an Action library called **BusyBotsLibrary**, and then add two Action networks, perhaps called **SharedNetwork** and **AltEnemyNetwork**.

If these networks had shared features you may decide to create all the shared features in one network first, duplicate it and then rename it. This would be achieved using the Action Library Explorer.

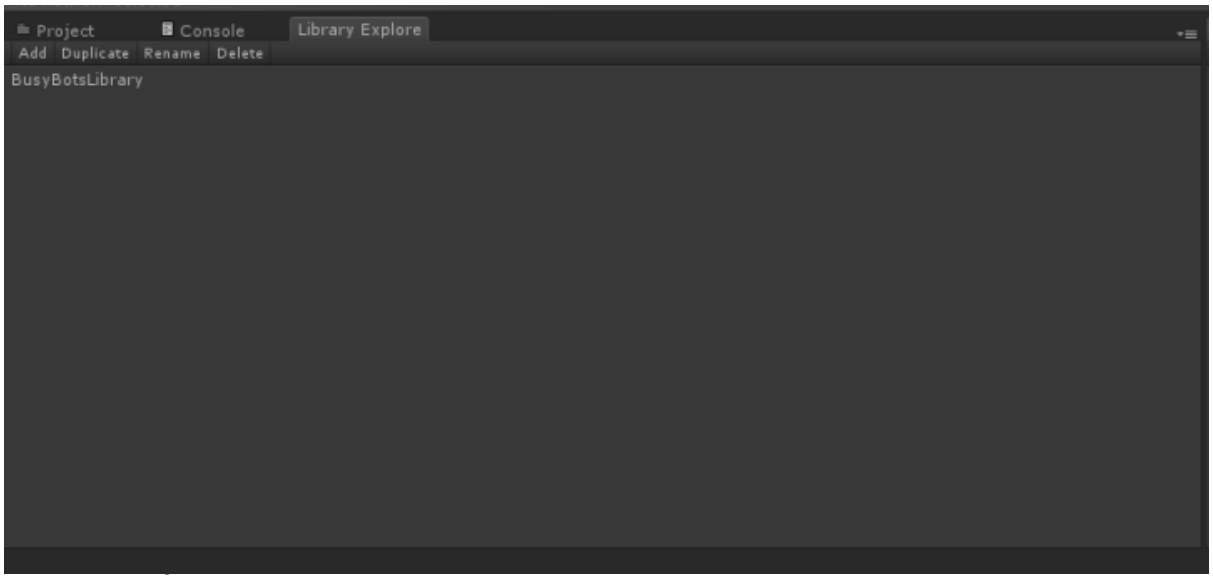
The Action Library Explorer

The **Action Library Explorer** provides a convenient tool to edit a network inside a particular library. To open it, navigate to **Window->Action Library Explorer**. This will open up the Library Explorer displaying a message that no Action library is currently open, as shown in the image below.



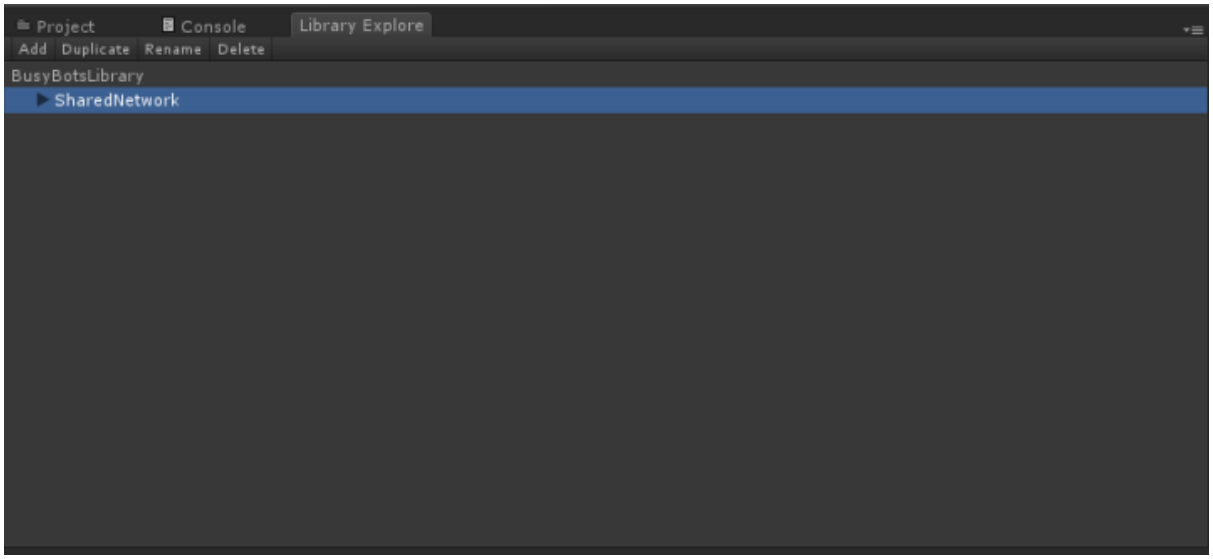
To remedy this situation, focus the project view, click **Create** and select **Action Skill Library**. This will open a dialog box where you can name your new Action library. Continuing with the BusyBots example, let's name this library BusyBotsLibrary and press **OK**.

Focus the Library Explorer. You will now notice that the name of your library is listed at the top of the window as shown below.



Next to the name of the library there is an arrow. Click on it: a message will appear underneath the library name explaining that this library contains no networks. To add a network, click **Add** on the toolbar above the library name. A new dialog box will open prompting you to pick a name for this new network. Let's name this network SharedNetwork.

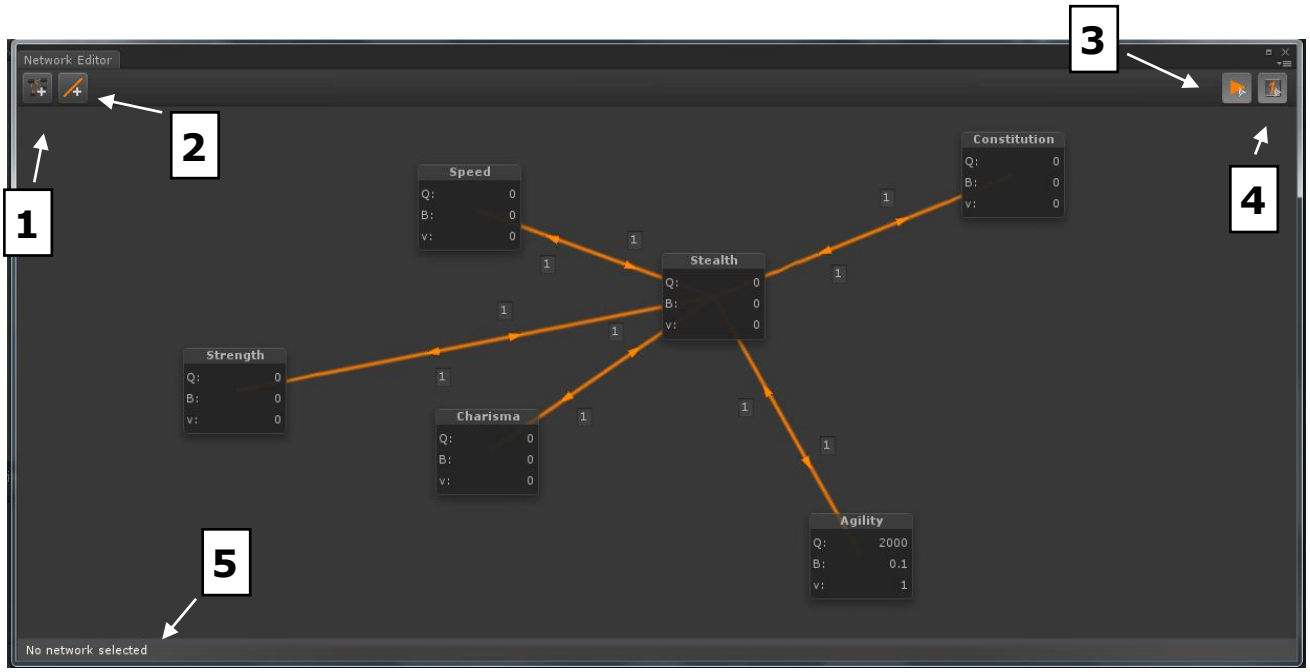
The Action Library Explorer should look like this (see below):



Now, let's take a look at building up a network in the Network Editor.

The Action Network Editor

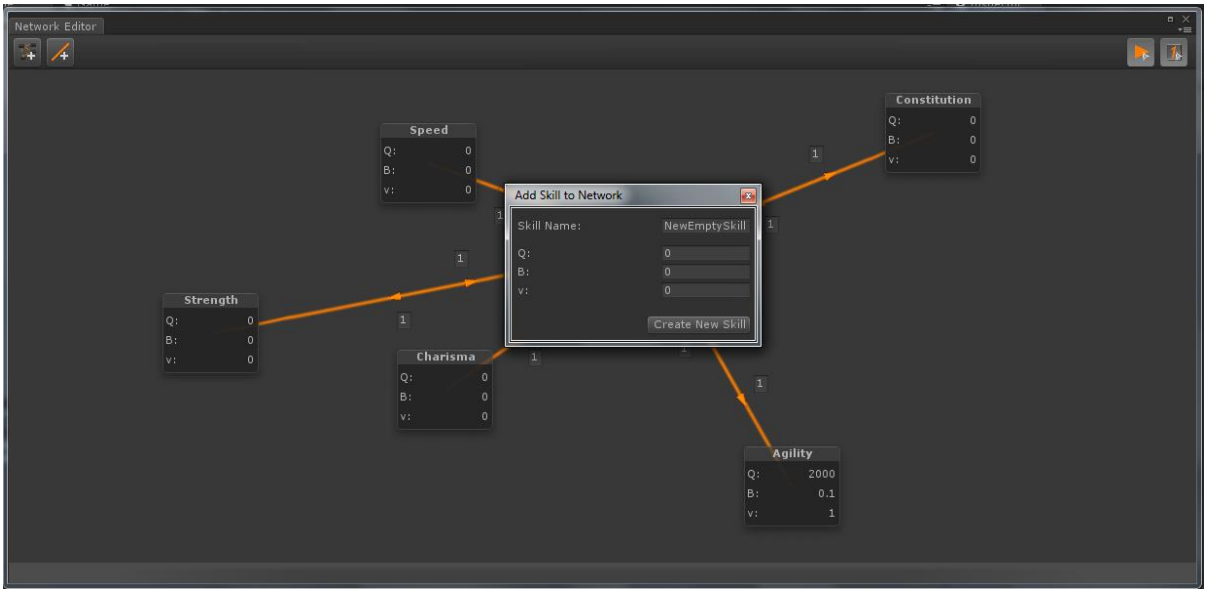
The Action Network Editor is the graphical user interface for the creation of Action networks. If it is not already open, navigate to **Window->Action Network Editor** to open it. The diagram overleaf highlights the important features of the editor.



Key:

1 – Create new skill
2 – Create new connection
3 – Show connection arrows
4 – Show connection values
5 – Tooltip panel

Let's create our first skill in the SharedNetwork Action network. In the Library Explorer, make sure that SharedNetwork is highlighted. In the Action Network Editor window, click the **Create New Skill** button. This will bring up the dialog box shown below:

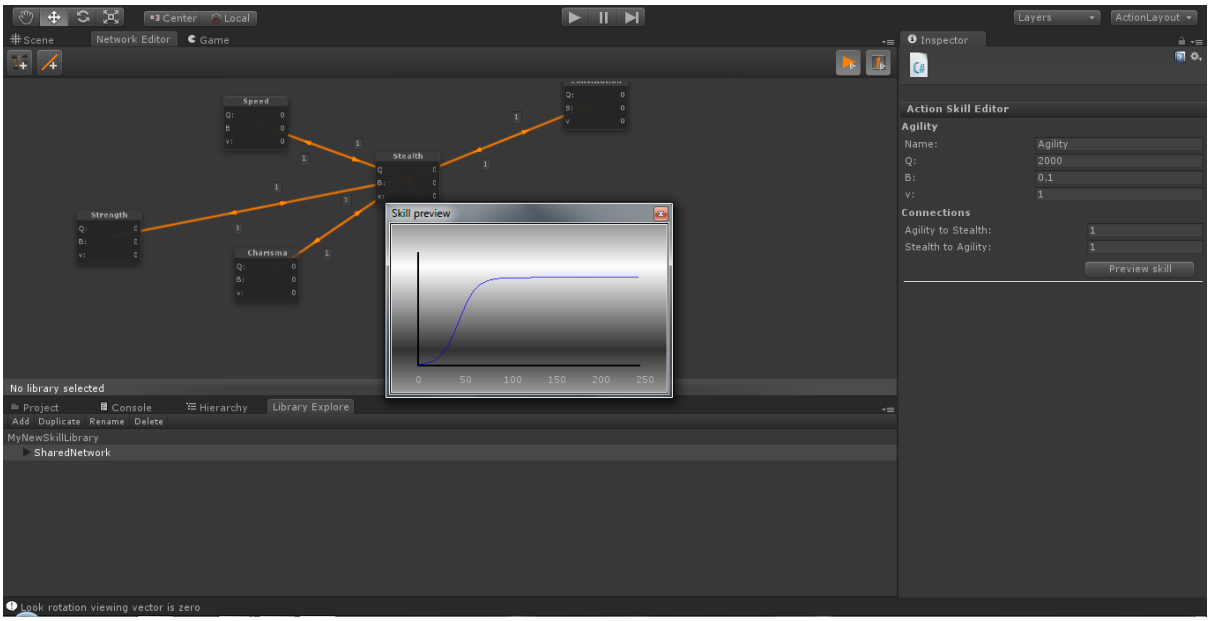


This dialog box contains four fields: **Skill Name**, **Q**, **B** and **v**. The Q, B and v fields allow you to define the parameters that control the profile of the skill curve associated to this skill. Although this profile will change throughout the game as the user provides input to the network, the initial profile of a skill significantly affects its future profile. This is analogous to reality, where genetic characteristics of a person have a lasting effect on the way they improve in a skill.

The differences between each of these parameters and the effects they have on the profile of a skill will be easier to understand once we have learnt more about the Network Editor.

For now, let's name this skill **Speed** and set Q to 2000, B to 0.1 and v to 1.

Press **Create New Skill** - a box representing the Speed skill we just created will now be anchored to your mouse. Click on a position in the Network Editor which isn't occupied by a toolbar to place the skill in the network. This will dock the skill to the Network Explorer in an area known as the **Network Canvas**. Click anywhere on the newly created skill to highlight it. The box should now turn green, and the Inspector will now contain information about the skill. Click on the **Preview Skill** button in the Inspector. This will bring up a utility box showing a graph of the skill's profile as it currently stands.



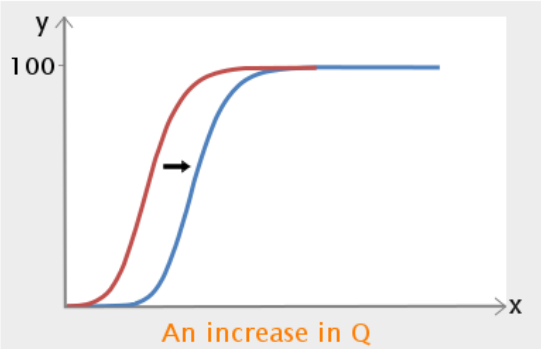
As explained in the introduction, as experience (shown on the x-axis) is applied to the skill, the level of the skill (shown on the y-axis) increases.

Keeping the Skill Preview Window open, change the values of Q, B and v for this Skill and notice how the profile of the skill changes.

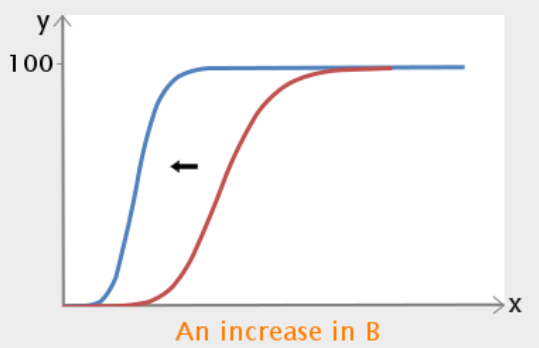
Changing Q, B and v have highly non-linear effects on the shape of the skill curve. However, we can say that:

- **An increase in Q leads to a shift in the profile of the skill to the right.**
- **An increase in B leads to a shift in the profile of the skill to the left.** Higher B values leads to a skill reaching as close as possible to level 100 (known as **saturation**) sooner than lower B values.
- **An increase in v decreases the steepness of the learning curve of the skill.**

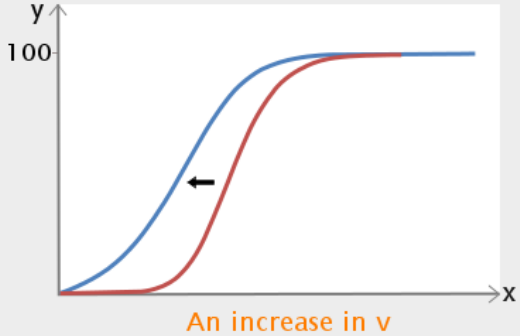
The graphs below shows graphical examples of changes in Q, B and v, with the other parameters fixed.



An increase in Q



An increase in B



An increase in v

Note: when defining an Action Skill, we recommend that you spend some time thinking about how this skill should work in your game. Some useful questions to consider are:

- How much experience must the player input to this skill before they reach saturation?
- How steep should the learning curve of the skill be? Can a character pick the skill up quickly or does it take a considerable amount of experience before the character can see some progress? This factor is mainly controlled by the v parameter but the Q parameter is also important to consider.

Now that we have seen all the features of one skill, let us create another skill. Call it what you want and set the skill parameters as you wish. Place it anywhere on the skill canvas.

Controlling the Network Editor

Let's now examine the controls of the Action Editor.

To **pan** around the Canvas, click and drag the middle mouse button or press 'Alt+Left Mouse Button'.

To **zoom** in or out on the Canvas, scroll the mouse wheel, or click and drag 'Alt+Right Mouse Button'.

To **centre** a skill in the Network Canvas, in the Action Library Explorer expand the current network by clicking on the arrow next to its name and highlight the skill you wish to find.

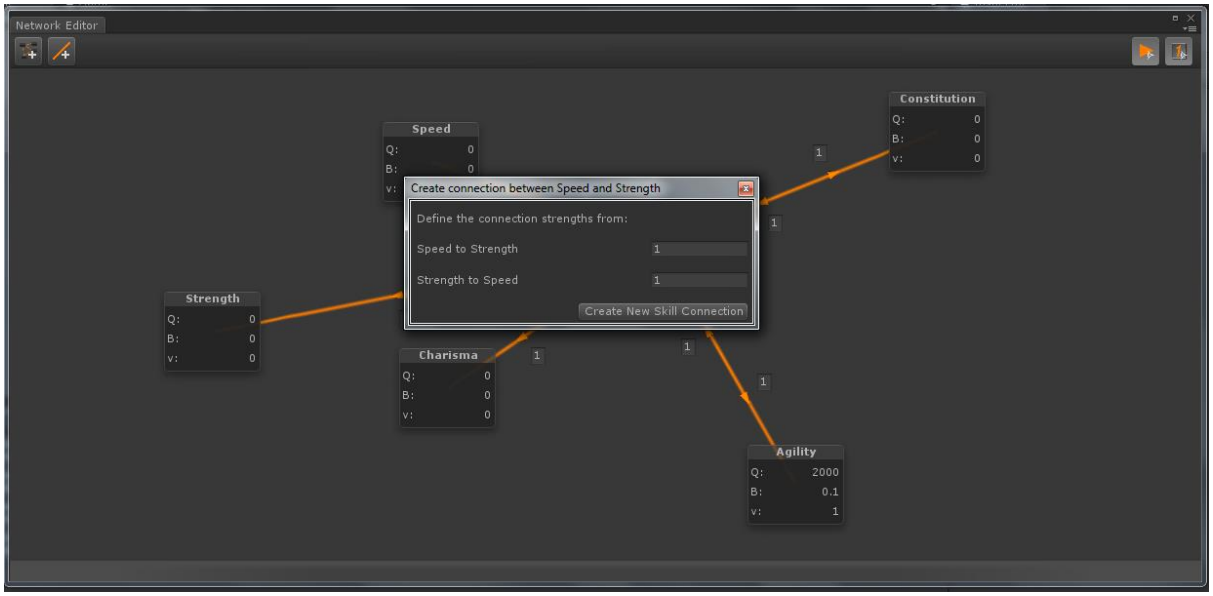
To create a **selection box**, left click anywhere in the Network Canvas except for on a skill, and drag the mouse.

To move several skills at once, shift click on all the selections you wish to make or select them using a selection box, click and hold on one of your selected skills and drag the mouse.

Connecting Skills

Now let's create a **connection** between our two skills. Recall that a connection between two skills means that at least one skill has a positive effect on the other. Mathematically this means that if skill A is connected to skill B, then an increase in the level of skill A will have a positive effect on skill B.

There are several ways to create a connection in the Network Editor. First of all, make sure you have no skills selected by clicking on an empty part of the skill canvas. Click the **Create New Connection** button or use the keyboard shortcut 'ctrl+D', and click on one skill. A line will now be drawn with its base at the centre of the skill you clicked on and its pointer end stuck to the mouse. Click on another skill: a dialog box will appear like the one shown overleaf:



Enter any non-negative values into the fields representing the relationship from skill A to skill B, and from skill B to skill A. The higher the numbers you enter the stronger the relationship between the skills will be.

VERY IMPORTANT NOTE: to ensure convergence of the Action Computational Engine (ACE) (i.e. to get a meaningful result after the network has been updated during gameplay), it is required that the strengths between skills are less that **1 divided by the number of skills in the network and greater than or equal to 0**. However, if this condition is not satisfied, this does not necessarily mean that Action will not produce a result after each calculation: we simply cannot guarantee that you won't get a NotANumberException or extremely long convergence times when running coAdjoint Action in your game.

Press okay in the dialog box. You will now see a directed orange line joining the two skills in your network together. If you entered 0 as one of the strengths, then there will only be one arrow on your line. If you entered non-zero values, then you will see the strengths of the connections floating above the arrows. To turn off the arrows and the values, press the arrows and values toggles in the upper right hand corner.

You can edit an Action connection by clicking on it in the Network Canvas, or by selecting at least one of the skills it connects. The Inspector will contain information about the strength of the connection between the skills. To delete a connection, select a connection and press the **Delete** key on the keyboard, or press the **Delete Connection** button in the Inspector.

You now have all the knowledge you need to work with the Action Editor for Unity! Create as many skills and connections as you like - you may want to use the **duplicate** button in the Library Explorer to clone skills.
To use it, highlight a skill in the Library Explorer and press duplicate. Select it in the Network Editor and change the skill's name and parameters in the Inspector.

After you have created some more skills and connections, let's duplicate the SharedNetwork network in the Library Explorer and add some new skills to it to create the **AltEnemyNetwork** Action network mentioned earlier in this section. In the Library Explorer, highlight the SharedNetwork network and press **duplicate**. Rename it using the **rename** button. Highlight it in the Library Explorer: the network will now be visible in the Network Editor. Edit and add as many (or as few) skills as you want in your new network until you feel comfortable with using the Network Editor.

Using an Action Library at Runtime

You are now ready to build your first runtime library for use in your games. In the Project window, navigate to the **Libraries** folder. Highlight the **BusyBotsLibrary** asset, and click **Generate Runtime Library from Asset** in the Inspector. After a short time, a .NET assembly will be created in the **Builds** folder of your project (if the runtime library does not appear in the Builds folder, right click in the Project window and press **Refresh**). This assembly contains all the metadata about your network but is compiled for use in your scripts.

NOTE: before building your library, it is highly recommended that you check over your networks for repeated names and skills with zero parameters. Such skills are likely to cause a NotFiniteNumberException when used in your scripts.

Let us test out your runtime library in a scene. Create a new scene and create a cube in this scene. Create a new C#, Javascript or Boo script and place it on the cube we just created. Open up the script in your preferred Integrated Development Environment or text editor.

Using your library in a script couldn't be simpler. As with any assembly in Unity, we must include the name of the assembly at the beginning of the script. Following the BusyBots example, type '`using BusyBotsLibrary;`' at the top of your script (see the Unity API reference to find out the appropriate code for UnityScript and Boo). We can now instantiate our networks. The runtime library contains classes with exactly the same names as our networks that we built in the Action Editor, which we can instantiate (in C#) by typing

```
SharedNetwork mySharedNetwork = new SharedNetwork();
```

The example above used the default constructor: for overloaded constructors see the API reference in section 4. In the default `Monobehaviour.Start()` method, instantiate one SharedNetwork network and one VarEnemyNetwork network using the default constructors for each class.

Let us now provide some experience to one skill in each network. The method to do this is called `LevelUp`, and takes an enumerator of the skills in the network in its first argument and a double value representing the desired increase in experience.

For example, if SharedNetwork contained a skill called Speed, to add 3 experience points to the speed skill we would write

```
mySharedNetwork.LevelUp(SharedNetwork.Skills.Speed, 3.0);
```

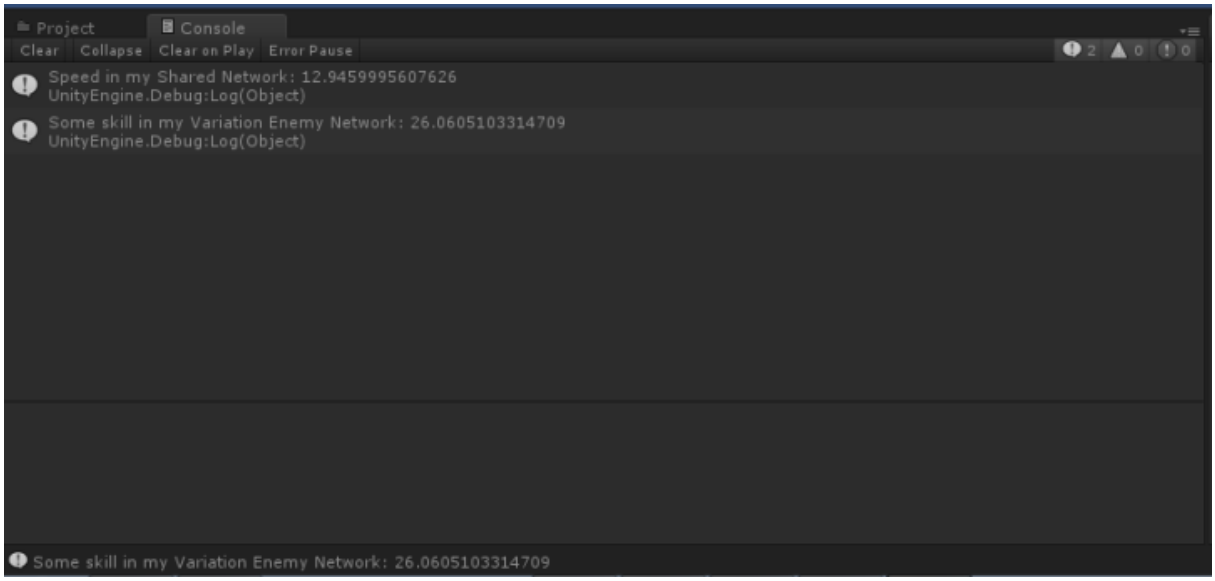
Again, in `Monobehaviour.Start()`, add some experience to a skill in your instance of SharedNetwork and to a skill in the instance of VarEnemyNetwork.

To find the level of each skill, we shall use the `GetLevelAsDouble` method. `GetLevelAsDouble` takes a skill enumerator as its only argument and returns a double level of the skill. In the `Monobehaviour.Update()` method, use `Debug.Log` to print out the levels of the skills you've updated to the console. If you're using C#, your script should look something like this;

```
using UnityEngine;
using BusyBotsLibraryBuild;

public class Test : MonoBehaviour {


    // Use this for initialization
    void Start () {


        SharedNetwork mySharedNetwork = new SharedNetwork();

        VarEnemyNetwork myVarEnemyNetwork = new
          VarEnemyNetwork();
```

```
            mySharedNetwork.LevelUp(SharedNetwork.Skill.Speed,
            3.0);

        myVarEnemyNetwork.LevelUp
          (VarEnemyNetwork.Skill.SomeSkill,3.0);
    }

    void Update () {

        if(Input.anyKeyDown())
        {
            Debug.Log("Speed in my Shared Network: " +
                  mySharedNetwork.GetLevelAsDouble
                  (SharedNetwork.Skill.Speed));

            Debug.Log("Some Skill in var Enemy Network: " +
                  myVarEnemyNetwork.GetLevelAsDouble
                       (VarEnemyNetwork.Skill.SomeSkill));
        }
    }
}
```

Run your game by pressing play in the Unity Editor. After pressing any key, your console should look something like the following image:



If it does, congratulations! If it doesn't, go back to the Action Editor, check over each skill and make sure each skill has non-zero parameters.

Feel free to experiment with levelling up connected skills to see how connections change the level up process of skills.

Note: when running your scene, a new game object is created in the hierarchy. This object, called '_IntAct', is a wrapper for the underlying mathematics of coAdjoint Action known as the Action Computational Engine (ACE), and will be created automatically when you instantiate and use a runtime network.

Note: the Action Computational Engine uses multithreading. This means that if you call the `LevelUp` method and check the level in the same frame it's unlikely to have updated, since the calculation may not have completed.

Note: There are no restrictions on the number of networks you can include in a library. However, using multiple Action libraries in a project is not recommended and is currently untested. Instead, if you need to add extra Networks to an existing Library, use the Action Editor to add to the existing Action library and generate a new runtime library to use in your game.

Reference API

This section contains information about the methods you can call on a network contained inside a runtime library assembly. For this section a runtime Action network will be called NetworkName.

Constructors

Each Action network class contained in a runtime library assembly has 3 constructors:

public NetworkName(): constructs an instance of NetworkName with all skill levels set to 1.

public NetworkName(double level): constructs an instance of NetworkName with all skill levels initialised to level.

public NetworkName(double[] levels): constructs an instance of NetworkName with skill levels initialised by the levels array. Before using this method, make sure you have examined the Skills enumerator to check the index of each skill in the array.

Very Important Note: When using the overloaded constructors for NetworkName, it is recommended that the levels used to initialise the skills are not higher than 99 or lower than 1, as this may cause an error.

Enumerators

Public enum Skill: enumerator containing the names of all the skills in your network with zero offset.

Methods

public void SetAllLevels(double level): Sets the level of each skill in NetworkName to level.

public void SetLevel(NetworkName.Skill skill, **double** val): sets the skill level of skill to val.

public double GetLevelAsDouble(NetworkName.Skill skill): returns the level of the skill skill as a double.

public float GetLevelAsFloat(NetworkName.Skill skill): returns the level of the skill skill as a float.

public double GetExperienceAsDouble(NetworkName.Skill skill): returns the experience of the skill as a double.

public float GetExperienceAsFloat(NetworkName.Skill skill): returns the experience of the skill as a float.

public void LevelUp(NetworkName.Skill skill, double val): adds experience val to the Skill skill, causing the skill level to increase. **Note:** when this method returns the level of skill has **not increased.** The skill is put on a queue until the ACE is free to update it.

public void Remove(): Removes all skills in NetworkName from the ACE queue.