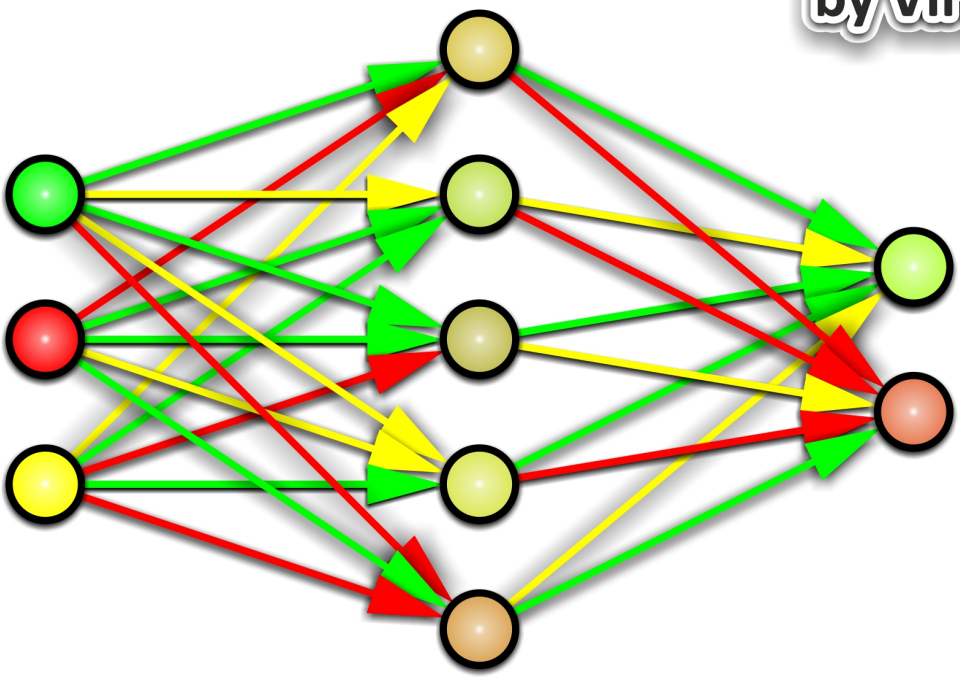
ARTIFICIAL NEURAL NETWORK
PERCEPTRON
by VirtualSUN



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.

2018

CONTENT

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, storage, and loading of ANNs during the learning process. This is the interface for the perceptron in the game mode.

PerceptronVisualization.cs is a non MonoBehaviour script for the perceptron visualization. It shows all the layers, neurons, and units among the ANN neurons. It is used by the **PerceptronInterface.cs** script, but you can also use it in your own scripts in case you need it.

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.

InterfaceGUI.cs is a non MonoBehaviour script for interfaces' scripts.

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

A detailed description of the main scripts.

public class PerceptronLernByBackPropagation – is a non MonoBehaviour script for an artificial neural network (ANN). Type of the perceptron.

The main variables of the Perceptron.cs script	
public float AFS	A size of the activation function.
public bool B	If it is “true”, 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.
public bool AFWM	If it is “true”, then all the neurons take the values from -1 to 1. This also applies to incoming and outgoing neurons. If it is “false” then all the neurons take values from 0 to 1.
public float[] Input	The input layer of the ANN. The size of the array indicates the number of the input values, with a bias neuron (if B == true, the last neuron is always equal to 1).
public int[] NIHL	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 B == true, the last neuron of each layer is always equal to 1).
public float[] Output	Output layer of the ANN. The size of the array indicates the number of the output values. It never contains a bias neuron.
public float[][] Neuron	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.
public float[][][] NeuronWeight	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
public void CreatePerceptron		Creation of perceptron with the help of parameters.
	float ActivationFunctionScale	A Size of an activation function.
	bool Bias	If it is "true", 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.
	bool ActivationFunctionWithMinus	If it is "true", then all the neurons’ values are from -1 to 1. It also applies to the input and output neurons. If it is "false", the neurons take values from 0 to 1.
	int NumberOfInputs	The amount of the input neurons, without a bias neuron.
	int[] NumberOfNeuronInHidenLayers	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.
	int NumbersOfOutputs	The number of output neurons.
public void Load		Load the perceptron from the file.
	string PerceptronFile	Filename for loading.
public void PerceptronSolution		Perceptron solution.
private float Sumator		The total sum of all the values of the neurons multiplied by their weight.
	float[] Neuron	The neuron of a certain layer.
	float[] NeuronWeight	The weights of the same layer as the neuron.
private float ActivationFunction		The neurons activation function.
	float Sum	The total sum of all the values of neurons multiplied by their weight.
private void CreatingNeurons		Creation of the neurons and their units with each other.
	StreamReader SR	The specified stream from the load file. Use "null" if the file is not used.
public void Save		Save perceptron parameters to a file.
	string PerceptronFile	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 PerceptronLernByBackPropagation.cs script	
public int LearningSpeed	The number of learning steps per a frame of the game mode.
public int LearnIteration	A counter of learning steps.
public float LearningRate	The power of a change of weight units during the learning process. It affects the speed and quality of the learning.
public float DesiredMaxError	It specifies the maximum difference between the correct response and the ANN response.
public float MaxError	The maximum difference between the correct response and the ANN response.
public bool Learned	If the difference between the correct response and the ANN response is less than DesiredMaxError, then the perceptron is considered to be trained.
public bool ShuffleSamples	If it is "true", then samples of tasks and responses will be shuffled during the training.

The Command of the Script	The Variables of the Command	Description
public void Learn		The first variation the command. Teaching of the percepton by the back propagation method with the content of a certain number of tasks with responses. public void Learn can use ShuffleSamples.
	Perceptron PCT	The perceptron is to be taught.
	float[][] Task	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.
	float[][] Answer	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.
public void Learn		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 LearningSpeed, DesiredMaxError and ShuffleSamples. It is recommended to use DesiredMaxError = 0.
	Perceptron PCT	The perceptron is to be taught.
	float[] Task	The array of one task for each input neuron of the perceptron.
	float[] Answer	The array of one response for each output neuron of the perceptron.
public void Learn		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 LearningSpeed, DesiredMaxError and ShuffleSamples. It is recommended to use DesiredMaxError = 0.
	Perceptron PCT	The perceptron is to be taught.
	float[] Answer	The array of one response for each output neuron of the perceptron.
private void ShufflingSamples		Shuffling of the tasks and responses samples.
	float[][] Task	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.
	float[][] Answer	The 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.
public void ModificateStartWeights		Modification of the weight units of the perceptron before learning.
	Perceptron PCT	The perceptron is to be taught.
	bool MSW	If it is "true", then the weight units are modified by a special algorithm. If it is "false", then all the weight units are generated randomly in the range from -0.5 to 0.5.
private float DX		Searching of the derivative from the value of a particular neuron.
	float X	The value of a neuron.
	float ActivationFunctionScale	The size of the activation function of the perceptron.
	bool ActivationFunctionWithMinus	If the perceptron uses a negative value.

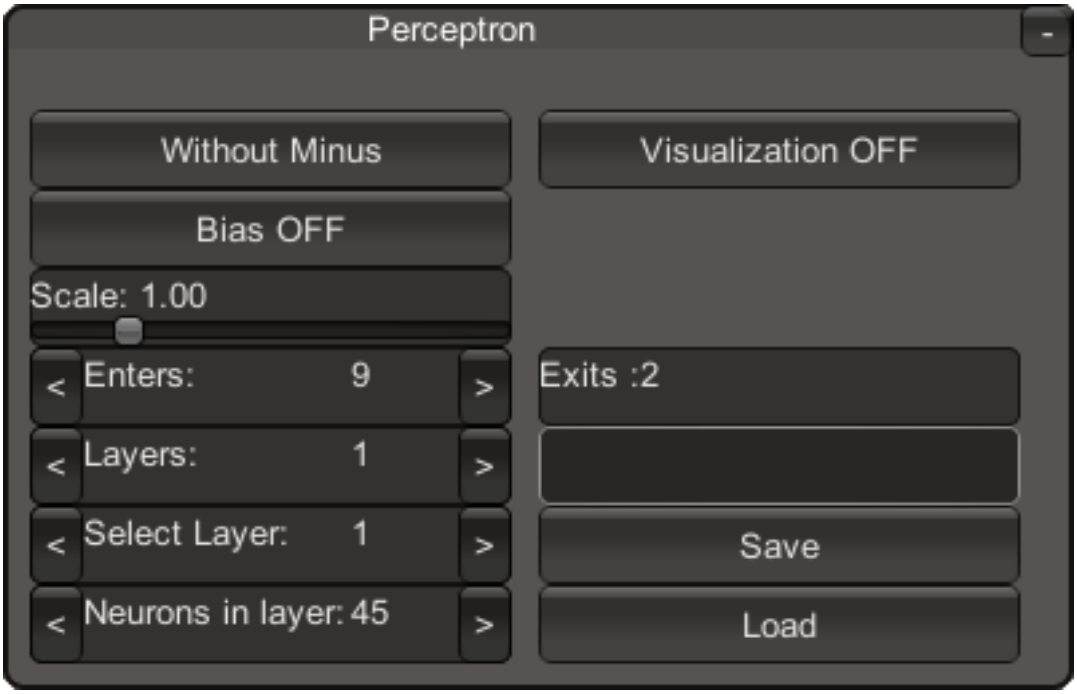
public class PerceptronLernByRandomGeneration is a non MonoBehaviour script for teaching the perceptron by the random generation method of a certain number of "clones" of the object is to be taught.

The main variables of the PerceptronLernByRandomGeneration.cs script	
public int AmountOfChildren	The number of "children" in an each generation.
public int BestGeneration	The best generation at the moment.
public int Generation	The total number of generations.
public int ChildrenInGeneration	The number of "children" in the last generation.
public float BestLongevity	The best "longevity" at the moment.
public float ChildrenDifference	The difference of the weight units among the generations.
public float ChildrenDifferenceAfterEffects	The difference of weight units with the affect coefficient among the generations.
public bool ChildrenGradient	If it is "true", then the command smoothes the difference of weight units linearly among the "children" in the generation.
public float 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.
public float 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.
public float Chance	A chance to choose the better from the worse, current generation. It changes because of the parameter public float ChanceCoefficient with each new generation.
public float 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.

The Command of the Script	The Variables of the Command	Description
public void StudentData		A data collection for the learning.
	GameObject Student	The main game object (GameObject), that has to learn.
	Object HereIsANN	A script, that contains the perceptron.
	string PerceptronName	The name of the variable (Perceptron), that was given to the perceptron in the script, which contains the perceptron (HereIsANN).
	Object StudentControls	The control script of main game object.
	string StudentCrash	The name of the variable (bool), that was given to the reason for the "crash" of the game object in the control script (StudentControls).
	string StudentLife	The name of the variable (float), that was given to the "longevity" of the game object in the control script (StudentControls).
public void Learn		Teaching the perceptron by the random generation method.
	Perceptron PCT	The perceptron is to be taught.
public void 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.
	Perceptron PCT	The perceptron is to be taught.
public void Reset		Reset learning information.

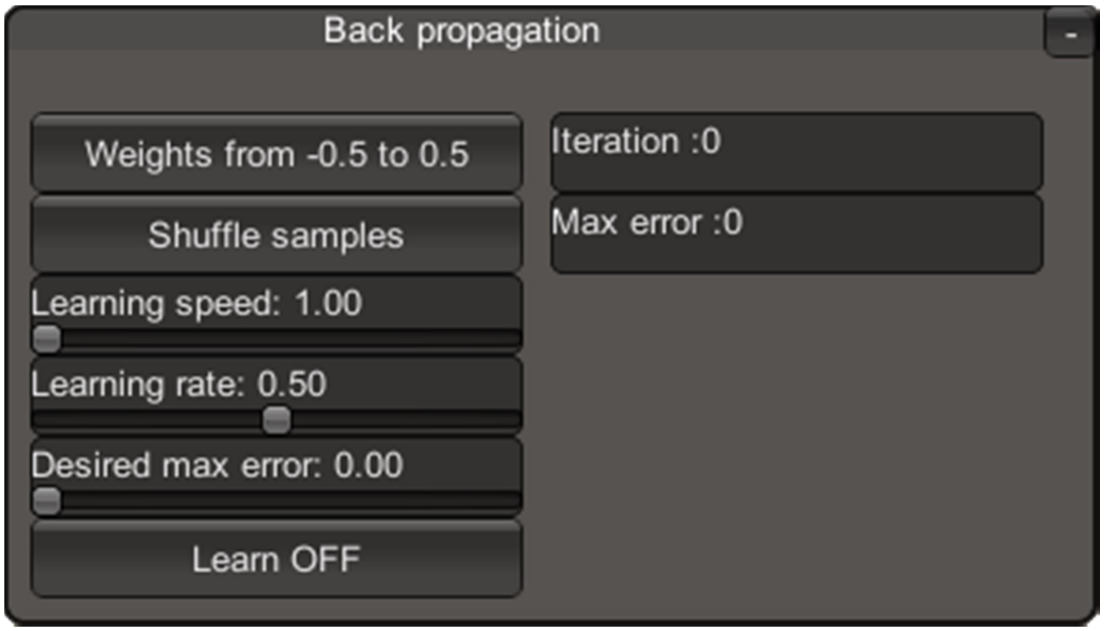
Description of the auxiliary interfaces.

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.



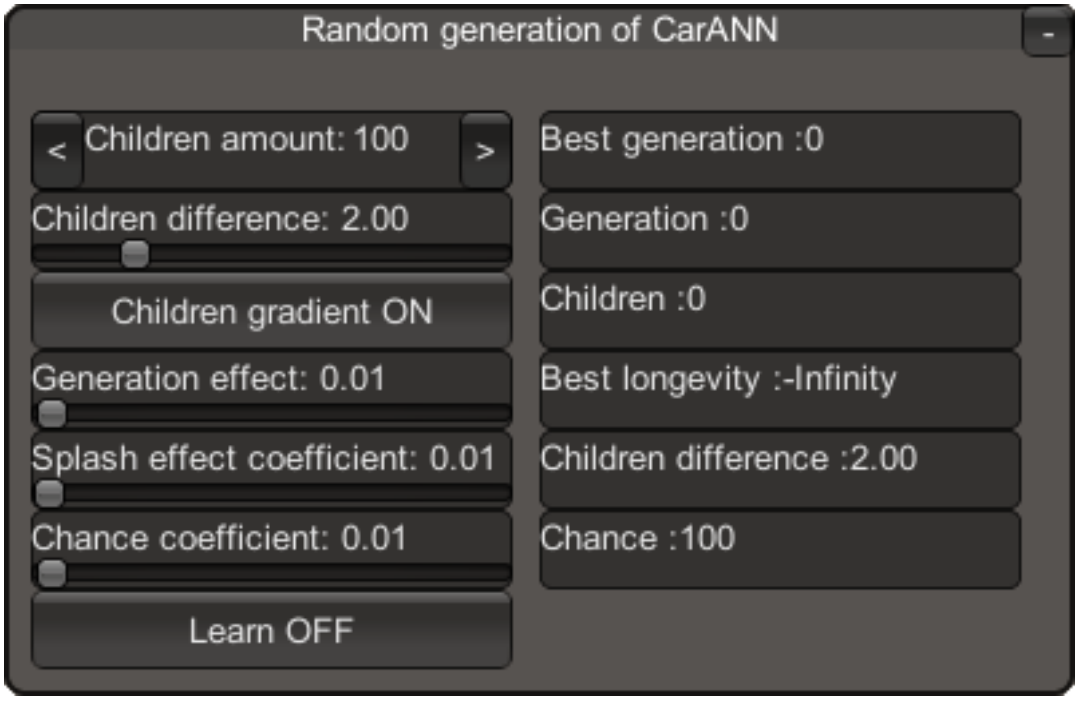
Without Minus With Minus	If "With Minus" is shown, then all the neurons' values are from -1 to 1. This also applies to the input and output neurons. If "Without Minus" is shown, then all the neurons' values are from 0 to 1. It controls the parameter public bool AFWM.
Bias OFF Bias ON	If "Bias ON" is 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 public bool B.
Scale	The scale of the activation function. The controls the parameter public float AFS.
Enters	It indicates the number of neurons in the input layer, the bias neuron is not taken into consideration. It affects the following arrays float[] Input, public float[][] Neuron and public float[][][] NeuronWeight.
Layers	Layers It indicates the number of the hidden layers. It affects the following arrays public int[] NIHL, public float[][] Neuron and public float[][][] NeuronWeight.
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 public int[] NIHL, public float[][] Neuron and public float[][][] NeuronWeight.
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.
Exits	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 public void Save.
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 public void Load.

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 public void ModificateStartWeights.
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 public bool ShuffleSamples.
Learning speed	It indicates the number of training steps per frame of the game mode. It controls the parameter public int LearningSpeed.
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 public float LearningRate.
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 public float DesiredMaxError.
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 public void Learn.
Iteration	It shows the amount of training steps. It gets data from the parameter public int LearnIteration.
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 public float MaxError.

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.



Children amount	The amount of "children" in each generation. It controls the parameter public int AmountOfChildren.
Children difference	The difference of weight units among the generations. It controls the parameter public float ChildrenDifference.
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 public float GenerationEffect.
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 public float GenerationSplashEffect.
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 public float ChanceCoefficient.
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 public void Learn. If "Learn OFF" is used and the training took place, then it stops the training. It uses the command public void StopLearn.
Best Generation	It shows the best generation number at the very moment. It gets the data from the parameter public int BestGeneration.
Generation	It shows what generation is like at the moment. It gets the data from the parameter public int Generation.
Children	It shows the number of "children" in the current generation. It gets the data from parameter public int ChildrenInGeneration.
Best longevity	It shows a better longevity at the moment. It gets the data from the parameter public float BestLongevity.
Children difference	The difference of weight units among the generations with the coefficients of affect. It gets the data from the parameter public float ChildrenDifferenceAfterEffects.
Chance	It shows the chance of choosing the current worse generation. It changes with each new generation. It gets the data from the parameter public float Chance.

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 1, "public bool AFWM"). 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 them depends on you. 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. 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 this perceptron.

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.

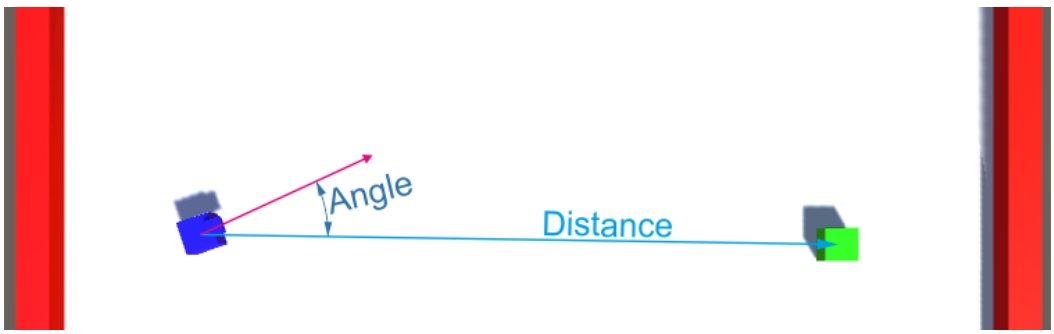
The Tutorial folder has already prepared scripts and a scene for the task.

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:**

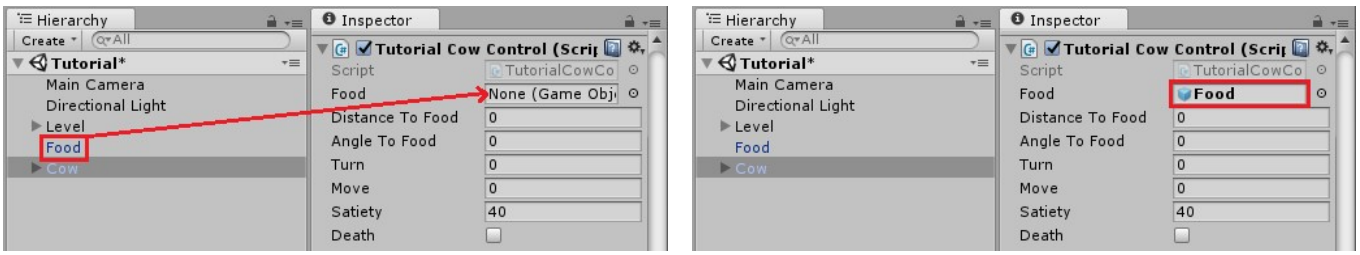
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

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)

    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;
    }
}
```

"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 System.Collections;
using System.Collections.Generic;
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();
            }
        }
    }

    void MoveFood()          //Move food
    {
        //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 10). 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.

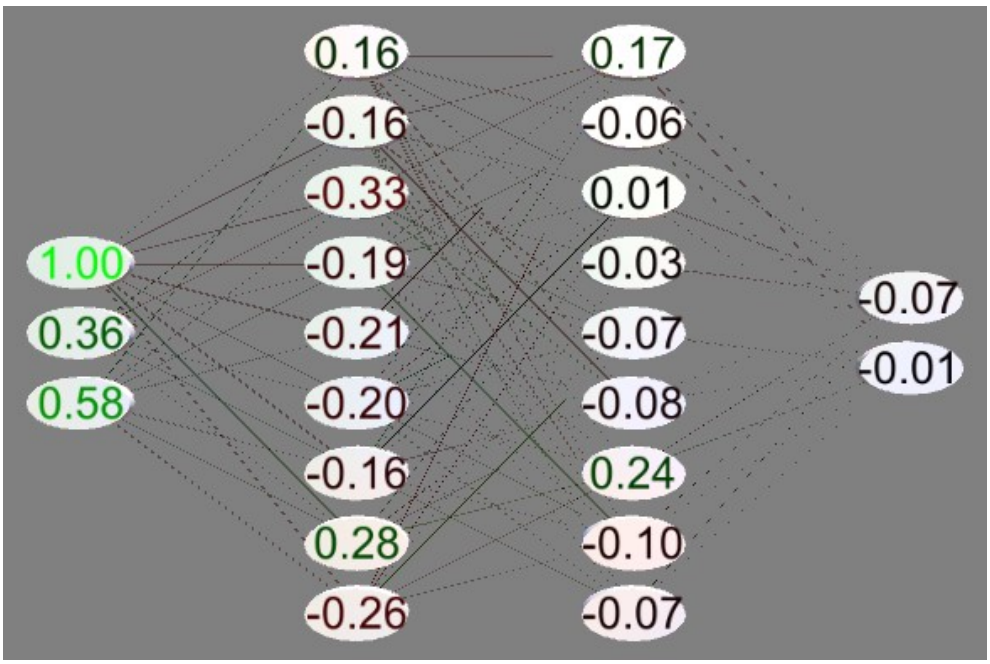
And finally, it will be convenient to change the parameters in the game mode, if we add the perceptron interface.

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 System.Collections;
using System.Collections.Generic;
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>();

        //Hiden 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 10 and 12), 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 knowing, 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 14) and add it to the "cow".

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

```
using System.Collections;
using System.Collections.Generic;
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 be 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 7) 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 8).

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 8).

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 7).

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 System.Collections;
using System.Collections.Generic;
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 7) 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 8).

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 7).

Lesson # 3. Training by the random generation method.

For this lesson, you will need to enter "longevity" into the "cow" script (see Page 11)

```
using System.Collections;
using System.Collections.Generic;
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".

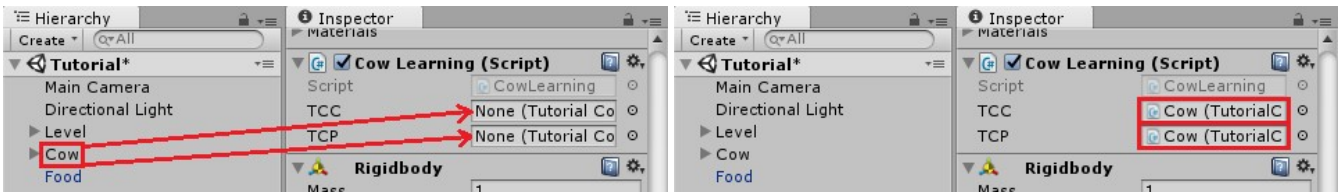
```
    . . .
    void Update()
    {
        . . .
        if (Death)
        {
            . . .
            LifeTime = 0;
        }
        . . .
```

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

```
using System.Collections;
using System.Collections.Generic;
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 description on Page 6.

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 7) for this task, without changing the conversion of the variables in the perceptron and its learning) and change the training settings (see Page 9) . Click Learn OFF (see Page 9).

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 9). 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 7).

How to save and load the perceptron settings.

We have already reviewed how to create the perceptron (see Page 13). See Page 7 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.

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 weight.

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 translation the document. ☺