

# PuzzLearn User Manual

## Contents

1.	Introduction.....	2
1.1.	About PuzzLearn .....	2
1.2.	About this manual.....	2
1.3.	Downloading the BizHawk emulator.....	2
1.4.	Retrieving ROM files .....	2
2.	Running the Learning Algorithm .....	3
2.1.	How PuzzLearn learns .....	3
2.1.1.	Neural networks .....	3
2.1.2.	The NEAT algorithm.....	4
2.1.3.	PuzzLearn's testing process.....	5
2.2.	Running the Lua script .....	5
2.3.	Creating a New Session.....	7
2.3.1.	The neural network display form .....	8
2.4.	Loading a session.....	10
3.	The Configuration File Builder.....	11
3.1.	Basic information .....	11
3.1.1.	Hexadecimal.....	11
3.1.2.	Bytes.....	11
3.1.3.	Number addresses.....	12
3.1.4.	Information addresses.....	13
3.1.5.	Using BizHawk's RAM search.....	13
3.2.	Structure information .....	16
3.2.1.	The main menu .....	16
3.2.2.	Planes .....	17
3.2.3.	Objects.....	19
3.2.4.	Regions .....	20
3.2.5.	XY regions .....	21
3.2.6.	Database settings.....	23
3.2.7.	Putting it together .....	25
3.3.	Online functionality .....	26
3.3.1.	Downloading files .....	26
3.3.2.	Managing your files.....	27

## 1. Introduction

This section includes some basic prerequisite information regarding PuzzLearn and this manual. It is recommended that you read this before moving on to other sections.

### 1.1. About PuzzLearn

PuzzLearn is a simple neural network building program that learns how to play video game. It was specifically designed with SNES puzzle games in mind, but can work with games from other consoles or genres. It consists of two major components:

- A Lua learning script, which works with the BizHawk emulator and uses the neuro-evolution of augmenting topologies (NEAT) algorithm to learn how to play video games.
- A configuration file builder, which can be used to create configuration files that are used by the learning script to determine how to parse the emulator's memory when playing a game.

### 1.2. About this manual

This manual is intended to document and explain all of PuzzLearn's features and how to use them. It contains all information necessary to both run PuzzLearn's learning script and create configuration files. With this manual, any PuzzLearn user with the necessary resources should be able to use the configuration builder to configure a game and run the learning algorithm. It is intended to be read before or while using PuzzLearn.

### 1.3. Downloading the BizHawk emulator

PuzzLearn's learning script works with the BizHawk emulator, a multi-console video game emulator with several advanced features for enthusiast use. BizHawk is available for download at <http://tasvideos.org/BizHawk/ReleaseHistory.html>. Windows users must additionally download and run the prerequisite installer available at the top of the page.

PuzzLearn was tested using BizHawk version 2.3.1. Earlier versions may not be able to run PuzzLearn.

### 1.4. Retrieving ROM files

PuzzLearn will not work if the emulator does not have a file describing a video game's read only memory, or ROM. In order to legally download a ROM onto your computer, you must obtain a copy of the video game you intend to use and a video game backup device compatible with the game's cartridge or disc. Follow the backup device's instructions to download the game onto your computer.

As an alternative, certain homebrew video games are available as free downloads online. While fewer options are available, they can be a time- and cost-effective alternative for those who do not already own a backup device.

## 2. Running the Learning Algorithm

This section describes how to begin running the learning algorithm and how to use its features.

### 2.1. How PuzzLearn learns

While not required in order to use PuzzLearn effectively, understanding how PuzzLearn works will help you parse what the learning script displays and get more out of PuzzLearn. This subsection briefly describes PuzzLearn's learning algorithm.

#### 2.1.1. Neural networks

PuzzLearn creates evolving neural networks in order to learn and improve. A neural network contains the following components:

- **Nodes:** Nodes are used to hold values as they move from the input to the output. Types of nodes include:
  - **Input nodes:** These describe the input supplied to the network. PuzzLearn's input nodes hold either a 0 or a 1, but other networks can use different values.
  - **Hidden nodes:** These are the internal nodes of the neural network. They are used to hold intermediary values and help describe the internal logic of the network.
  - **Output nodes:** These describe the output produced by the network. The values of the output nodes are determined by the network and their meaning is typically parsed by some other part of the program. PuzzLearn's output nodes correspond to what buttons should be pressed during a given frame, where a value greater than 0 indicates that the button should be pressed and any other value indicates that it should not.
- **Links:** Links connect one node to another, propagating one node's value up to another node. Each link has a weight, which describes what effect the input node of the link should have on its output node.

Note that the terminology for neural networks is not set in stone – nodes are often called neurons and links are often called edges, referencing neurological concepts that inspired neural networks. For simplicity's sake, this manual will refer to them as nodes and links. In a typical neural network, the structure of the network – or how the nodes and links are connected together – remains static, while the weights of the links are modified. Figure 1 shows an example of a simple processed neural network.

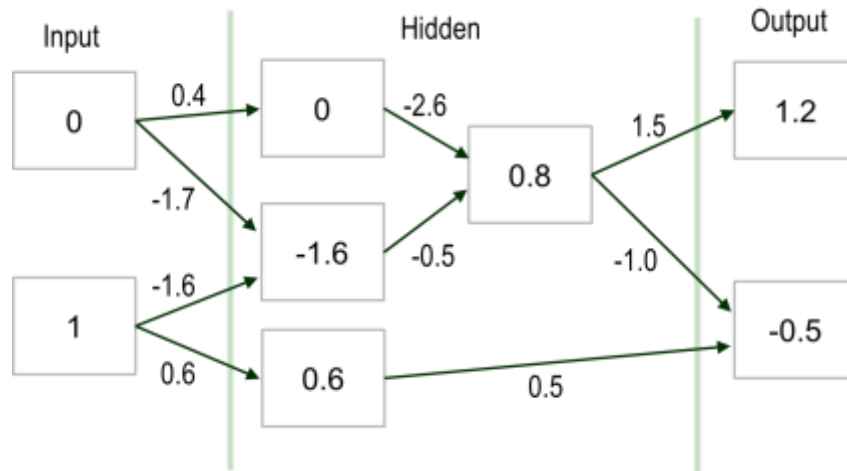


Figure 1: Example of a simple neural network

### 2.1.2. The NEAT algorithm.

PuzzLearn uses the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, which was first described in the paper “Evolving Neural Networks through Augmenting Topologies” by Kenneth O. Stanley and Risto Miikkulainen. It differs from many neural network algorithms in that it modifies the structure of a set of networks as it trains by modifying and combining them through successive generations. The input and output nodes of a NEAT network remain fixed.

The general process for the NEAT algorithm is as follows:

1. For the first generation, create a population of simple neural networks, typically just a few links from the input nodes to the output nodes. In this generation, each network will be part of a different species, indicating that none of them are similar enough to be combined.
2. Test these networks, scoring them in some way and assigning them a fitness value.
3. Once all networks are tested, begin creating a new generation using the following process:
  1. Remove the lowest-scoring networks from each species. (No networks will be removed in the first generation.)
  2. Calculate an adjusted fitness for each network, which is the fitness of the network divided by the number of networks in the same species. (In the first generation, the adjusted fitness will equal the fitness of the single network.)
  3. Find the total adjusted fitness of each species by adding up the adjusted fitness of each network in the species. In the next generation, the number of networks belonging to a certain species will be equal to the population size times the total adjusted fitness of the species divided by the sum of total adjusted fitnesses for all species.
  4. Remove species with low enough total adjusted fitnesses that they will not have any networks in the next generation.

5. Populate the new generation with an appropriate number of children from each species, which are created by randomly modifying or combining randomly networks from the species. Modifications include adding a link between nodes, adding a node to the network, and disabling or enabling a link (rendering it inactive or active in the next test).
  6. For each species from the previous generation, create a new species for the new generation based on a randomly selected network from the old species.
  7. Sort the children networks into these species based on their similarity to their base networks. If a network is different enough from all base networks, it will instead be used as the basis for a new species.
4. Once the new generation is complete, repeat the process from step 2.

In order to track the history of each modification, each link in a NEAT network contains an innovation number, which represents when the link was first created. Innovation numbers are used in two contexts:

- When combining networks, innovation numbers prevent two links with the same origin from being added to the child network twice
- Innovations are key in determining whether two networks are part of the same species, with the same-species threshold determined by the amount of links with a different origin and the difference in weights for links with a shared origin.

### 2.1.3. PuzzLearn's testing process

PuzzLearn uses the NEAT algorithm as described in order to generate neural networks. In order to test an individual network, it does the following:

1. Load a starting game state provided by the user.
2. For each frame, process the neural network to retrieve button outputs and supply them to the emulator.
3. Continue until a timeout period ends or the game reaches an end state.

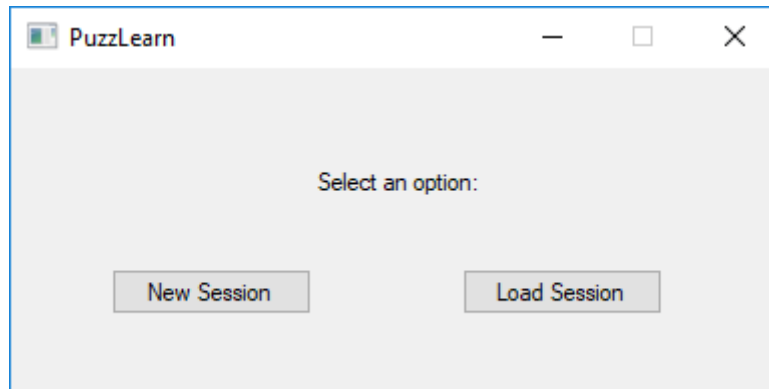
## 2.2. Running the Lua script

To begin running the Lua script, do the following:

1. Move or copy the contents of the Lua folder included with PuzzLearn (the PuzzLearn.lua file and the PuzzLearnData folder) to the Lua subfolder within the BizHawk emulator folder.
2. From the folder you installed BizHawk, run the EmuHawk.exe executable. This will open BizHawk.
3. From the main task bar, select Tools > Lua console. This will open the Lua console in a new window.

4. In the newly opened Lua console, click on the open folder icon directly underneath the file taskbar item. This will open an open file dialog in BizHawk's Lua folder.
5. Select and open the PuzzLearn.lua file that you placed in the folder earlier.

After completing these steps, the PuzzLearn main menu window should appear, as shown in figure 2.



*Figure 2: The PuzzLearn main menu*

Before proceeding with either of the two options provided, you must open the video game ROM that you wish to use with PuzzLearn by clicking File > Open Rom in BizHawk's main window, then navigating to and selecting the desired ROM in the newly opened dialog. If you have opened that ROM recently, you may also open it more quickly by selecting File > Recent ROMs.

### 2.3. Creating a New Session

By clicking the “New Session” button, you will proceed to the form shown in figure 3.

The image shows a Windows-style dialog box titled "PuzzLearn - New Session". It has a standard title bar with minimize, maximize, and close buttons. The dialog contains three rows of controls:

- The first row has the label "Configuration: (Not specified)" followed by a "Select" button.
- The second row has the label "State: (Not specified)" followed by a "Select" button.
- The third row has the label "File name:" followed by a single-line text input box.

At the bottom of the dialog, there are two buttons: "Confirm" on the left and "Cancel" on the right.

Figure 3: The new session form

In order to begin a session, you must enter the following data:

- Configuration:** Click on the top select button to open a file dialogue requesting a PuzzLearn configuration file (.plcf). Some preconfigured ones for certain games are included in the PuzzLearnData\Config folder. If you need a different file, use the configuration file builder to check if there are any available for download for the game you wish to play (see [section 3.3.1](#) for more information). Otherwise, you will need to create your own configuration file. Consult [section 3](#) for guidance on this.
- State:** To create a BizHawk save state (.state) that PuzzLearn will use as a starting point for each learning iteration, navigate through the game’s menus until you reach a point where you begin actually playing the game, then click on File > Save State > Save Named State in the emulator. It is recommended, but not required, that you save any states intended to be used with PuzzLearn in the PuzzLearnData\States folder, in order to prevent accidental modification of them that may interfere with the learning process. Figure 4 shows an example of a good starting state. Once your state is created, click on the middle select button and select your state.
- File name:** Enter text in the text box to determine what the session will be called. This file will be saved in the same folder as the PuzzLearn script, but can be moved later.



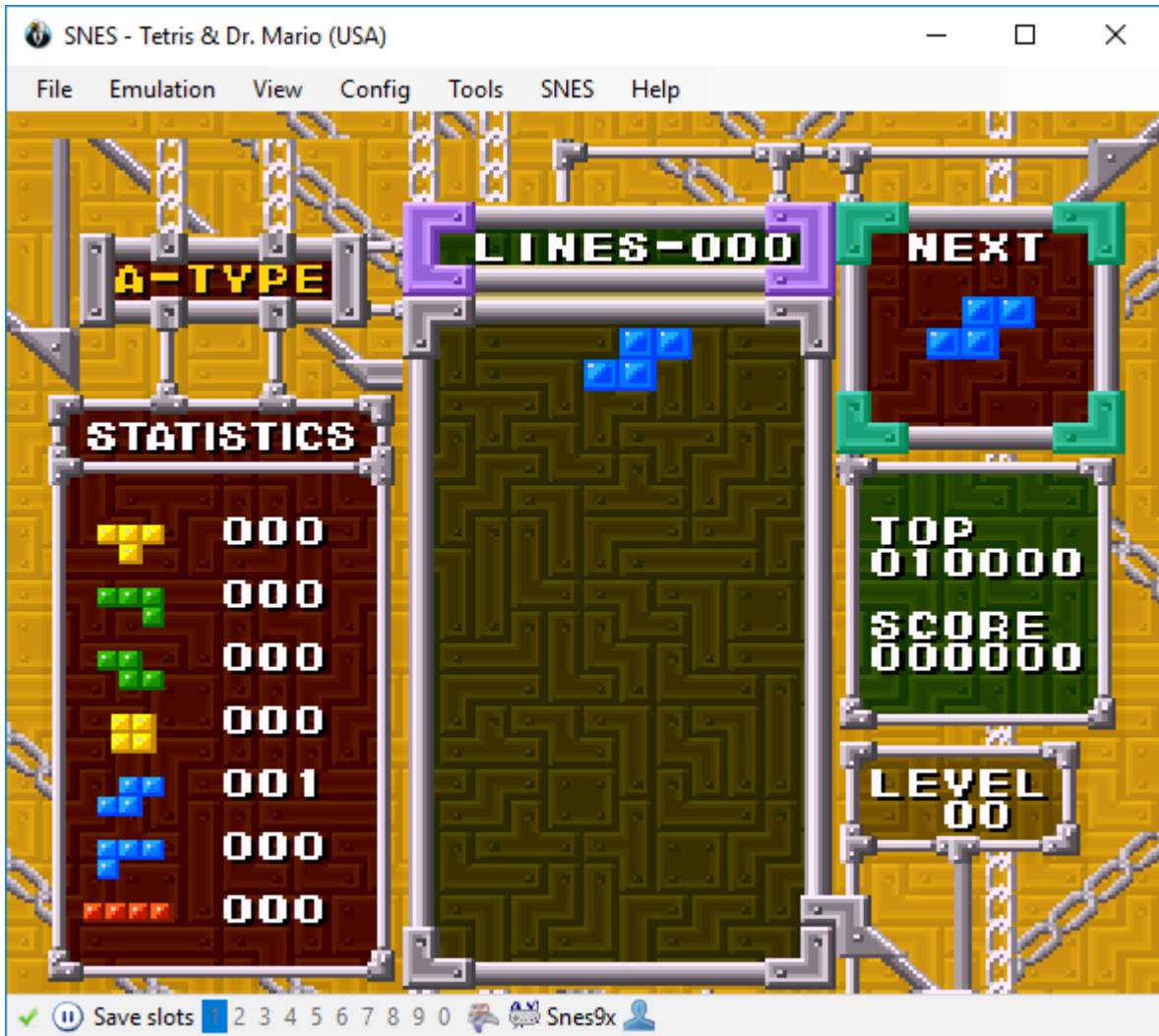


Figure 4: Example of a good starting state

Once this information is entered, click the “Confirm” button. PuzzLearn will then create a PuzzLearn session file (.pls), begin the learning algorithm, and proceed to the network display window.

### 2.3.1. The neural network display form

While the learning algorithm is running, a display form will show a visual representation of the neural network, showing its inputs, middle logic-defining nodes, and outputs. Figure 5 shows example window. Different configuration files will result in slightly different windows.

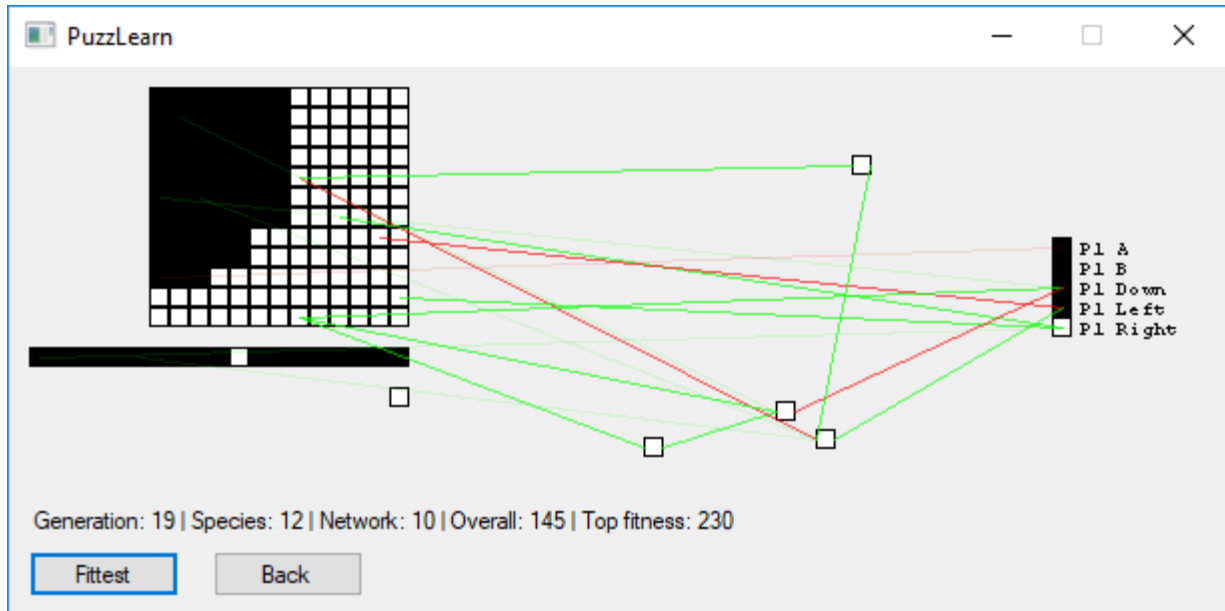


Figure 5: Neural network display window while playing Tetris

Elements of the display window include:

- **Left: Inputs.** This is a visualization of the inputs that are being given to the neural network, effectively what it “sees” as it runs through each genome. Inputs can be split into three types:
  - **Top: Address planes.** This is a 2D coordinate table where each cell stores a specific value based on information from the configuration file builder. The resulting window should mirror what the emulator is currently doing. Certain address planes will use several different colors, where each one represents a different type of value in the cell. Figure 5 uses white for “contains a block” and black for “empty.” A configuration file is not limited to one plane, but can have 0 or more depending on the game. For more details about address planes and the structures within them, go to [section 3.2.2](#).
  - **Middle: Information addresses.** Represents a piece of miscellaneous information read from an address in BizHawk’s memory, where each cell in the row represents a specific nonnumerical value within the game state. Like planes, there can be 0 or more information addresses in a configuration file. For more detailed information about information addresses and their contents, see [section 3.1.4](#).
  - **Bottom: Bias node.** Regardless of the contents of the configuration file, there will be a single node at the bottom of the inputs that is always on. This is used to bias the network, making certain hidden and output nodes more likely to be on or off, regardless of the game’s state.
- **Middle: Hidden nodes.** These nodes and the connections between them represent the logic of the neural network being tested. Some notes on hidden nodes:

- Black, translucent squares are off and negative or zero-valued, while white, opaque squares are on and positive.
- Lines at the left side of a node are inward connections that modify that node's value. Lines at the right are outward connections that propagate its value to other nodes.
- Translucent lines represent inactive connections from off nodes, while opaque ones are active from on nodes.
- Green lines represent positive connections, which will result in a positive output from a positive node and a negative output from a negative node. Red lines represent negative connections, which will give a negative output from a positive node and a positive output from a negative node.
- **Right: Output buttons.** These nodes represent the buttons that are being pressed in a given frame. White nodes are pressed, black nodes are not. The buttons that can be pressed are determined by the configuration file.
- **Bottom: Navigation options.** Clicking the back button will stop the learning algorithm and return to the main menu. Clicking the fittest button will stop the current genome, load and run the highest scoring genome generated by the network, then reload the current genome.

## 2.4. Loading a session

In order to load a session, click the “Load Session” button on the main menu. This will open a file dialogue asking for a PuzzLearn session file (.pls). Here you can select and load a premade session. If it can be opened successfully, PuzzLearn will continue learning from where that session left off .

### 3. The Configuration File Builder

This section describes the elements of the configuration file builder.

#### 3.1. Basic information

This subsection describes some basic concepts and structures used in PuzzLearn, then explains how to find addresses using BizHawk. It is recommended that you make sure that you understand these before moving on to the rest of this section.

##### 3.1.1. Hexadecimal

BizHawk uses hexadecimal values to represent memory addresses in the emulator. For consistency, addresses in PuzzLearn are also represented in hexadecimal. Hexadecimal is a base 16 numbering system. While 1 through 9 are the same in decimal (base 10) and hexadecimal, hexadecimal then continues such that A = 10, B = 11, C = 12, D = 13, E = 14, F = 15, and 10 = 16. Table 1 describes more hexadecimal numbers and how to interpret them.

Hexadecimal	Interpretation	Decimal
10	$1 * 16 + 0 * 1$	16
14	$1 * 16 + 4 * 1$	20
3A	$3 * 16 + 10 * 1$	58
FF	$15 * 16 + 15 * 1$	255
100	$1 * 16^2 + 0 * 16 + 0 * 1$	256
2FB	$2 * 16^2 + 15 * 16 + 11 * 1$	763

*Table 1: Hexadecimal interpretation*

While this information is useful for understanding how addresses in memory relate to one another, particularly when several nearby addresses have related information, translating addresses from hexadecimal to decimal is rarely necessary.

##### 3.1.2. Bytes

A byte is the basic size held within a unit of memory. It consists of 8 binary bits and can hold a value from 0 to 255 (or, in hexadecimal, FF) for a total of 256 possible values. When reading memory from the emulator, PuzzLearn reads a single byte held at the specified address.

### 3.1.3. Number addresses

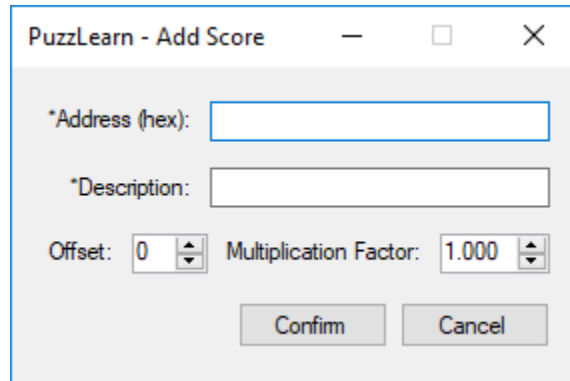


Figure 6: A number address form

The number address structure reads the byte stored at an address and interprets it as a number. This number is then modified according to equation 1.

$$\text{Final Value} = (\text{Value} + \text{Offset}) * \text{Multiplication Factor}$$

Equation 1: Calculating a numerical value

For example, a read value of 58 in a structure with an offset of -2 and a multiplication factor of 0.25 will result in a final value of 14. The default offset of 0 and multiplication factor of 1 will result in an unchanged number.

Not unique to the number address structure are the address, a hexadecimal value representing what memory address this structure should evaluate, and the description, which is stored more for the user's benefit to summarize what this structure is and what it does. Additionally, most forms share the "Confirm" and "Cancel" buttons. Confirm will confirm all changes and add the configured structure. Cancel will remove all changes and leave the base structure unchanged.

Within PuzzLearn's structures, numerical addresses are stored in groups. This is because it sometimes takes more than one byte to store a number; a score going up to 1,000,000 obviously has more than 256 values and requires more than one address to store. To handle this situation, PuzzLearn groups numbers together, adds all of them up, and uses the sum. In the score example, the first address may be multiplied by 1, the next multiplied by 256 (100 in hexadecimal), the next by 65,536 (10,000 in hexadecimal), and so on, which will produce a value equal to the displayed score.

### 3.1.4. Information addresses

The screenshot shows a window titled "PuzzLearn - Add Information". It has a standard Windows-style title bar with minimize, maximize, and close buttons. Inside the window, there are several input fields and a table. At the top, there is a field labeled "\*Address (hex):" followed by an empty text box. Below that is a field labeled "\*Description:" followed by an empty text box. In the center, there is a table with two columns: "Address Value" and "Category". The table body is currently empty. To the right of the table, there are three spinners: "Address value:" with a value of 0, "Category:" with a value of 0, and "Default Value:" with a value of 0. Below these spinners are three buttons: "Add/Update", "Delete", and "Default Value:". At the bottom of the window are two buttons: "Confirm" and "Cancel".

Figure 7: The information address form

The information address structure is a way of parsing nonnumerical data from memory; for example, what shape and orientation the current block is in Tetris. It does this with a table of address values and corresponding categories. Each category corresponds to a different specific game state, with the 0 category reserved for a null state giving no input. Each address value can belong to only one category, but a category can match multiple values. If the value stored in memory does not correspond to any value in the table, it will instead use the default value.

Information addresses and their categories exist in different contexts, which are described in more detail in sections [3.2.1](#) and [3.2.2](#). It is expected that the categories within each context start at 1 and successively increase by 1. Failure to do this – for example, by using categories 1 and 3 but not 2, or 2 and 3 but not 1 – will result in the final configuration file generating an input array with nodes that will never be called. This will not prevent the program from running, but will slow down the learning process.

### 3.1.5. Using BizHawk's RAM search

In order to retrieve addresses for these and other structures, you must use BizHawk's RAM search, accessible from Tools > RAM Search.

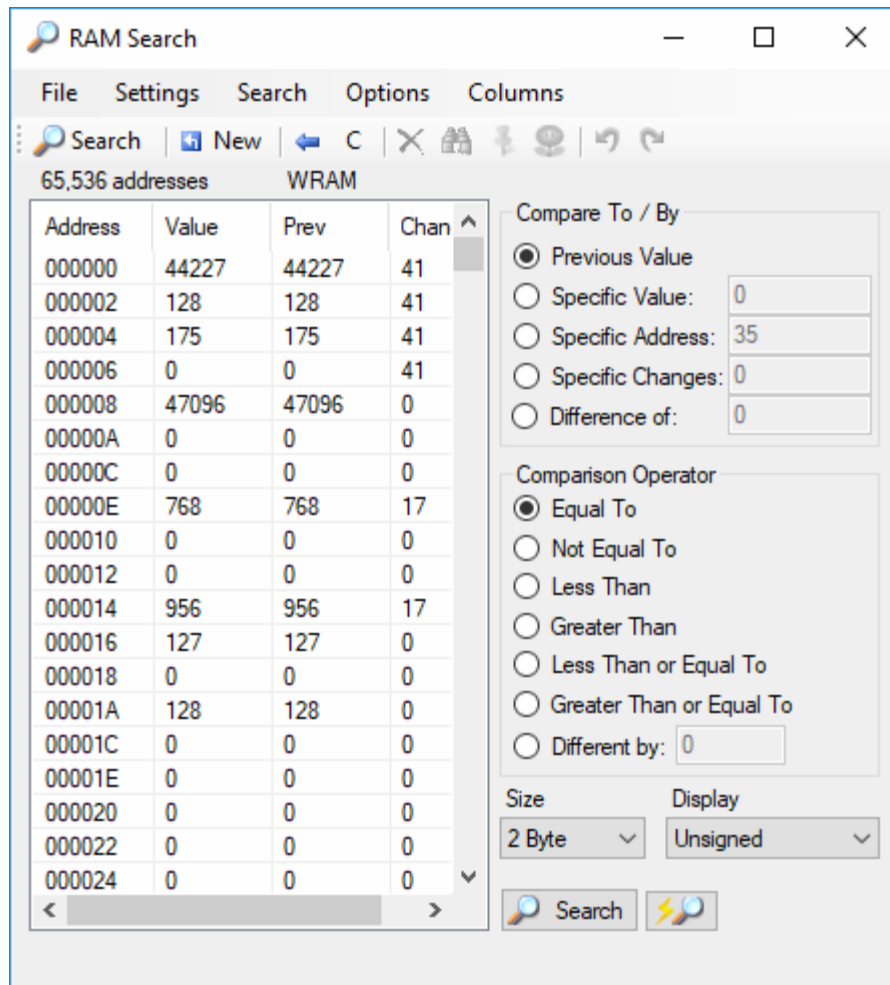


Figure 8: The RAM Search window

If necessary, change the size to 1 byte, as PuzzLearn reads memory 1 byte at a time. The general process for finding the address holding a certain is as follows:

1. Determine what value you are searching for – for example, the X position of the game's cursor or the value of a specific square in the game board.
2. Create a starting state in which it will be easy to change that value.
3. Click New in the RAM search to start a new search.
4. Modify the value, select a comparison operator in the RAM search that matches how you expect the value to change, and click the Search button to filter out values that do not match the condition. For example, an X position often decreases as it moves left, so selecting "Less Than," moving to the left, and clicking search should leave the X position while filtering out others. Note leaving the value unchanged and selecting the "Equal To" operator will also filter out unwanted addresses. During this step, it may be useful to pause emulation by pressing the pause key on your keyboard or selecting Emulation > Pause.

5. Repeat step 4 several times until only a few addresses are left. If none of them remain, start over from step 3, possibly modifying your process.
6. While looking at the values in the RAM search, modify the value and see if any of them seem to change in the way that you expect. If none of them do, start over from step 3.
7. Once you determine one or more addresses that seem to represent the desired variable, double click on them to add them to a RAM watch.

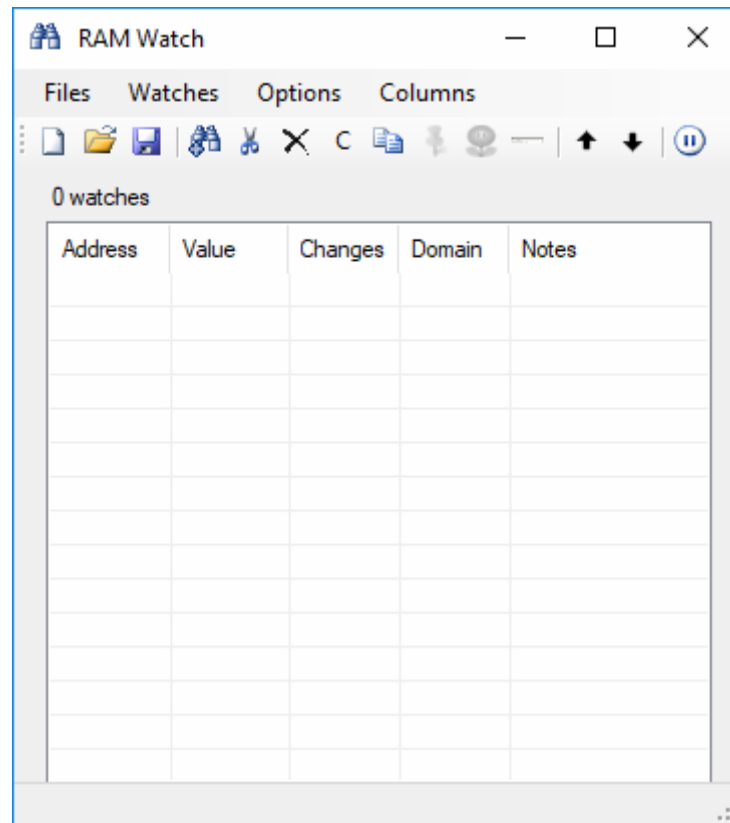


Figure 9: The RAM Watch window

8. In order to further test this address or these addresses, right click one of them and select “Freeze Address.” This will forcibly prevent the address’s value from changing. See if the emulated game reacts as you would expect. Repeat this for any addresses until you find the most likely candidate
9. As a final test, right click and select “Poke,” which will allow you to forcibly change this value. Change it to a value that the address has previously held. See if this has the desired effect.
10. Once you are certain what address stores the desired value, double click the address in the RAM watch and add a descriptive note.
11. Remove any unnecessary addresses from the RAM watch.



12. Save the RAM watch by pressing Ctrl + S or selecting File > Save. This will record the address and its description for future use.

### 3.2. Structure information

This section describes what the remaining memory structures and how they, as well as the basic ones discussed earlier, fit together to create a neural network input.

#### 3.2.1. The main menu

The PuzzLearn main menu provides a starting point for creating configuration files, containing the topmost, general structures and organization.

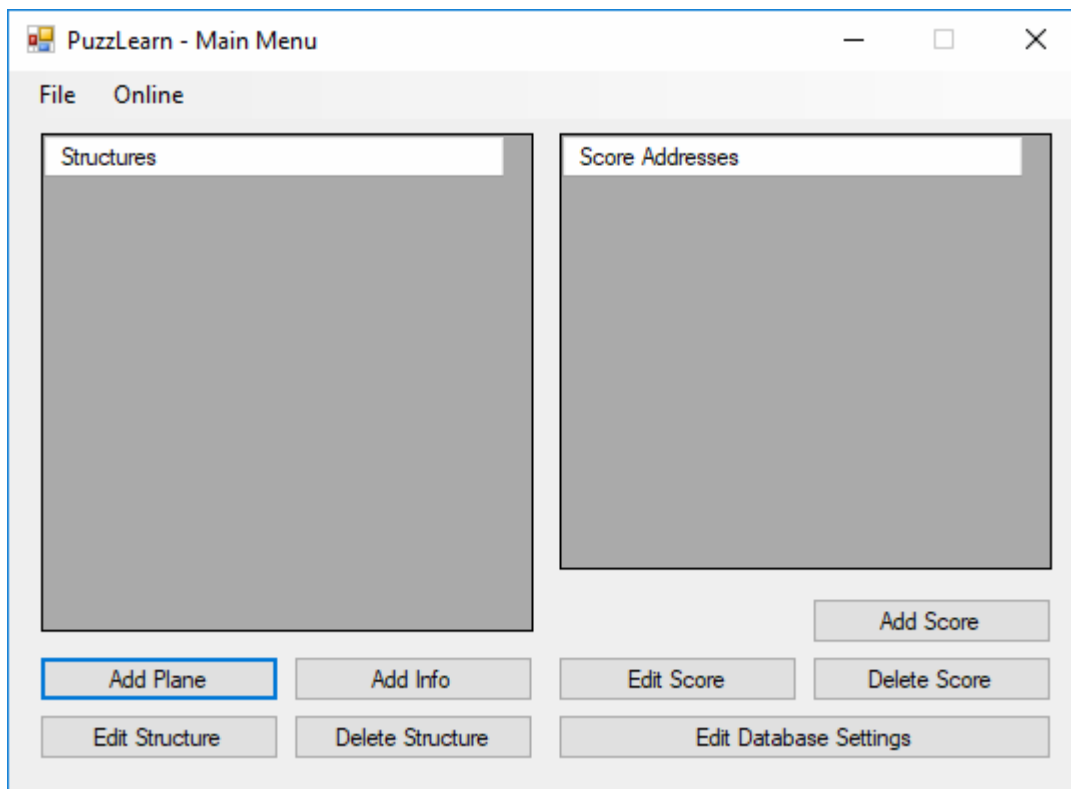


Figure 10: The configuration file builder's main menu

The main menu provides access to the following:

- **Structures:** These are the memory structures that PuzzLearn will evaluate. Each one will produce its own output array of 0s and 1s and is used separately to produce a visualization of the network. Related buttons include:
  - **Add Plane:** Opens a form to add a plane to the list of structures. Planes are described in [section 3.2.2](#).

- **Add Info:** Opens a form to add a miscellaneous information address to the list of structures, which is not related to any specific plane but contains useful information for the learning script. An example information address that would go here is the current Tetris block's shape and orientation. Each information address in the list of structures exists in an independent context, and does not need to share values with other addresses.
- **Edit Struct:** Opens a form to edit the values of the currently selected structure. Select a structure by clicking on it
- **Delete Struct:** Removes the currently selected structure from the list.
- **Score addresses:** This is a list of number addresses that will be summed together to produce a score value, which will be used to determine how successful a neural network is after it has been tested. Related buttons include:
  - **Add Score:** Opens a form to add a number address to the list of score addresses.
  - **Edit Score:** Opens a form to edit the currently selected score address.
  - **Delete Score:** Removes the currently selected score address from the list.
- **Edit Database Settings:** This button opens a form containing miscellaneous database settings, which are discussed in [section 3.2.6](#).
- **File:** Contains options for starting a new file, saving the file, saving the file under a different name, and opening a file.
- **Online:** Contains options for two online features, described in [section 3.3](#).

### 3.2.2. Planes

The plane structure defines a 2-dimensional table of values, read and interpreted from BizHawk's memory through the contained structures, that is converted to a series of inputs for the neural network function. The values stored in this table correspond to categories provided by information addresses and other structures. All categories within the plane exist in a shared context, which shares the rules described in [section 3.1.4](#). Note that it is the greater plane context, not the individual structures, that must follow these rules; if one structure uses categories 1 and 3 while another uses categories 2 and 3, the entire context of the plane will have values 1, 2, and 3, satisfying the conditions for a successful configuration file.

The screenshot shows a window titled "PuzzLearn - Add Plane". It contains three list boxes on the left: "Structures", "X Addresses", and "Y Addresses". To the right of each list box are buttons for "Add", "Edit", and "Delete". The "Add Object" button is highlighted with a blue border. Below the list boxes are input fields for "Min:" and "Max:" with spinners, and a "Default Value:" field with a spinner. At the bottom are "Confirm" and "Cancel" buttons.

Figure 11: The plane form

Options in the plane form include:

- **Structures:** A list of PuzzLearn structures contained by the plane, which are used to determine the values in its tables. Related settings include:
  - **Add Object/Region/XY Region:** Opens a form to add the corresponding structure to the structures list, as described in the following sections.
  - **Edit:** Opens a form to edit the selected structure.
  - **Delete:** Deletes the selected structure.
- **Width/X Addresses:** Contains information defining the left and right bounds of the plane. Related settings include:
  - **Min:** Defines the minimum X position, or left bound, of the plane.
  - **Max:** Defines the maximum X position, or right bound, of the plane.
  - **Add:** Opens a number address form that is used to define the center of plane. If the X addresses list contains any elements, all addresses within it will have their values summed, which will be used to define the relative center of the plane, and the minimum and maximum values will be relative to this value for each frame. For example, if the minimum is -5, the maximum is 5, and the current sum of X

addresses is 3, the plane will accept X positions from -2 to 8. If no values are in the X address list (equivalent to a constant sum of 0), the minimum and maximum will be static.

- **Edit:** Opens a form to edit the selected X address.
- **Delete:** Deletes the selected X address.
- **Height/Y Addresses:** Contains information defining the top and bottom bounds of the plane. All related settings are equivalent to settings for the X addresses, with the sole difference being that they are related to Y positions instead.
- **Default Value:** Defines the default value held by a cell in the table in the event that none of the structures within correspond to the given X and Y position. This value is included in the category context of the plane.

### 3.2.3. Objects

The screenshot shows a window titled "PuzzLearn - Add Object". Inside, there are two large list boxes labeled "X Addresses" and "Y Addresses". To the right of each list box are three buttons: "Add", "Edit", and "Delete". Below these list boxes is a section labeled "Information Address" with a text box containing "(Not available)" and two buttons: "Add/Edit" and "Delete". At the bottom of the window, there is a "Fixed Value:" label next to a spinner box showing the number "0". Below that is a text box labeled "\*Description:". At the very bottom are two buttons: "Confirm" and "Cancel".

Figure 12: The object form

The object structure defines an object within the game that can move around and with a game state that is either static or dynamic. Options in the object form include:

- **X Addresses:** The sum of this list of number addresses defines the X position of the object. As in other cases, items can be added, edited, and deleted.
- **Y Addresses:** The sum of this list of number addresses defines the Y position of the object.
- **Information Address and Fixed Value:** If an information address is defined, the category read by that information address will be applied to this object. Otherwise, the fixed value will be the constant value of the object.

When reading the object, the coordinate in the plane that the object occupies will hold either the value of the information address's category or the fixed value.

### 3.2.4. Regions

Figure 13: The region form

The region structure defines a series of structures that repeats several times in a predictable pattern. This saves time having to define each one individually. Options in the region form include:

- **Structures:** Defines the first set of structures that will be repeated for a set period. Structures can be objects or other regions. As with other lists, items can be added, edited, and deleted.

- **Start Address:** This defines where the region starts, and should match the first address of the first set of structures.
- **End Address:** This defines where the region ends, and should match the first address of the last set of structures.
- **Address Increment:** This defines how many addresses apart each set of structures is.

To illustrate how regions are defined, suppose a game has an object with an information address at 0100, an X address at 0101, and a Y address at 0102. Another object has information at 0104 (4 addresses ahead of the first one), X at 0105, and Y at 0106. This pattern repeats until a last object at 013C, 013D, and 013E, after addresses stop following this pattern. In this case, the region structure should define the first object, have a start address of 0100, an end address of 013C, and an address increment of 4.

### 3.2.5. XY regions

The screenshot shows a dialog box titled "PuzzLearn - Add XY Region". It contains the following elements:

- \*Start Address (hex): [text input field]
- \*Description: [text input field]
- Width: 1 [spin button]
- Height: 1 [spin button]
- Row Offset: 1 [spin button]
- X Start: 0 [spin button]
- Y Start: 0 [spin button]
- A table with two columns: "Address Value" and "Category". The table body is currently empty.
- Address value: 0 [spin button]
- Category: 0 [spin button]
- Add/Update [button]
- Delete [button]
- Default Value: 0 [spin button]
- Confirm [button]
- Cancel [button]

Figure 14: The XY region form

The XY region defines a block of information addresses where each one corresponds to the state of a specific XY position within the plane. For example, the game board in Tetris has a large block of addresses determining what, if anything, should occupy each square within the area that blocks are dropped. Options in the XY region form are as follows:

- **Start Address:** The first address in the XY region.

- **Width:** The length, or X dimension, of the XY region.
- **Height:** The height, or Y dimension, of the XY region.
- **Row Offset:** How many addresses are between each row. This is used in the case where some addresses should be ignored. For example, if each row is 8 addresses apart but only the first 6 matter, the width should be 6 and the row offset should be 8.
- **X Start:** This is the X position that the top left address corresponds to. Each successive address will have an X position 1 higher than the last until the row ends, after which it resets.
- **Y Start:** This is the Y position that the top left address corresponds to. Each new row will have a Y position 1 higher than the previous row.
- **Address Values and Categories:** These define what category an individual cell of the table should hold, and work the same way as the information address's values and categories, described in [section 3.1.4](#).

To illustrate how XY regions work, an XY region with start address 0100, width 5, height 5, row offset 8, X start 0, and Y start -1 would represent the following block of addresses, where 1s are relevant addresses and 0s are irrelevant.

0100	1	1	1	1	1	0	0	0	0107
0108	1	1	1	1	1	0	0	0	010F
0110	1	1	1	1	1	0	0	0	0117
0118	1	1	1	1	1	0	0	0	011F
0120	1	1	1	1	1	0	0	0	0127

The top left corner of the relevant addresses will have its category applied to the plane position (0, -1), the top right to (4, -1), the bottom left to (0, 3), and the bottom right to (4, 3).

## 3.2.6. Database settings

PuzzLearn - Database Settings

\*End Address (hex): 0

End Value: 0

Category	Color
0	#000000

Add Category

Edit Color

Button

Button Name:

Add Button

Delete Button

☐ Release

Population: 200 Stagnant Gen.: 15 Timeout: 600.00 Stagnant Timeout: 20.00

Confirm Cancel

Figure 15: The database settings form

The database settings form contains several pieces of miscellaneous information not tied to a specific structure but still used by the learning script. Settings include:

- **End Address and Value:** The end address and value define an end condition for a test run. When the value in the end addresses matches the end value, the learning algorithm will move on to the next network to text. An example of an end address would be an address defining the graphic displayed in a certain area, with its end value corresponding to the part of the “game over” graphic that should be displayed. Any address will work, as long as it only holds the end value after a failure state is reached.
- **Category Colors:** The list on the left side of the form defines what color each value in the plane will have in the display form. This can make the plane display better reflect the current game state; for example, in Dr. Mario, making the plane’s cells match the colors of the pills. Related options include:
  - **Add Category:** The configuration file builder will automatically add categories up to the maximum value held within all of the planes in the structures list, meaning all possible categories can be assigned. However, if you plan on adding more categories, clicking the “Add Category” button will add an additional one to the list.
  - **Edit Color:** Opens a color dialog to edit the color of the selected category. This dialog will include several preset colors, as well as a button labeled “Define Custom Colors >>” that will allow you to use a wider variety of colors.



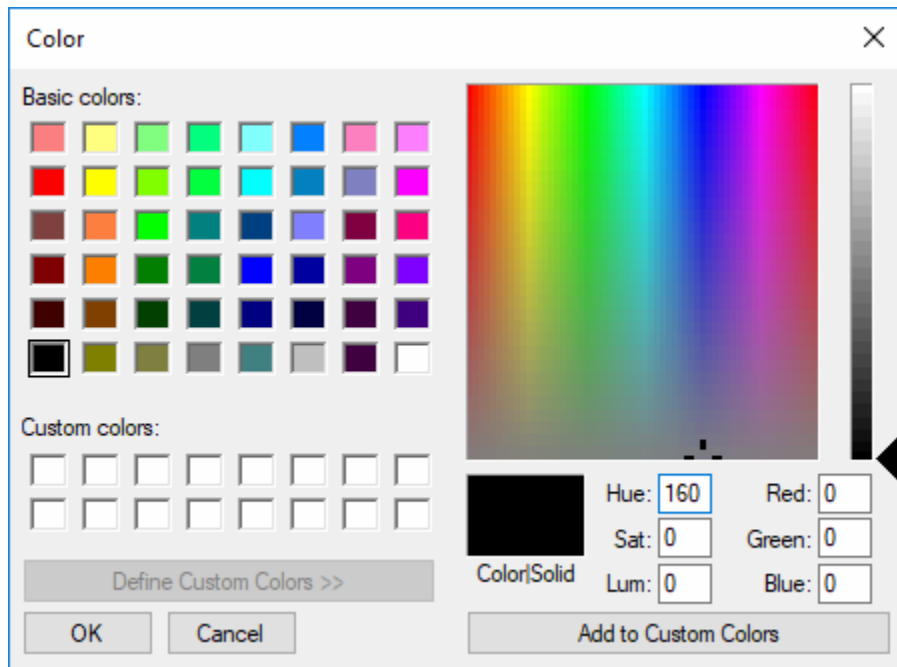


Figure 16: Color dialog

- **Buttons:** This list defines what output buttons are available when creating and testing neural networks. Related options include:
  - **Add Button:** Adds the line of text within the “Button Name” text box to the list of buttons. The “Button Name” text box includes a dropdown list for valid SNES button inputs. If a game from a different system requiring different inputs is being tested, enter a correct button line as described on the BizHawk site at <http://tasvideos.org/Bizhawk/LuaFunctions/JoypadTableKeyNames.html>.
  - **Delete Button:** Removes the selected button.
  - **Release:** If this option is checked, when testing each network the learning algorithm will release button inputs in between frames. Otherwise, the buttons will be held until the network specifically does not press them. Use if the game being configured does nothing when a button is held.
- **Population:** Defines the size of each generation.
- **Stagnant Generation (Stagnant Gen.):** Defines the threshold before a species becomes stagnant. If a species goes that many generations without its top fitness increasing, it and all of its networks will be removed from the pool.
- **Timeout:** Determines how long a network can be tested before automatically moving on to the next network, in seconds.

- **Stagnant Timeout:** Determines how long a network can go without its score increasing before moving on to the next network, in seconds.

### 3.2.7. Putting it together

Now that all aspects of PuzzLearn's memory structure are explained, you can begin finding addresses and arranging them together. A general guide for this process is as follows:

1. Begin playing the game you wish to create a configuration file for in the BizHawk emulator, preferably starting at the starting state that the learning script will use.
2. Try to determine what the most important pieces of information are while playing the game and how they might translate to structures within a plane or miscellaneous information addresses. If necessary, write these down somewhere.
3. Open BizHawk's RAM search.
4. For each miscellaneous information address, do the following:
  1. Use the methods described in [section 3.1.5](#) to find where the value is held and add it to the RAM watch.
  2. In the configuration file builder, add the address to the list of structures with a meaningful description, filling out all of its possible values with corresponding game states and giving a default value if necessary.
5. For each address plane, do the following:
  1. For every structure you think is in the plane, find all their associated values and add them, with meaningful labels, to the RAM watch. If you are unable to find a certain value, double-check that the RAM search is set to 1 byte. If it is, reconsider your approach. **Do not enter the value into the configuration file builder until all are recorded.** Internally, structures are processed in order, which can result in values being overwritten.

Some more notes on finding addresses:

- Commonly, related addresses will be grouped together (i.e. the X and Y position addresses of an object will often be nearby, and every address that is summed to produce a position or score will be next to each other).
  - For regions, it is enough to find the first iteration of the pattern, the first address of the second iteration, and the first address of the last iteration. The address increment is the difference between the second and first iterations.
  - For XY regions, finding one tile, resetting the search, and scrolling to that tile's address without filtering anything out will typically reveal where all the other tiles are and how they are stored.
2. Once all structures in the address plane are defined, determine the order in which they should be processed. Commonly, the order will be XY region(s), then normal region(s), then object(s).

3. Record the values in the configuration file builder in the order that they should be processed, starting with the first one. While doing this, ensure that the XY positions of each structure are coordinated and the categories of each structure are consistent and follow the rules described in [section 3.1.4](#).
6. Find and add all score address to the score address list.
7. Fill in the remaining information to the database settings. Some notes:
  - When finding the end address and end value, make sure that the end address will not hold the end value during regular play.
  - Higher values for population will result in more progress per generation, but more time within each generation. Generally speaking, keep populations lower for games with longer playtimes before an end state is reached.
  - Higher values for the stagnant generation will keep dead-end species longer, but allow more chances for an underperforming species to grow, with its different type of logic possibly surpassing the current best-performing species. Generally speaking, set a high stagnant generation value if the game is particularly complex.
  - The timeout value exists to prevent a network from running forever. Ideally, an end state or a stagnant timeout should happen reached before the timeout happens. In most games, it is safe to set the timeout value high.
  - The stagnant timeout value exists to stop a network when it is not going to score anymore. A stagnant timeout that is too low can eliminate a network that was going to score, while a stagnant timeout that is too high will unnecessarily increase the amount of time it takes to process a non-scoring network. Try to set the stagnant timeout as low as it should take to score even one additional point. This can vary a large amount depending on the game. If you are unsure, you can set the stagnant timeout to the same value as the normal timeout, effectively eliminating it, but depending on the game, this can significantly increase the amount of training time.
8. Save the finished file. Preferably, save two copies, one wherever you can easily access, which can be overwritten; and one in the PuzzLearnData\Config folder, which should not be. Modifying or moving a configuration file can prevent sessions that use them from working.
9. Test out your configuration file with the learning script.

### 3.3. Online functionality

The configuration file builder contains some features for sharing and downloading configuration files easily. This subsection describes how to use them.

#### 3.3.1. Downloading files

In order to access the download files menu, click on “Online > Download” from the main menu. The download form should appear.

The screenshot shows a web application window titled "PuzzLearn - Download". The main heading is "Download". Below it is a search input field and a "Search" button. A list of file names is displayed below the search bar, with "Tetris and Dr. Mario - Tetris" highlighted in blue. At the bottom of the window, there are two buttons: "Download" and "Back".

Figure 17: The download form

It will display a list of file names for configuration files made and uploaded by other users. Options for this form include:

- **Search:** Filters your files down to ones containing the text in the search text box.
- **Download:** Downloads the currently selected file to the configuration file builder. Note that this file still needs to be saved locally.
- **Back:** Returns to the main menu.

### 3.3.2. Managing your files

Each online file PuzzLearn stores has an owner in order to prevent them from being edited and deleted by those who have not made the file. In order to access and modify your files, click “Online > My files” from the main menu. A login form should appear.

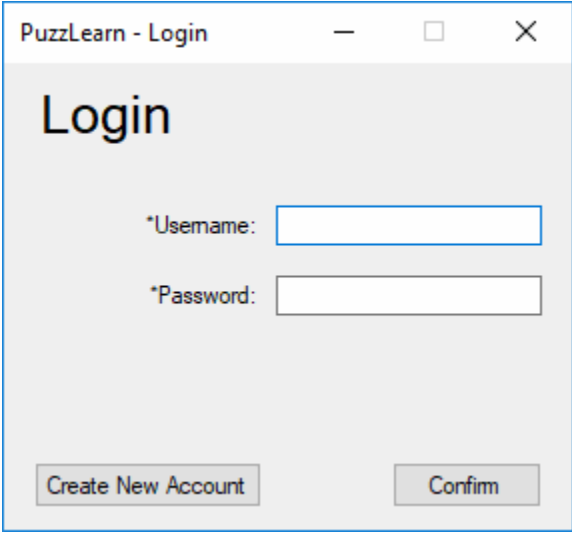
A screenshot of a web browser window titled "PuzzLearn - Login". The window has a light gray background. At the top, the title bar shows "PuzzLearn - Login" with standard minimize, maximize, and close buttons. Below the title bar, the word "Login" is displayed in a large, bold, black font. Underneath, there are two input fields: the first is labeled "\*Username:" and the second is labeled "\*Password:". Both fields are empty. At the bottom of the window, there are two buttons: "Create New Account" on the left and "Confirm" on the right. Both buttons are gray with black text.

Figure 18: The login form

Click “Create New Account” to create your own account.

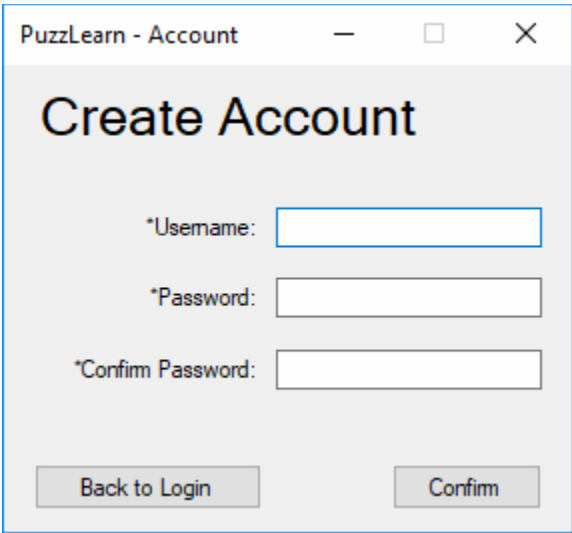
A screenshot of a web browser window titled "PuzzLearn - Account". The window has a light gray background. At the top, the title bar shows "PuzzLearn - Account" with standard minimize, maximize, and close buttons. Below the title bar, the words "Create Account" are displayed in a large, bold, black font. Underneath, there are three input fields: the first is labeled "\*Username:", the second is labeled "\*Password:", and the third is labeled "\*Confirm Password:". All three fields are empty. At the bottom of the window, there are two buttons: "Back to Login" on the left and "Confirm" on the right. Both buttons are gray with black text.

Figure 19: The create account form

Fill in a username, password and reentered password and click confirm to create your account. This will take you to your files, which will be empty to begin. In future sessions, enter your username and password to access your files.

The screenshot shows a web application window titled "PuzzLearn - My Files". Inside, there's a section titled "My Files". Below this title is a search bar with a "Search" button. A table is displayed with a single row. The table has a header row with "File name" and a data row with "Tetris and Dr. Mario - Tetris". Below the table, there's a label "Upload file name:" followed by a text input field. At the bottom of the form, there are two rows of buttons: the first row contains "Download", "Upload", "Update", and "Delete"; the second row contains "Back" and "Sign Out".

Figure 20: The “My Files” form

Options for this form include:

- **Search:** Filters your files down to ones containing the text in the search text box.
- **Download:** Downloads the currently selected file to the configuration file builder.
- **Upload:** Uploads the current contents of the configuration file builder to the online database with the given file name, allowing other users to download and modify it. It is recommended that the upload file name contains the full name of the game and, if applicable, the game mode it is meant to be played in.
- **Update:** Updates the contents of the currently selected file to instead hold the current contents of the configuration file builder.
- **Delete:** Deletes the currently select file from the online database, removing it permanently.
- **Back:** Returns to the main menu.
- **Sign Out:** Signs out for the current session and returns to the login form.