



A set of scripts for creating and learning your own artificial neural network.

Creating your own artificial neural network (ANN) is not a big issue. But to teach the ANN to do certain tasks is a real challenge. In order to ease the task of creating and teaching the ANN, I wrote a set of scripts, that will do everything for you. The only thing that is needed from you is to “explain” the ANN properly, what information is needed to be learned.

2019

CONTENT

<u>A general scripts description.....</u>	<u>3</u>
<u>The detailed description of the main scripts.....</u>	<u>4</u>
<u>Perceptron.cs.....</u>	<u>4</u>
<u>PerceptronLernByBackPropagation.cs.....</u>	<u>5</u>
<u>PerceptronLernByRandomGeneration.cs.....</u>	<u>6</u>
<u>ANN.cs.....</u>	<u>7</u>
<u>ANNNode.cs.....</u>	<u>8</u>
<u>ANNConnection.cs.....</u>	<u>8</u>
<u>ANNLearnByNEAT.cs.....</u>	<u>9</u>
<u>A description of auxiliary interfaces.....</u>	<u>10</u>
<u>The perceptron interface.....</u>	<u>10</u>
<u>The perceptron back propagation interface.....</u>	<u>11</u>
<u>The perceptron random generation interface.....</u>	<u>12</u>
<u>ANN interface.....</u>	<u>13</u>
<u>ANN learn by NEAT interface.....</u>	<u>14</u>
<u>How to use.....</u>	<u>15</u>
<u>The task “Mission is “Not to Die”</u>	<u>15</u>
<u>How to create the perceptron.....</u>	<u>18</u>
<u>Lesson # 1. Training with a sample of tasks and responses.....</u>	<u>20</u>
<u>Lesson # 2. Learning with a “teacher”</u>	<u>23</u>
<u>Lesson # 3. Training by the random generation method.....</u>	<u>25</u>
<u>How to save and load the perceptron settings.....</u>	<u>27</u>
<u>How to create the ANN.....</u>	<u>28</u>
<u>Lesson # 4. Training by NEAT.....</u>	<u>30</u>
<u>How to save and load the ANN settings.....</u>	<u>32</u>
<u>The conclusion.....</u>	<u>33</u>
<u>Contacts.....</u>	<u>33</u>

[A general scripts description.](#)

Perceptron.cs is a non MonoBehaviour script of an artificial neural network (ANN) the perceptron type. This script will automatically create the perceptron based on your parameters, fulfill a task (if the training takes place), save and load your ANN.

PerceptronInterface.cs script can be used to facilitate the creation, visualization, saving and loading of ANNs during the learning process. This is the interface for the perceptron in the game mode.

PerceptronLernByBackPropagation.cs is a non MonoBehaviour script for teaching the perceptron by the back propagation method. It is enough for you to specify a selection of tasks with responses or create a “teacher”, and ANN will start its training. And with the help of flexible settings, you can quickly and easily complete the training of the ANN.

PerceptronBackPropagationInterface.cs is an interface script for teaching the perceptron by the back propagation method in the game mode.

PerceptronLernByRandomGeneration.cs is a non MonoBehaviour script for teaching the perceptron by the randomly generating method of a certain amount of “clones” of the object, which you want to teach. All the generated “clones” receive the randomly modified perceptron. They receive it from the object of training of the first generation and from the best one of the previous generation on all of the subsequent generations. A large number of flexible learning settings allow one to get the self-taught perceptron.

PerceptronRandomGenerationInterface.cs is an interface script for teaching the perceptron by the random generation method in a game mode.

ANN.cs is a non MonoBehaviour script of an artificial neural network (ANN). This script will automatically create the ANN based on your parameters, fulfill a task (if the training takes place), save and load your ANN.

ANNNode.cs is a non MonoBehaviour script that contains the core information of the ANN’s node.

ANNConnection.cs is a non MonoBehaviour script that contains the core information of the ANN’s connection between nodes.

ANNInterface.cs script can be used to facilitate the creation, visualization, saving and loading of ANNs during the learning process. This is the interface for the ANN in the game mode.

ANNLearnByNEAT.cs is a non MonoBehaviour script for teaching the ANN by NEAT (NeuroEvolution of Augmenting Topologies). With the help of flexible settings, you can quickly and easily complete the training of the ANN.

ANNLearnByNEATInterface.cs is an interface script for teaching the ANN by the NEAT in the game mode.

DrawANNWeight.cs is a non MonoBehaviour script (static) for the perceptron’s weights visualization. It is used by the **PerceptronInterface.cs** script.

InterfaceGUI.cs is a non MonoBehaviour script (static) for interfaces’ scripts.

Formulas.cs is a non MonoBehaviour script (static) for fulfilling cyclic and alike tasks.

ActivationFunctions.cs is a non MonoBehaviour script (static) that contains activation functions.

A detailed description of the main scripts.

`public class Perceptron` – is a non MonoBehaviour script for an artificial neural network (ANN), type of the perceptron.

The main variables of the Perceptron.cs script	
<code>public int AFT</code>	The number of the activation function.
<code>public float AFS</code>	A size of the activation function.
<code>public bool B</code>	If it is “ <code>true</code> ”, then an additional displacement neuron is created in each layer, except for the original layer. This neuron is always equal to one. Its weight units affect all the neurons in the next layer, except for a bias neuron.
<code>public bool AFWM</code>	If it is “ <code>true</code> ”, then all the neurons take the values from -1 to 1. This also applies to incoming and outgoing neurons. If it is “ <code>false</code> ” then all the neurons take values from 0 to 1.
<code>public float[] Input</code>	The input layer of the ANN. The size of the array indicates the number of the input values, with a bias neuron (if <code>B == true</code> , the last neuron is always equal to 1).
<code>public int[] NIHL</code>	The number of neurons in the hidden layers, with a bias neuron. The size of the array indicates the number of the hidden layers, with a bias neuron (if <code>B == true</code> , the last neuron of each layer is always equal to 1).
<code>public float[] Output</code>	Output layer of the ANN. The size of the array indicates the number of the output values. It never contains a bias neuron.
<code>public float[][] Neuron</code>	The value of neurons. The first part of the array corresponds to the layer of the perceptron. The second part of the array is the number of neurons in the layer.
<code>public float[][][] NeuronWeight</code>	The value of the relations among the neurons. The first part of the array corresponds to the layer of the ANN. The second part of the array is the number of the neuron of the next layer. The third part of the array is the number of the current layer of the neuron.

The Command of the Script	The variables of the Commands	Description
<code>public void CreatePerceptron</code>		Creation of perceptron with the help of parameters.
	<code>float AFT</code>	The number of the activation function.
	<code>float AFS</code>	A Size of an activation function.
	<code>bool B</code>	If it is « <code>true</code> », then an additional bias neuron is created in an each layer, except for the original layer. This neuron is always equal to one. Its weight units affect all the neurons of the next layer.
	<code>bool AFWM</code>	If it is « <code>true</code> », then all the neurons’ values are from -1 to 1. It also applies to the input and output neurons. If it is « <code>false</code> », the neurons take values from 0 to 1.
	<code>int NumberOfInputs</code>	The amount of the input neurons, without a bias neuron.
	<code>int[] NIHL</code>	An array that specifies the number of neurons (without a bias neuron) in each hidden layer of the ANN. The size of the array indicates the number of hidden layers.
	<code>int NumbersOfOutputs</code>	The number of output neurons.
<code>public void Load</code>		Load the perceptron from the file.
	<code>string PerceptronFile</code>	Filename for loading.
<code>public void Solution</code>		Perceptron solution.
<code>private float Sumator</code>		The total sum of all the values of the neurons multiplied by their weight.
	<code>float[] Neuron</code>	The neuron of a certain layer.
	<code>float[] NeuronWeight</code>	The weights of the same layer as the neuron.
<code>private void CreatingNeurons</code>		Creation of the neurons and their units with each other.
	<code>StreamReader SR</code>	The specified stream from the load file. Use « <code>null</code> » if the file is not used.
<code>public void Save</code>		Save perceptron parameters to a file.
	<code>string PerceptronFile</code>	The name of the file is to be loaded.

`public class PerceptronLernByBackPropagation` is a non `MonoBehaviour` script for teaching the perceptron by the back propagation method.

The main variables of the <code>PerceptronLernByBackPropagation.cs</code> script	
<code>public int LearningSpeed</code>	The number of learning steps per a frame of the game mode. If the time spent on 1 frame exceeds 1 second, « <code>LearningSpeed</code> » will be continued from the next frame.
<code>public int LearnIteration</code>	A counter of learning steps.
<code>public float LearningRate</code>	The power of a change of weight units during the learning process. It affects the speed and quality of the learning.
<code>public float DesiredMaxError</code>	It specifies the maximum difference between the correct response and the ANN response.
<code>public float MaxError</code>	The maximum difference between the correct response and the ANN response.
<code>public bool Learned</code>	If the difference between the correct response and the ANN response is less than <code>DesiredMaxError</code> , then the perceptron is considered to be trained.
<code>public bool ShuffleSamples</code>	If it is « <code>true</code> », then samples of tasks and responses will be shuffled during the training.
<code>public bool WithError</code>	If « <code>true</code> », then back propagation work only with output what have error more then <code>DesiredMaxError</code> .

The Command of the Script	The Variables of the Command	Description
<code>public void Learn</code>		The first variation the command. Teaching of the perceptron by the back propagation method with the content of a certain number of tasks with responses. <code>public void Learn</code> can use <code>ShuffleSamples</code> .
	<code>Perceptron PCT</code>	The perceptron is to be taught.
	<code>float[][] Task</code>	The array of tasks. The first part of the array contains the task number. The second part of the array is the number of the input neuron.
	<code>float[][] Answer</code>	An array of responses. The first part of the array contains the response number. The second part of the array is the number of the output neuron.
<code>public void Learn</code>		The second variation the command can be used. Teaching the perceptron by the back propagation method with the contents of one task with the response. It does not use <code>LearningSpeed</code> , <code>DesiredMaxError</code> and <code>ShuffleSamples</code> . It is recommended to use <code>DesiredMaxError = 0</code> .
	<code>Perceptron PCT</code>	The perceptron is to be taught.
	<code>float[] Task</code>	The array of one task for each input neuron of the perceptron.
	<code>float[] Answer</code>	The array of one response for each output neuron of the perceptron.
<code>public void Learn</code>		The third variation the command is to be used. Teaching the perceptron by the back propagation method with the contents of one response. Use it, if the task is directly input into the input layer of the perceptron. It does not use <code>LearningSpeed</code> , <code>DesiredMaxError</code> and <code>ShuffleSamples</code> . It is recommended to use <code>DesiredMaxError = 0</code> .
	<code>Perceptron PCT</code>	The perceptron is to be taught.
	<code>float[] Answer</code>	The array of one response for each output neuron of the perceptron.
<code>public void ModificateStartWeights</code>		Modification of the weight units of the perceptron before learning.
	<code>Perceptron PCT</code>	The perceptron is to be taught.
	<code>bool MSW</code>	If it is « <code>true</code> », then the weight units are modified by a special algorithm. If it is « <code>false</code> », then all the weight units are generated randomly in the range from -0.5 to 0.5.

`public class PerceptronLernByRandomGeneration` is a non MonoBehaviour script for teaching the perceptron by the random generation method of a certain number of «clones» of the object is to be taught.

The main variables of the PerceptronLernByRandomGeneration.cs script	
<code>public int</code> AmountOfChildren	The number of «children» in an each generation.
<code>public bool</code> ChildrenByWave	If «true» - by waves. If «false» - use maximum in one time.
<code>public int</code> ChildrenInWave	The maximum number of «children» at the same time in the generation.
<code>public int</code> BestGeneration	The best generation at the moment.
<code>public int</code> Generation	The total number of generations.
<code>public int</code> ChildrenInGeneration	The number of «children» in the last generation.
<code>public float</code> BestLongevity	The best «longevity» at the moment.
<code>public float</code> ChildrenDifference	The difference of the weight units among the generations.
<code>public float</code> ChildrenDifferenceAfterEffects	The difference of weight units with the affect coefficient among the generations.
<code>public bool</code> ChildrenGradient	If it is «true», then the command smoothes the difference of weight units linearly among the «children» in the generation.
<code>public float</code> GenerationEffect	The reduction factor of the effect on the difference of weight units among the generations. The command works under the condition that it is not equal to zero and the current generation is better than the previous one.
<code>public float</code> GenerationSplashEffect	The increasing factor of the effect on the difference of weight units among the generations. The command works under the condition that it is not equal to zero and the current generation is worse than the previous one.
<code>public float</code> ChangeWeightSign	
<code>public float</code> Chance	A chance to choose the better from the worse, current generation. It changes because of the parameter <code>public float</code> ChanceCoefficient with each new generation.
<code>public float</code> ChanceCoefficient	The affect coefficient on the chance of random choice of the worse, current generation. The command works under the condition that it is not equal to zero.
<code>public bool</code> Autosave	Enable / disable auto save while ANN learning.
<code>public int</code> AutosaveStep	Step of auto save.
<code>public string</code> AutosaveName	Start part of the file name for auto-save.
<code>public bool</code> IgnoreCollision	Enable / disable the "children's" collision.

The Command of the Script	The Variables of the Command	Description
<code>public void</code> StudentData		A data collection for the learning.
	<code>GameObject</code> Student	The main game object (<code>GameObject</code>), that has to learn.
	<code>Object</code> HerelsANN	A script, that contains the perceptron.
	<code>string</code> PerceptronName	The name of the variable (<code>Perceptron</code>), that was given to the perceptron in the script, which contains the perceptron (HerelsANN).
	<code>Object</code> StudentControls	The control script of main game object.
	<code>string</code> StudentCrash	The name of the variable (<code>bool</code>), that was given to the reason for the «crash» of the game object in the control script (StudentControls).
	<code>string</code> StudentLife	The name of the variable (<code>float</code>), that was given to the «longevity» of the game object in the control script (StudentControls).
<code>public void</code> Learn		Teaching the perceptron by the random generation method.
	<code>Perceptron</code> PCT	The perceptron is to be taught.
<code>public void</code> StopLearn		The immediate stop of learning with the transfer of weight units information of the better perceptron from a better generation to the perceptron, that is being trained.
	<code>Perceptron</code> PCT	The perceptron is to be taught.
<code>public void</code> Reset		Reset learning information.

`public class ANN` – is a non MonoBehaviour script for an artificial neural network (ANN).

The main variables of the ANN.cs script	
<code>public float AFT</code>	The number of the activation function.
<code>public float AFS</code>	A size of the activation function.
<code>public bool AFWM</code>	If it is “ <code>true</code> ”, then all the neurons take the values from -1 to 1. This also applies to incoming and outgoing neurons. If it is “ <code>false</code> ” then all the neurons take values from 0 to 1.
<code>public float[] Input</code>	The input layer of the ANN. The size of the array indicates the number of the input values.
<code>public float[] Output</code>	Output layer of the ANN. The size of the array indicates the number of the output values.
<code>public Dictionary<int, ANNNode> Node</code>	Dictionary of the nodes (neurons) of ANN. Contains each node of the ANN.
<code>public Dictionary<int, ANNConnection> Connection</code>	Dictionary of the connections (neurons) of ANN. Contains each node of the ANN.

The Command of the Script	The variables of the Commands	Description
<code>public void Create</code>		Creation of ANN with the help of parameters.
	<code>float AFT</code>	The number of the activation function.
	<code>float AFS</code>	A Size of an activation function.
	<code>bool AFWM</code>	If it is « <code>true</code> », then all the neurons’ values are from -1 to 1. It also applies to the input and output neurons. If it is « <code>false</code> », the neurons take values from 0 to 1.
	<code>int NumberOfInputs</code>	The amount of the input neurons, without a bias neuron.
	<code>int NumbersOfOutputs</code>	The number of output neurons.
<code>public void Load</code>		Load the ANN from the file. Supports “ANN Perceptron” storage files.
	<code>string ANNFile</code>	Filename for loading.
<code>public void Solution</code>		ANN’s solution.
<code>private void CreatingNeurons</code>		Creation of the neurons and their units with each other.
	<code>StreamReader SR</code>	The specified stream from the load file. Use « <code>null</code> » if the file is not used.
<code>public void Save</code>		Save ANN’s parameters to a file.
	<code>string ANNFile</code>	The name of the file is to be loaded.
<code>public void FixConnections()</code>		Arrange the numbering of nodes in the connections.
<code>public void NumberingCorrection()</code>		Sort node numbering in order.

`public class ANNNode` is a non MonoBehaviour script that contains the core information of the ANN's node.

The main variables of the ANNNode.cs script	
<code>public float Neuron</code>	The value of neuron.
<code>public float Bias</code>	The value of neuron's bias.
<code>public ArrayList ConnectionIn</code>	List of input connections numbers.
<code>public int Fullness</code>	The fullness of the neuron connections when solving.
<code>public Vector2 Position</code>	Position at visualization.
<code>public int AFT</code>	The number of the neuron activation function. If AFT == -1 then AFT = default activation function of ANN.
<code>public bool UseMemory</code>	Neuron's memory usage.
<code>public float NeuronMemory</code>	The value of neuron's memory.
<code>public float WeightMemory</code>	The value of neuron's memory weight.

The Command of the Script	The variables of the Commands	Description
<code>public ANNNode</code>		Input information about the neuron.
	<code>float Neuron</code>	The value of neuron (node).
	<code>float Bias</code>	The value of neuron's bias.
	<code>ArrayList ConnectionIn</code>	List of input connections numbers.
	<code>int Fullness</code>	The fullness of the neuron connections when solving.
	<code>Vector2 Position</code>	Position at visualization.
	<code>int AFT</code>	The number of the neuron activation function. If AFT == -1 then AFT = default activation function of ANN.
	<code>bool UseMemory</code>	Neuron's memory usage.
	<code>float WeightMemory</code>	The value of neuron's memory weight.
	<code>string Info</code>	Text view of a neuron (node).
<code>public override string ToString</code>		Transfers information of the node to a text view.

`public class ANNConnection` is a non MonoBehaviour script that contains the core information of the ANN's connection.

The main variables of the ANNConnection.cs script	
<code>public int In</code>	Input neuron number.
<code>public int Out</code>	Output neuron number.
<code>public float Weight</code>	Connection weight.
<code>public bool Enable</code>	Connection activity.
<code>public bool IsMemory</code>	If «true» - the connection gets the data of the neuron from the previous solution.
<code>public float PreviousData</code>	The value of the input neuron of the weight from the previous solution.

The Command of the Script	The Variables of the Command	Description
<code>public void ANNConnection</code>		Input information about the connection.
	<code>public int In</code>	Input neuron number.
	<code>public int Out</code>	Output neuron number.
	<code>public float Weight</code>	Connection weight.
	<code>public bool Enable</code>	Connection activity.
	<code>string Info</code>	Text view of a connection.
<code>public override string ToString()</code>		Transfers information of the connection to a text view.

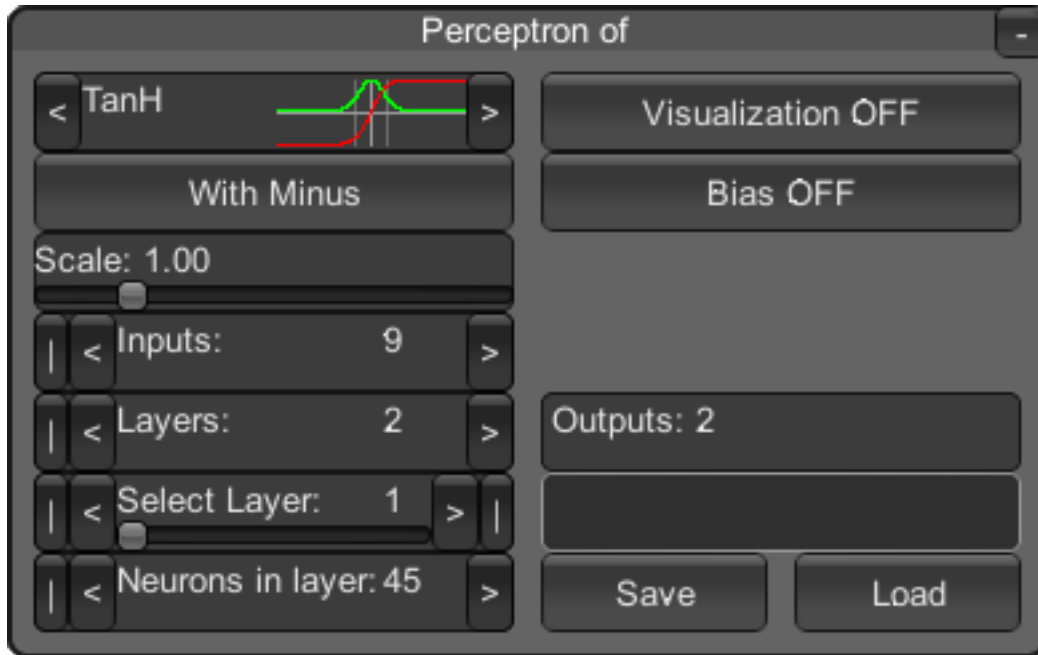
`public class ANNLearnByNEAT` is a non MonoBehaviour script for teaching ANN.

The main variables of the ANNLearnByNEAT.cs script	
<code>public ANN Ann</code>	ANN what is need to be learning.
<code>public int AmountOfChildren</code>	The number of «children» in an each generation.
<code>public int ChildrenInWave</code>	Amount of «children» in each generation's wave.
<code>public bool ChildrenByWave</code>	If «true» - by waves. If «false» - use maximum in one time.
<code>public int BestGeneration</code>	The best generation at the moment.
<code>public int Generation</code>	The total number of generations.
<code>public int ChildrenInGeneration</code>	The number of «children» in this generation.
<code>public bool Cross</code>	Permission to «cross» the two best «children».
<code>public bool PerceptronStart</code>	If «true», then in the first generation of children all input neurons will be connected with the output neurons.
<code>public bool Autosave</code>	Auto-save when learning.
<code>public int AutosaveStep</code>	Step of auto-save.
<code>public string AutosaveName</code>	The initial file name for auto-save.
<code>public bool IgnoreCollision</code>	If «true», it will be ignored the collision of children.
<code>public float BestLongevity</code>	The best «longevity» at the moment.
<code>public float ChildrenDifference</code>	The difference of the weight units among the generations.
<code>public float Chance</code>	A chance to choose the better from the worse, current generation. It changes because of the parameter <code>public float ChanceCoefficient</code> with each new generation.
<code>public float ChanceCoefficient</code>	The affect coefficient on the chance of random choice of the worse, current generation. The command works under the condition that it is not equal to zero.
<code>public float MutationAddNeuron</code>	Proportion of using mutation «add neuron».
<code>public float MutationAddWeight</code>	Proportion of use of mutation «add weight».
<code>public float MutationChangeOneWeight</code>	Proportion of use of mutation «change one weight».
<code>public float MutationChangeWeights</code>	Proportion of using mutation «change all weights».
<code>public float MutationChangeOneBias</code>	Proportion of using mutation «change one bias».
<code>public float MutationChangeBias</code>	Proportion of using mutation «change all bias».
<code>public float MutationSum</code>	The sum of the proportional values of the mutation.
<code>public int AddingWeightsCount</code>	The number of simultaneous adding of neuronal relationships to each «child».
<code>public float ChangeWeightSign</code>	The chance to change the sign of the weight.
<code>public bool[] UseMemoryWeight</code>	If "true", ANN can get "memory weights". [0] - hidden layer. [1] - from outputs to the hidden layer.
<code>public float[] UseNeuronsMemory</code>	If "true", ANN can get "Memory for neurons". [0] - hidden layer. [1] - outputs layer.
<code>public List<int> AFT</code>	List of functions of activation of neurons. If the list is empty, neurons will use the default activation function in ANN.

The Command of the Script	The Variables of the Command	Description
<code>public void StudentData</code>		A data collection for the learning.
	<code>GameObject Student</code>	The main game object (<code>GameObject</code>), that has to learn.
	<code>Object HereIsANN</code>	A script, that contains ANN.
	<code>string ANNName</code>	The name of the variable (<code>ANN</code>), that was given to the ANN in the script, which contains ANN (<code>HereIsANN</code>).
	<code>Object StudentControls</code>	The control script of main game object.
	<code>string StudentCrash</code>	The name of the variable (<code>bool</code>), that was given to the reason for the «crash» of the game object in the control script (<code>StudentControls</code>).
	<code>string StudentLife</code>	The name of the variable (<code>float</code>), that was given to the «longevity» of the game object in the control script (<code>StudentControls</code>).
<code>public void Learn</code>		Teaching the ANN.
<code>public void StopLearn</code>		The immediate stop of learning with the transfer of weight units information of the better ANN from a better generation to ANN, that is being trained.
<code>public void Reset</code>		Reset learning information.

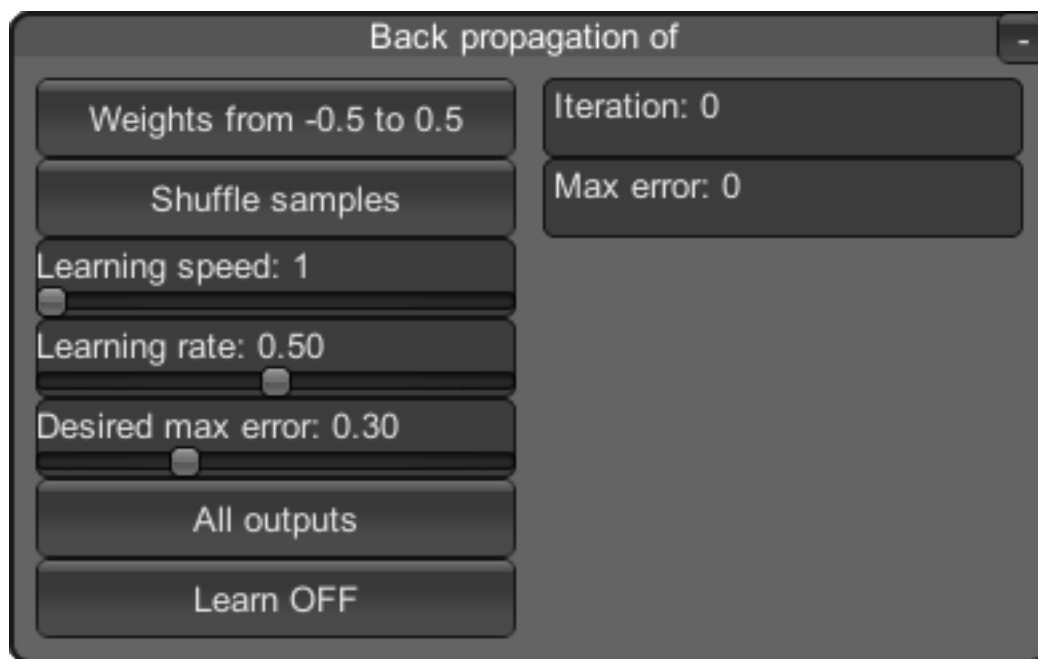
Description of the auxiliary interfaces.

PerceptronInterface.cs - this script can be used to facilitate the creation, storage, and loading of ANN during the learning process. This is the interface for the perceptron in the game mode. It controls the parameters of the **Perceptron.cs** script.



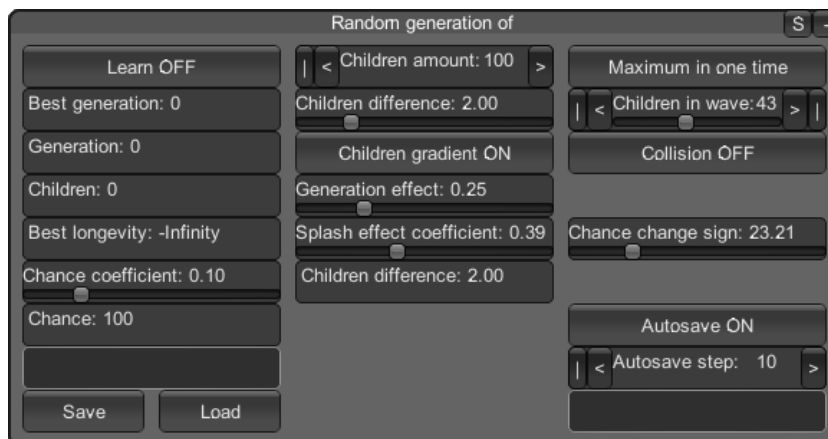
Activation functions	The choice of the activation function and its visualization (the «red» graph - the activation function, «green» - the derivative of this function) for all neurons in the ANN.
Without Minus With Minus	If «With Minus» is shown, then all the neurons' values are from -1 to 1. This also applies to the input and output neurons. If «Without Minus» is shown, then all the neurons' values are from 0 to 1. It controls the parameter <code>public bool AFWM</code> .
Bias OFF Bias ON	If «Bias ON» is shown, then an additional bias neuron is created in each layer, except for the output layer. This neuron is always equal to one. Its weight units affect all the neurons in the next layer, except for the bias neuron. If «Bias OFF» is shown, then there is no bias neuron. The controls the parameter <code>public bool B</code> .
Scale	The scale of the activation function. The controls the parameter <code>public float AFS</code> .
Inputs	It indicates the number of neurons in the input layer, the bias neuron is not taken into consideration. It affects the following arrays <code>float[] Input</code> , <code>public float[][] Neuron</code> and <code>public float[][][] NeuronWeight</code> .
Layers	Layers It indicates the number of the hidden layers. It affects the following arrays <code>public int[] NIHL</code> , <code>public float[][] Neuron</code> and <code>public float[][][] NeuronWeight</code> .
Select Layer	Selecting the hidden layer is to be changed.
Neurons in layer	It indicates the number of the neurons in the selected hidden layer. It affects the following arrays <code>public int[] NIHL</code> , <code>public float[][] Neuron</code> and <code>public float[][][] NeuronWeight</code> .
Visualization OFF Visualization ON	If «Visualization ON» is shown, then it shows the structure of the perceptron with the given parameters. If «Visualization OFF» is shown, then it does not show the structure of the perceptron with the given parameters. It uses the PerceptronVisualization.cs script.
Outputs	It shows the number of the output neurons. Their number is to be specified, when creating the perceptron.
GUI.TextField	The filename for the saving or loading the perceptron.
Save	It saves the parameters and all the weight units of the perceptron into the file, if the filename is specified. It controls the command <code>public void Save</code> .
Load	It loads the parameters and all the weight units of the perceptron into the file, if the filename is specified. It controls the command <code>public void Load</code> .

PerceptronBackPropagationInterface.cs – is an interface script for learning the perceptron by the back propagation method in the game mode. It controls the parameters of the **PerceptronLernByBackPropagation.cs** script.



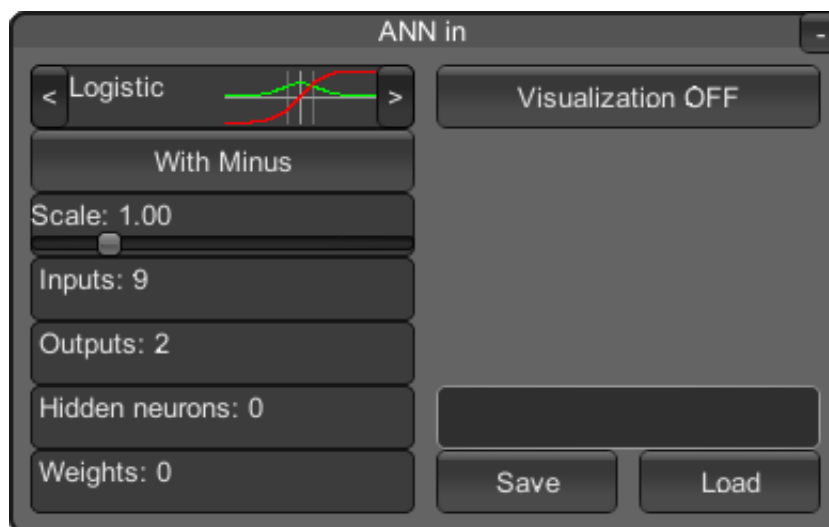
Weights from -0.5 to 0.5 Mod Weights	Modification of the weight units of the perceptron before learning. If « Mod Weights » is used, then the weight units are modified by a special algorithm. If « Weights from -0.5 to 0.5 » is used, then all the weight units are randomly generated in the range from -0.5 to 0.5. It controls the command <code>public void ModificateStartWeights</code> .
Samples one by one Shuffle samples	Shuffling the samples of the tasks and responses. If « Shuffle samples » is used, then the samples of the tasks and responses are shuffled during the training. If « Samples one by one » is used, then the samples go in a specified order. It controls the parameter <code>public bool ShuffleSamples</code> .
Learning speed	It indicates the number of training steps per frame of the game mode. It controls the parameter <code>public int LearningSpeed</code> .
Learning rate	It indicates the change power of weight units during the training. It affects the speed and quality of the training. It controls the parameter <code>public float LearningRate</code> .
Desired max error	It indicates the maximum difference between the correct response and the response of the ANN among all the sample tasks and responses. It controls the parameter <code>public float DesiredMaxError</code> .
All outputs Errorred outputs	If « Errorred outputs », then back propagation work only with output what have error more then <code>DesiredMaxError</code> .
Learn OFF Learn ON	If « Learn ON » is showed, then it starts the training of the perceptron by the back propagation method. It controls the first variation of the command <code>public void Learn</code> .
Iteration	It shows the amount of training steps. It gets data from the parameter <code>public int LearnIteration</code> .
Max error	It displays the maximum response error of the output neurons and the sample response among all the sample responses. It gets data from the parameter <code>public float MaxError</code> .

PerceptronRandomGenerationInterface.cs - is an interface script for training the perceptron by the random generation method in the game mode. It controls the parameters of the **PerceptronLernByRandomGeneration.cs** script.



Learn OFF Learn ON	If «Learn ON» is used, then it starts the training of the perceptron by the random generation method. It controls the command <code>public void Learn</code> . If «Learn OFF» is used and the training took place, then it stops the training. It uses the command <code>public void StopLearn</code> .
Best Generation	It shows the best generation number at the very moment. It gets the data from the parameter <code>public int BestGeneration</code> .
Generation	It shows what generation is like at the moment. It gets the data from the parameter <code>public int Generation</code> .
Children	It shows the number of «children» in the current generation. It gets the data from parameter <code>public int ChildrenInGeneration</code> .
Best longevity	It shows a better longevity at the moment. It gets the data from the parameter <code>public float BestLongevity</code> .
Chance coefficient	The coefficient of influence on the chance of the random choice the current worst generation. It affects under the circumstance, that it is not equal to 0. It controls the parameter <code>public float ChanceCoefficient</code> .
Chance	It shows the chance of choosing the current worse generation. It changes with each new generation. It gets the data from the parameter <code>public float Chance</code> .
GUI.TextField	Filename to save or load learning settings.
Save / Load	Save/Load learning settings.
Children amount	The amount of «children» in each generation. It controls the parameter <code>public int AmountOfChildren</code> .
Children difference	The difference of weight units among the generations. It controls the parameter <code>public float ChildrenDifference</code> .
Children gradient OFF Children gradient ON	If «Children gradient ON» is used, then it smoothes linearly the difference of the weight units among the «children» in the generation.
Generation effect	The reduction factor of the effect on the difference of the weight units among the generations. It affects it under the circumstance that it is not equal to zero and the current generation is better than the previous one. It controls the parameter <code>public float GenerationEffect</code> .
Splash effect coefficient	The magnification factor of the influence on the difference in weight units among the generations. It affects it under the circumstance that it is not equal to zero and the present generation is worse than the previous one. It controls the parameter <code>public float GenerationSplashEffect</code> .
Children difference	The difference of weight units among the generations with the coefficients of affect. It gets the data from the parameter <code>public float ChildrenDifferenceAfterEffects</code> .
Maximum in one time By waves	Ability to study by waves in a generation. A constant maximum number or a certain amount in a single wave.
Children in wave	The number of «children» in one wave.
Collision ON Collision OFF	Enable / disable collision between «children».
Chance change sign	The chance of changing the weight sign to the opposite after the generation impact on it.
Autosave OFF Autosave ON	Enable/disable autosave of the perceptron during training.
Autosave step	Save the perceptron at each specified step.
GUI.TextField	The filename for the saving or loading the perceptron.

ANNInterface.cs - this script can be used to facilitate the creation, storage, and loading of ANN during the learning process. This is the interface for the ANN in the game mode. It controls the parameters of the **ANN.cs** script.



Activation functions	The choice of the activation function and its visualization (the «red» graph - the activation function, «green» - the derivative of this function) for all neurons in the ANN.
Without Minus With Minus	If "With Minus" is shown, then all the neurons' values are from -1 to 1. This also applies to the input and output neurons. If "Without Minus" is shown, then all the neurons' values are from 0 to 1. It controls the parameter public bool AFWM .
Scale	The scale of the activation function. The controls the parameter public float AFS .
Inputs	It indicates the number of neurons in the input layer.
Outputs	It indicates the number of neurons in the output layer.
Hidden neurons	Total number of hidden neurons.
Weights	Total number of weights.
Visualization OFF Visualization ON	If "Visualization ON" is shown, then it shows the structure of ANN with the given parameters. If "Visualization OFF" is shown, then it does not show the structure of ANN with the given parameters.
GUI.TextField	The filename for the saving or loading ANN.
Save	It saves the parameters and all the weight units of ANN into the file, if the filename is specified. It controls the command public void Save .
Load	It loads the parameters and all the weight units of the ANN from the file, if the filename is specified. It controls the command public void Load .

ANNLearnByNEATInterface.cs - is an interface script for teaching the ANN by NEAT (NeuroEvolution of Augmenting Topologies). It controls the parameters of the **ANNLearnByNEAT.cs** script.



ANNLearnByNEATInterface

Learn OFF Learn ON	If "Learn ON" is used, then it starts the training of the ANN. It controls the command <code>public void Learn</code> . If "Learn OFF" is used and the training took place, then it stops the training. It uses the command <code>public void StopLearn</code> .
Best Generation	It shows the best generation number at the very moment. It gets the data from the parameter <code>public int BestGeneration</code> .
Generation	It shows what generation is like at the moment. It gets the data from the parameter <code>public int Generation</code> .
Children	It shows the number of "children" in the current generation. It gets the data from parameter <code>public int ChildrenInGeneration</code> .
Best longevity	It shows a better longevity at the moment. It gets the data from the parameter <code>public float BestLongevity</code> .
Chance coefficient	If !=0 than the chance of changing the weight sign to the opposite after the generation impact on it. Manipulates the parameter <code>public float ChanceCoeefficient</code> .
Chance	It shows the chance of choosing the current worse generation. It changes with each new generation. It gets the data from the parameter <code>public float Chance</code> .
Children amount	The amount of "children" in each generation. It controls the parameter <code>public int AmountOfChildren</code> .
Crossing OFF Crossing ON	Enable / Disable "cross" the two best "children". It controls the parameter <code>public bool Cross</code> .
No weights Perceptron	If the "Perceptron" shows, then in the first generation of children all the input neurons will be connected to the output neurons. It controls the parameter <code>public bool PerceptronStart</code> .
Ratio add weight	Proportion to use the "add weight" mutation. It controls the parameter <code>public float MutationAddWeight</code> .
Ratio change 1 weight	Proportion of use of mutation "change one weight". It controls the parameter <code>public float MutationChangeOneWeight</code> .
Ratio change weights	Proportion of using mutation "change all weights". It controls the parameter <code>public float MutationChangeWeights</code> .
Ratio add neuron	Proportion of using mutation to "add neuron". It controls the parameter <code>public float MutationAddNeuron</code> .
Ratio change 1 bias	Proportion of using mutation "change one bias". It controls the parameter <code>public float MutationChangeOneBias</code> .
Ratio change bias	Proportion of mutation use "change all bias". It controls the parameter <code>public float MutationChangeBias</code> .
M	
Maximum in one time By waves	Ability to study by waves in a generation. A constant maximum number or a certain amount in a single wave. It controls the parameter <code>public bool ChildrenByWave</code> .
Children in wave	The number of "children" in one wave. It controls the parameter <code>public int ChildrenInWave</code> .
Collision ON Collision OFF	Enable / disable collision between "children". It controls the parameter <code>public bool IgnoreCollision</code> .
Maximum count of weights	The maximum number of simultaneous connections between neurons in each "child". It controls the parameter <code>public int AddingWeightsCount</code> .
Chance change sign	The chance to change the sign of weight. It controls the parameter <code>public float ChangeWeightSign</code> .
Children difference	The difference of weight units among the generations. It controls the parameter <code>public float ChildrenDifference</code> .
Autosave OFF Autosave ON	Enable/disable auto-save of ANN during training. It controls the parameter <code>public bool Autosave</code> .
Autosave step	Save the ANN at each specified step. It controls the parameter <code>public int AutosaveStep</code> .
GUI.TextField	The filename for the saving or loading ANN.
A	
Hidden X Hidden Hidden ↔ Hidden	If "true", ANN can get "memory weights" in the hidden layer.
Hidden → Outputs Hidden ↔ Outputs	If "true", ANN can get "memory weights" between hidden and output layers.
M-N in HL chance	If "true", ANN's neurons can get "memory" in the hidden layer.
M-N in OL chance	If "true", ANN's neurons can get "memory" in the output layer.
"Scroll list"	List of activation functions for new neurons.

How to use.

First of all, a specific task for ANN has to be created. It is to be remembered, that the ANN must receive a certain input data and it has to be properly prepared. The perceptron accepts input data from 0 to 1, or from -1 to 1 ([see Page 4](#), «public bool AFWM» for perceptron; [see Page 7](#), «public bool AFWM» for ANN). It is also necessary to determine the number of the input data.

The same applies to the output data. Consequently, they must be correctly converted for the correct response to the task.

The amount of the hidden layers and the neurons in perceptron depends on you, but for ANN - depends on learning of ANN. There is no use sometimes to create a hidden layer for certain types of tasks, but sometimes on the contrary, more layers and more neurons are needed. In any case, their amount has a different influence on the quality and speed of learning of the ANN.

Next step. It is needed to decide, what method to teach ANN with. If there already are prepared samples of tasks and responses, or the task can be solved by a third-party «teacher», then the back propagation method can be used (for perceptron). If there are no prepared samples or a «teacher», then the random generation method can be used.

I prepared a set of lessons for a better understanding how to use the perceptron and ANN.

The task “Mission is “Not to Die”.

Let's look into the prepared task: There is a «cow» which eventually wants to eat. There is «food», that the «cow» can eat. The «cow» will die, if it does not eat or eat too much. The «cow» can only eat with the front side of the body, otherwise it will die. The «cow» must be taught to eat properly and in time.

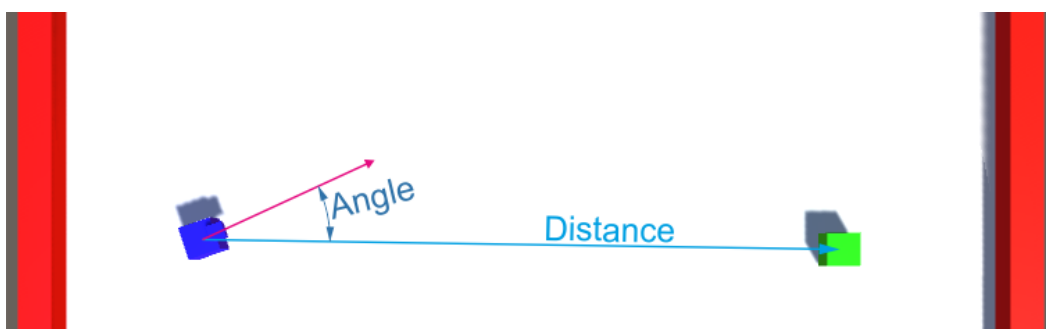
The Tutorial folder has already prepared scripts and a scene for the task.

The «cow» gets three meanings:

1. The distance to «food». The level diagonal is about 41.
2. The angle of rotation with the sign relative to the front of the «cow» to «food». From -180 to 180.
3. The «cow's fullness» that decreases over time. 50 is the maximum value for the «survival».

The «cow» is guided by two values:

1. Turn to food. From -1 to 1.
2. Move to food. From -1 to 1.



Also, the «cow» has the additional values for getting the above mentioned.

The «Cow» script is the following **TutorialCowControl.cs**:

```
using UnityEngine;

public class TutorialCowControl : MonoBehaviour
{
    public GameObject Food;           //Food's GameObject
    public float DistanceToFood = 0;  //Distance to food
    public float AngleToFood = 0;     //Angle to food

    public float Turn = 0;             //Turn of cow
    public float Move = 0;             //Move of cow
    public float Satiety = 40;         //Satiety of cow
    public bool Death = false;        //If true - cow will die (reset position)

    public float LifeTime = 0;        //Lesson 3 & 4 !!!

    void Update()
    {
        //Max & min turn
        if (Turn > 1)
            Turn = 1;
        else if (Turn < -1)
            Turn = -1;

        //Max & min move
        if (Move > 1)
            Move = 1;
        else if (Move < -1F)
            Move = -1F;

        //Cow reset
        if (Death)
        {
            Satiety = 40;
            transform.position = new Vector3(0, 0.5F, 0);
            transform.eulerAngles = new Vector3(0, transform.eulerAngles.y, 0);
            Death = false;
        }

        //Controls of cow
        transform.Rotate(0, Turn * 10F, 0);
        transform.Translate(0, 0, Move / 10F);

        //Food info
        DistanceToFood = Vector3.Distance(transform.position, Food.transform.position);
        AngleToFood = Vector3.Angle(transform.forward, Food.transform.position - transform.position)
* Mathf.Sign(transform.InverseTransformPoint(Food.transform.position).x);

        //The satiety of the cow decreases with time
        Satiety -= Time.deltaTime;
        if (Satiety < 0 || Satiety > 50)
            Death = true;

        LifeTime += Time.deltaTime - (Mathf.Abs(AngleToFood) / 180F) * Time.deltaTime;
    //Lesson 3 & 4!!!
    }
}
```


«Food» affects the cow, when it is touched. If the «cow» eats «food» correctly (the angle of rotation does not exceed 5 degrees), then it increases its «fullness» (+15), and «food» changes its position. Otherwise it will die.

The «food» script is the following **TutorialFood.cs**:

```
using UnityEngine;

public class TutorialFood : MonoBehaviour
{
    private bool Moving = false;

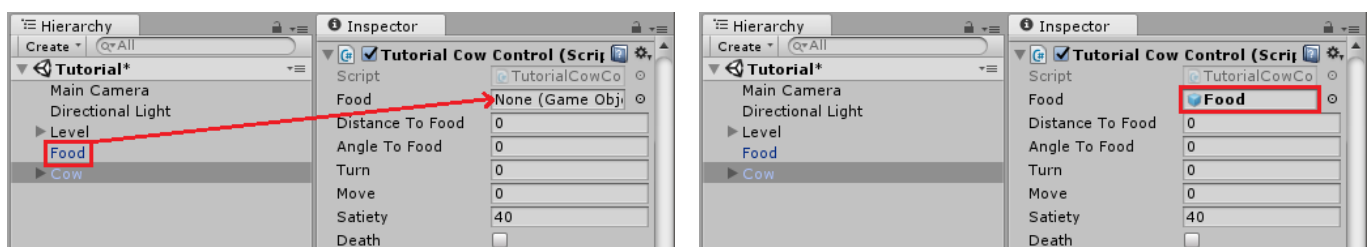
    void Start ()
    {
        MoveFood();          //Move food
    }

    void Update()
    {
        if (Moving)
            Moving = false;
    }

    void OnCollisionEnter(Collision col)
    {
        TutorialCowControl TPC = col.gameObject.GetComponent<TutorialCowControl>();
        if (TPC != null && !Moving)
        {
            //The cow must eat at a certain angle
            if (Mathf.Abs(TPC.AngleToFood) > 5)
                TPC.Death = true;
            else
            {
                TPC.Satiety += 15;
                MoveFood();
            }
        }
    }

    //Move food
    void MoveFood()
    {
        //Random position
        transform.position = new Vector3(Random.Range(-14F, 14F), 0.5F, Random.Range(-14F, 14F));
        Moving = true;
    }
}
```

Do not forget to indicate the «cow» where the «food» is:



There is the «cow» and the «food» control. Now the «cow's brain» has to be created.

How to create the perceptron.

To create the perceptron, you need to create a MonoBehaviour script and type in the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NameOfScript : MonoBehaviour
{
    public Perceptron PerceptronName = new Perceptron();
    . . .
    void Start()
    {
        PerceptronName.CreatePerceptron(1, false, true, 3, null, 2);
        . . .
    }

    void Update()
    {
        . . .
        PerceptronName.Input[0] = . . . ;
        PerceptronName.Input[1] = . . . ;
        . . .
        PerceptronName.Solution();
        . . .
        . . . = PerceptronName.Output[0];
        . . . = PerceptronName.Output[1];
        . . .
    }
    . . .
}
```

See the explanation on Page 4.

Before using `PerceptronName.Solution()` the converted input data has to be entered into the input neurons.

After using `PerceptronName.Solution()` – the output data of the output neurons has to be output and then converted.

For the task «Mission is not to die», we will create a «cow's brain» in a file with the following name **TutorialCowPerceptron.cs**. It has to be added to the «cow» object.

The «cow» has three main values that it has to take and two control values ([see Page 15](#)). Consequently, the perceptron with three input and two output neurons is needed. To make it easier, the perceptron with the negative (a minus) value is created ([see Page 4](#), `bool ActivationFunctionWithMinus`). And now the amount of hidden layers and neurons in them will depend on you and the training of ANN (I recommend you just to try to change the amount of hidden layers, the amount of neurons in them and see the result). But they are indicated in the example.

And finally, it will be convenient to change the parameters in the game mode, if we add the perceptron interface.

Do not forget, that to input data of the perceptron into the input layer, it is necessary to convert these data the way it is correctly taken by ANN. And to convert the output values the way to get the data you need.

The «cow's brain» script is the following **TutorialCowPerceptron.cs**:

```
using UnityEngine;

public class TutorialCowPerceptron : MonoBehaviour
{
    private TutorialCowControl THC;           //Cow control
    public Perceptron PCT = new Perceptron(); //Perceptron
    private PerceptronInterface PI;          //Perceptron interface

    void Start()
    {
        //Find cow control
        THC = gameObject.GetComponent<TutorialCowControl>();

        //Hidden layers and neurons
        int[] Layers = new int[2];
        Layers[0] = 9;
        Layers[1] = 9;

        //Create perceptron
        PCT.CreatePerceptron(1, false, true, 3, Layers, 2);

        //Add perceptron interface to game object & add perceptron to interface
        PI = gameObject.AddComponent<PerceptronInterface>();
        PI.PCT = PCT;
    }

    // Update is called once per frame
    void Update()
    {
        //Convert vaule
        PCT.Input[0] = THC.AngleToFood / 180F; //Work with angles. Min vaule = -180, max vaule = 180
        PCT.Input[1] = THC.DistanceToFood / 41F; //Work with distance. Max vaule = 41
        PCT.Input[2] = THC.Satiety / 50F;       //Work with satiety. Min vaule = 0, max vaule = 50
        PCT.Solution();                          //Perceptron solution

        //For this tutorial not need to convert vaule
        THC.Turn = PCT.Output[0];
        THC.Move = PCT.Output[1];
    }
}
```

Here's how the «brain» will look like with the specified parameters from the script above:



Lesson # 1. Training with a sample of tasks and responses.

To train the perceptron to solve the task «Mission is not to die» ([see Page 15](#)), a selection of the tasks is required. You can create this selection by yourself and it will take some time, or you can simply make a generator. As you know, that the «cow» will die when «fullness» will be less than 0 or more than 50, and food gives +15 to «fullness», it should be indicated that to «eat» when «cows» is «full» with values more than 35, is forbidden. Also, «cows» must not «eat», if the angle of rotation of the front side of the «cow» and the distance to food is more than 5 degrees.

A definite conclusion can be made:

Under any conditions, the «cow» has to turn to «food». If the «cow» is far from «food», then it has to move to «food» to a certain distance. If the «cow» is not «hungry» enough, then it has to wait for a certain level of «fullness». When it is «hungry», it must eat «food», when «cow» looks at «food».

The following will do:

```
Turn = AngleToFood / 180F;  
Move = 0F;  
if (DistanceToFood > 3.5F && Mathf.Abs(AngleToFood) < 45)  
    Move = 1F;  
else if (Satiety < 25 && Mathf.Abs(AngleToFood) < 5)  
    Move = 1F;
```

Create a **TutorialCowPerceptron.cs** script ([see Page 16](#)) and add it to the «cow».

Create a **CowLearning.cs** script and add it to the «cow».

```
using UnityEngine;  
  
public class CowLearning : MonoBehaviour  
{  
    void Start ()  
    {  
        . . .  
    }  
}
```

A sample of tasks and responses has to be created (the more, the better. But be careful, as the entire selection takes place in one frame of the game mode during the training). Create two two-dimensional arrays. The first dimension is for the task/response number, the second one is for each neuron of the input/output layer.

Variant 1. It can be a random task/response generator based on previous conclusions (do not forget to convert the tasks/responses):

```

. . .
float[][] Answers = new float[50][];
float[][] Tasks = new float[50][];
int i = 0;
while (i < Answers.Length)
{
    Tasks[i] = new float[3];
    if (i % 3 == 0)
    {
        Tasks[i][0] = Random.Range(-180F, 180F) / 180F;
        Tasks[i][1] = Random.Range(0F, 41F) / 41F;
        Tasks[i][2] = Random.Range(0F, 50F) / 50F;
    }
    else
    {
        Tasks[i][0] = Random.Range(-5F, 5F) / 180F;
        Tasks[i][1] = Random.Range(0F, 4F) / 41F;
        Tasks[i][2] = Random.Range(0F, 40F) / 50F;
    }
    Answers[i] = new float[2];
    Answers[i][0] = Tasks[i][0];
    Answers[i][1] = 0;
    if (Tasks[i][1] > 3.5F / 41F && Mathf.Abs(Tasks[i][0]) < 45F / 180F)
        Answers[i][1] = 1;
    else if (Tasks[i][2] < 25F / 50F && Mathf.Abs(Tasks[i][0]) < 2.5F / 180F)
        Answers[i][1] = 1;
    i++;
}
. . .

```

Variant 2. It can be an ordered task/response generator based on previous conclusions (do not forget to convert the tasks/responses):

```

. . .
int i = 0;
int c = -1;
int p = -1;
int a = 9; //number of angular variations
int d = 4; //number of distance variations
int s = 4; //number of variations of hunger
float[][] Answers = new float[a * d * s][];
float[][] Tasks = new float[a * d * s][];
while (i < Answers.Length)
{
    if (i % (a * s) == 0)
        c++;
    if (i % a == 0)
        p++;
    Tasks[i] = new float[3];
    Tasks[i][0] = ((-90F + 180F / (a - 1) * (i % a)) / (c + p + 1)) / 180F;
    Tasks[i][1] = ((41F - 41F / d * (c % d)) / (c + 1)) / 41F;
    Tasks[i][2] = (50F - 50F / s * (p % s)) / 50F;
    Answers[i] = new float[2];
    Answers[i][0] = Tasks[i][0];
    Answers[i][1] = 0;
    if (Tasks[i][1] > 3.5F / 41F && Mathf.Abs(Tasks[i][0]) < 45F / 180F)
        Answers[i][1] = 1;
    if (Tasks[i][2] < 25F / 50F && Mathf.Abs(Tasks[i][0]) < 2.5F / 180F)
        Answers[i][1] = 1;
    i++;
}
. . .

```

It will be more convenient to add a learning interface by the back propagation method:

```
    . . .
    PerceptronBackPropagationInterface PLBBPI =
gameObject.AddComponent<PerceptronBackPropagationInterface>();
    . . .
```

The created tasks/responses arrays are entered into the learning interface:

```
    . . .
    PLBBPI.Task = Tasks;
    PLBBPI.Answer = Answers;
    . . .
```

Specify «brain» (perceptron) to the interface:

```
    . . .
    PLBBPI.PCT = gameObject.GetComponent<TutorialCowPerceptron>().PCT;
}
}
```

Now you can run the game mode. If you want or need to configure the perceptron as you wish (but, do not change **Enters** and do not use **Without Minus** ([see Page 10](#)) for the following task, without changing the conversion of the variables in the perceptron and its learning) and to change the learning settings (do not play with **Learning speed**, in case of large sizes of task/response selection, it can start to lock up. I recommend you to leave = 1). Click **Learn OFF** ([see Page 11](#)).

Now your perceptron will start learning. The learning process will be considered complete when the **Max error** is less than **Desired max error** ([see Page 11](#)).

Sometimes the **Max error** will be greater than the **Desired max error** even after long learning. This is usually due to incorrectly converted input / output values, or the selection itself contains errors / inaccuracies, or because of insufficient amount of hidden layers (or neurons in them). In this case, you have to look for errors and fix them, or try to change the hidden layers.

The perceptron can be saved, and then loaded through the interface of the perceptron ([see Page 10](#)).

Lesson # 2. Learning with a «teacher».

This lesson is very similar to [lesson # 1](#), but instead of a selection one has to create a «teacher». To create the «teacher» use the algorithm from the previous lesson:

```
Turn = AngleToFood / 180F;  
Move = 0F;  
if (DistanceToFood > 3.5F && Mathf.Abs(AngleToFood) < 90)  
    Move = 1F;  
else if (Satiety < 35 && Mathf.Abs(AngleToFood) < 5)  
    Move = 1F;
```

Create a CowLearning.cs script (or rewrite it, if you have read [Lesson # 1](#)) and add it to the «cow»:

```
using UnityEngine;  
  
public class CowLearning : MonoBehaviour  
{  
    . . .
```

The «teacher» will need access to the variables of the «cow»:

```
. . .  
private TutorialCowControl TCC;  
. . .
```

The «teacher» will give only one selection of responses, as the task will go from the control of the «cow». Create a two-dimensional array of selection responses. The first dimension is = 1 (only one selection), the second one for each neuron of the output layer.

```
. . .  
public float[][] Answer = new float[1][];  
. . .
```

It will also be convenient to add a learning interface by the back propagation method.

```
. . .  
private PerceptronBackPropagationInterface PLBBPI;  
void Start ()  
{  
    PLBBPI = gameObject.AddComponent<PerceptronBackPropagationInterface>();  
    . . .
```

Get access to the cow variables and specify the size of the selection array in the second dimension:

```
. . .  
TCC = gameObject.GetComponent<TutorialCowControl>();  
Answer[0] = new float[2];  
. . .
```

Specify «brain» (perceptron) interface:

```
    . . .
    PLBBPI.PCT = gameObject.GetComponent<TutorialCowPerceptron>().PCT;
}
```

Let the «teacher» work only when the learning mode is enabled in the interface:

```
    . . .
    void Update()
    {
        if (PLBBPI.Learn)
        {
            . . .
        }
    }
}
```

Now the «teacher» specifies the «brain», what to learn:

```
    . . .
    Answer[0][0] = TCC.AngleToFood / 180F;

    if (TCC.DistanceToFood > 3.5F && Mathf.Abs(TCC.AngleToFood) < 90)
        Answer[0][1] = 1;
    else if (TCC.Satiety < 35 && TCC.DistanceToFood > 0 && Mathf.Abs(TCC.AngleToFood) < 5)
        Answer[0][1] = 1;
    else if (TCC.Move > 0)
        Answer[0][1] = 0;

    PLBBPI.Answer = Answer;
}
}
```

Now you can run the game mode. You can set the perceptron as you wish or need to (but do not change **Enters** and do not use **Without Minus** ([see Page 10](#)) for this task, without changing the conversion of the variables in the perceptron and its learning) and change the learning settings (**Learning speed**, and **Desired max error** will not affect learning).

Click **Learn OFF** ([see Page 11](#)).

Recommendation: Use low **Learning rate** (0.01 - 0.1).

Now your perceptron is starting learning. But the learning process can take a long time. Only you decide when the learning process is finished, or you can add a parameter that specifies under which conditions the perceptron is taught (for example: a life time, a length of distance traveled, or a number of actions performed, etc.).

You can save the perceptron settings, and then load them through the interface of the perceptron ([see Page 10](#)).

Lesson # 3. Training by the random generation method.

For this lesson, you will need to enter «longevity» into the «cow» script ([see Page 16](#))

```
using UnityEngine;

public class TutorialCowControl : MonoBehaviour
{
    . . .
    public float LifeTime = 0;
    . . .
}
```

This is due to the fact that when learning by the random generation method, one needs to track a better «clone» in the generation.

In addition, it is better to increase «longevity», when the correct actions are done, and/or to reduce the «longevity», when the actions are done incorrectly.

Let the «longevity» increase over time. And let the «longevity» reduce, when the «cow» looks at «food» incorrectly.

```
. . .
void Update()
{
    . . .
    LifeTime += Time.deltaTime - (Mathf.Abs(AngleToFood) / 180F) * Time.deltaTime;
}
}
```

We will also reset the «longevity», when the «cow dies».

```
. . .
void Update()
{
    . . .
    if (Death)
    {
        . . .
        LifeTime = 0;
    }
    . . .
}
```

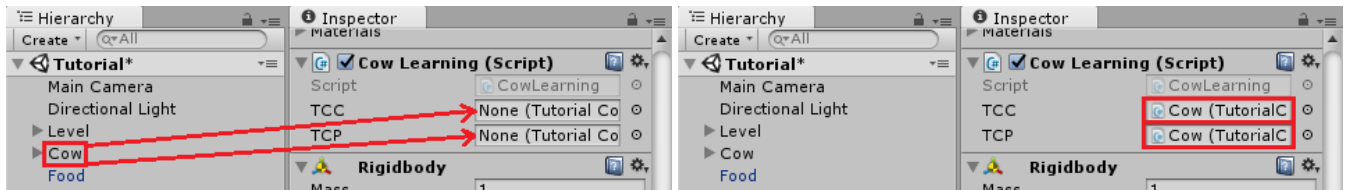
Create a **CowLearning.cs** script (or rewrite it, if you have read [Lesson # 1](#) or [Lesson # 2](#)) and add it to any object:

```
using UnityEngine;

public class CowLearning : MonoBehaviour
{
    . . .
}
```

Then, you need to specify the «cow» and the «cow's brain» control scripts:

```
...  
public TutorialCowControl TCC;  
public TutorialCowPerceptron TCP;  
...
```



Add an interface of the random generation method:

```
...  
private PerceptronRandomGenerationInterface PRGI;  
  
void Start()  
{  
    PRGI = gameObject.AddComponent<PerceptronRandomGenerationInterface>();  
    ...  
}
```

Specify the interface the perceptron that is needed to be taught:

```
...  
PRGI.PCT = TCP.PCT;  
...
```

To the random generation method add the proper information about the «cow»:

```
...  
    PRGI.PLBRG.StudentData(TCP.gameObject, TCP, «PCT», TCC, «Death», «LifeTime»);  
}  
}
```

See the explanation on Page 6.

Also, for more effective training, to the **TutorialFood.cs** add bonuses for «children» when they are right «eating food»:

```
...  
void OnCollisionEnter(Collision col)  
{  
    ...  
    if (TCC != null && !Moving)  
    {  
        ...  
        else  
        {  
            ...  
            TCC.LifeTime += 15;  
        }  
    }  
}  
...  
}
```

Now you can run the game mode. You can configure the perceptron as you wish or need to (But do not change **Enters** and do not use **Without Minus** ([see Page 10](#)) for this task, without changing the conversion of the variables in the perceptron and its learning) and change the training settings ([see Page 12](#)). Click **Learn OFF** ([see Page 12](#)).

Now your perceptron is starting learning. You can notice the effect of learning in the first generation. But the learning process can take a quite long time. The learning process can be considered completed if there is a significant difference between **Best Generation** and **Generation** ([see Page 12](#)). To stop the learning process, and to transfer the weight units of the learning to the perceptron, that is being learned, click **Learn ON**.

Change the parameters of the settings for better understanding the interface of the random generation method.

The perceptron configuration can be saved, and then loaded through the interface of the perceptron ([see Page 10](#)).

[How to save and load the perceptron settings.](#)

We have already reviewed how to create the perceptron ([see Page 18](#)). [See Page 10](#) to find out how to save/load settings and weight units with the help of the interface. Now we will consider how to save and load its settings and weight units in the scripts.

To save the settings and weight units of the perceptron, use the following command:

```
...  
PerceptronName.Save(«SaveName»);  
...
```

SaveName is the filename for saving.

The configuration file and the weight units will be saved to the following address:

```
Application.dataPath + «/ANN/PerceptronStatic/» + PerceptronFile + «.ann»
```

PerceptronFile is the filename for saving/loading ([see Page 4](#)).

Use the `PerceptronName.Load («LoadName»)` command instead of using the command `PerceptronName.CreatePerceptron (...)` to load the perceptron configuration and weight units:

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class NameOfScript : MonoBehaviour  
{  
    public Perceptron PerceptronName = new Perceptron();  
    ...  
    void Start()  
    {  
        PerceptronName.Load(«LoadName»);  
        ...  
    }  
}
```

LoadName is the filename for the loading.

How to create the ANN.

To create ANN, you need to create a MonoBehaviour script and type in the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NameOfScript : MonoBehaviour
{
    public ANN AnnName = new ANN();
    . . .
    void Start()
    {
        AnnName.Create(3, 2);
        . . .
    }

    void Update()
    {
        . . .
        AnnName.Input[0] = . . . ;
        AnnName.Input[1] = . . . ;
        . . .
        AnnName.Solution();
        . . .
        . . . = AnnName.Output[0];
        . . . = AnnName.Output[1];
        . . .
    }
    . . .
}
```

See the explanation on Page 4.

Before using AnnName.Solution() the converted input data has to be entered into the input neurons.

After using AnnName.Solution() – the output data of the output neurons has to be output and then converted.

For the task "Mission is not to die", we will create a "cow's brain" in a file with the following name **Lesson4TutorialCow.cs**. It has to be added to the "cow" object.

The "cow" has three main values that it has to take and two control values ([see Page 15](#)). Consequently, the ANN with three input and two output neurons is needed. To make it easier, the ANN with the negative (a minus) value is created ([see Page 4](#), bool AFWM). And now the amount of hidden neurons and connections with them will depend on the training of ANN.

And finally, it will be convenient to change the parameters in the game mode, if we add the ANN interface.

Do not forget, that to input data of ANN into the input layer, it is necessary to convert these data the way it is correctly taken by ANN. And to convert the output values the way to get the data you need.

The "cow's brain" script is the following **Lesson4TutorialCow.cs**:

```
using UnityEngine;
public class Lesson4TutorialCow : MonoBehaviour
{
    private TutorialCowControl THC;    //Cow control
    public ANN Ann = new ANN();        //ANN
    private ANNInterface NI;           //ANN interface

    void Start()
    {
        //Find cow control
        THC = gameObject.GetComponent<TutorialCowControl>();

        //Create ANN
        Ann.Create(3, 2);

        //Add ANN interface to game object & add ANN to interface
        NI = gameObject.AddComponent<ANNInterface>();
        NI.Ann = Ann;
    }

    void Update()
    {
        //Convert vaule
        Ann.Input[0] = THC.AngleToFood / 180F;    //Work with angles. Min vaule = -180, max vaule = 180
        Ann.Input[1] = THC.DistanceToFood / 41F;   //Work with distance. Max vaule = 41
        Ann.Input[2] = THC.Satiety / 50F;          //Work with satiety. Min vaule = 0, max vaule = 50
        Ann.Solution();                             //ANN solution
        //For this tutorial not need to convert vaule
        THC.Turn = Ann.Output[0];
        THC.Move = Ann.Output[1];
    }
}
```

Here's how the "brain" will look like with the specified parameters from the script above:



And YES - it's empty here.

Lesson # 4. Training by NEAT.

For this lesson, you will need to enter "longevity" into the "cow" script ([see Page 16](#))

```
using UnityEngine;

public class TutorialCowControl : MonoBehaviour
{
    . . .
    public float LifeTime = 0;
    . . .
}
```

This is due to the fact that when learning needs to track a better "clone" in the generation.

In addition, it is better to increase "longevity", when the correct actions are done, and/or to reduce the "longevity", when the actions are done incorrectly.

Let the "longevity" increase over time. And let the "longevity" reduce, when the "cow" looks at "food" incorrectly.

```
. . .
void Update()
{
    . . .
    LifeTime += Time.deltaTime - (Mathf.Abs(AngleToFood) / 180F) * Time.deltaTime;
}
}
```

We will also reset the "longevity", when the "cow dies". This is not a mandatory action (the script itself cancels "longevity").

```
. . .
void Update()
{
    . . .
    if (Death)
    {
        . . .
        LifeTime = 0;
    }
    . . .
}
```

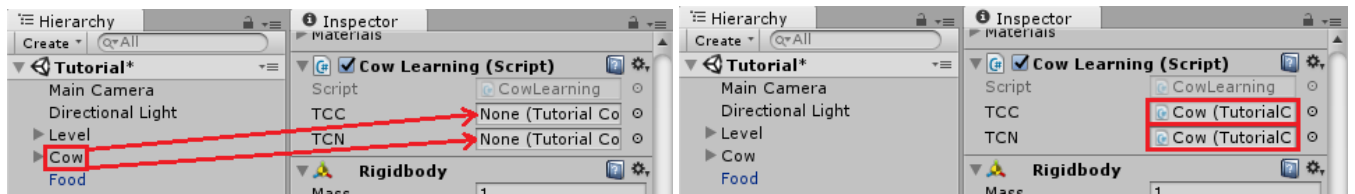
Create a **Lesson4CowLearning.cs** script and add it to any object:

```
using UnityEngine;

public class Lesson4CowLearning : MonoBehaviour
{
    . . .
}
```

Then, you need to specify the "cow" and the "cow's brain" control scripts:

```
. . .
public TutorialCowControl TCC;
public Lesson4TutorialCow TCN;
. . .
```



Add a learning interface:

```

. . .
private ANNLearnByNEATInterface NLI;

void Start()
{
    NLI = gameObject.AddComponent<ANNLearnByNEATInterface>();
    . . .

```

Specify the interface ANN that is needed to be taught:

```

. . .
NLI.PCT = TCP.Ann;
. . .

```

To the random generation method add the proper information about the "cow":

```

. . .
    NLI.NL.StudentData(TCP.gameObject, TCP, "Ann", TCC, "Death", "LifeTime");
}
}

```

See the explanation on Page 9.

Also, for more effective training, to the **TutorialFood.cs** add bonuses for "children" when they are right "eating food":

```

. . .
void OnCollisionEnter(Collision col)
{
    . . .
    if (TCC != null && !Moving)
    {
        . . .
        else
        {
            . . .
            TCC.LifeTime += 15;
        }
    }
}
. . .

```

Now you can run the game mode. Do not use **Without Minus** (see Page 13) for this task. Change the training settings according to your desire (see Page 14). Click **Learn OFF** (see Page 14).

Now your ANN is starting learning. You can notice the effect of learning in the first generation. But the learning process can take a quite long time. The learning process can be considered completed if there is a significant difference between **Best Generation** and **Generation** (see Page 14). To stop the learning process, and to transfer the weight units of the learning to the ANN, that is being learned, click **Learn ON**. The ANN configuration can be saved, and then loaded through the interface of the ANN (see Page 13).

How to save and load the ANN settings.

We have already reviewed how to create the ANN ([see Page 28](#)). [See Page 13](#) to find out how to save/load settings and weight units with the help of the interface. Now we will consider how to save and load its settings and weight units in the scripts.

To save the settings and weight units of the ANN, use the following command:

```
. . .  
AnnName.Save("SaveName");  
. . .
```

SaveName is the filename for saving.

The configuration file and the weight units will be saved to the following address:

```
Application.dataPath + "/ANN/ANN/" + ANNFile + ".ann"
```

ANNFile is the filename for saving/loading ([see Page 7](#)).

Use the AnnName.Load ("LoadName") command instead of using the command AnnName.Create (...) to load the ANN configuration and weight units:

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class NameOfScript : MonoBehaviour  
{  
    public ANN AnnName = new ANN();  
    . . .  
    void Start()  
    {  
        AnnName.Load("LoadName");  
        . . .  
    }  
}
```

LoadName is the filename for the loading.

The conclusion.

I hope you have enjoyed my work. I tried to ease the task of creating, learning, saving and loading the perceptron setting and its weights.

If you have any questions, or there are any suggestions for improving this work, write to VirtualSUN13@gmail.com. I'll be happy to answer.

Special thanks to Leonid Tereshonkov for a moral support and Yulia Pavlovych for helping in translation the document. ☺

PS: And do not forget to leave your feedback about the project (<http://u3d.as/1rPi>). I would be very grateful to you.

Contacts.

YouTube: <https://www.youtube.com/channel/UClblqhEzoATg-Jvk0AviVlw>

Unity Connect: <https://connect.unity.com/u/sergey-voroshilov-vladimirovich>

Twitter: https://twitter.com/sun_virtual

Instagram: <https://www.instagram.com/virtualsun13/>

Facebook group: <https://www.facebook.com/groups/VirtualSTAR/>

Facebook page: <https://www.facebook.com/VirtualSTAR13/>

Forum: <https://forum.unity.com/threads/released-ann-perceptron.558868/>

E-mail: VirtualSUN13@gmail.com