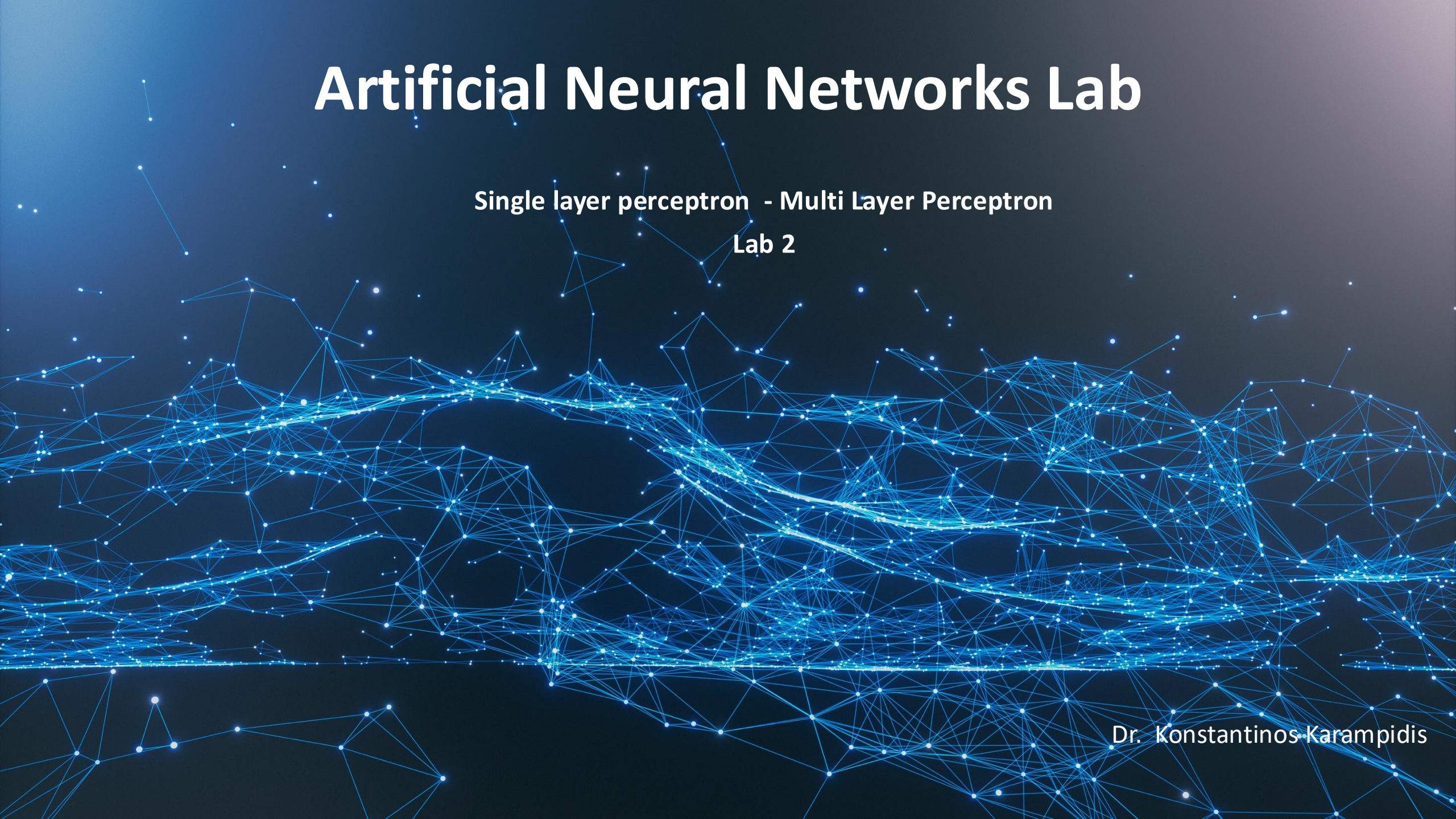
Artificial Neural Networks Lab

Single layer perceptron - Multi Layer Perceptron

Lab 2

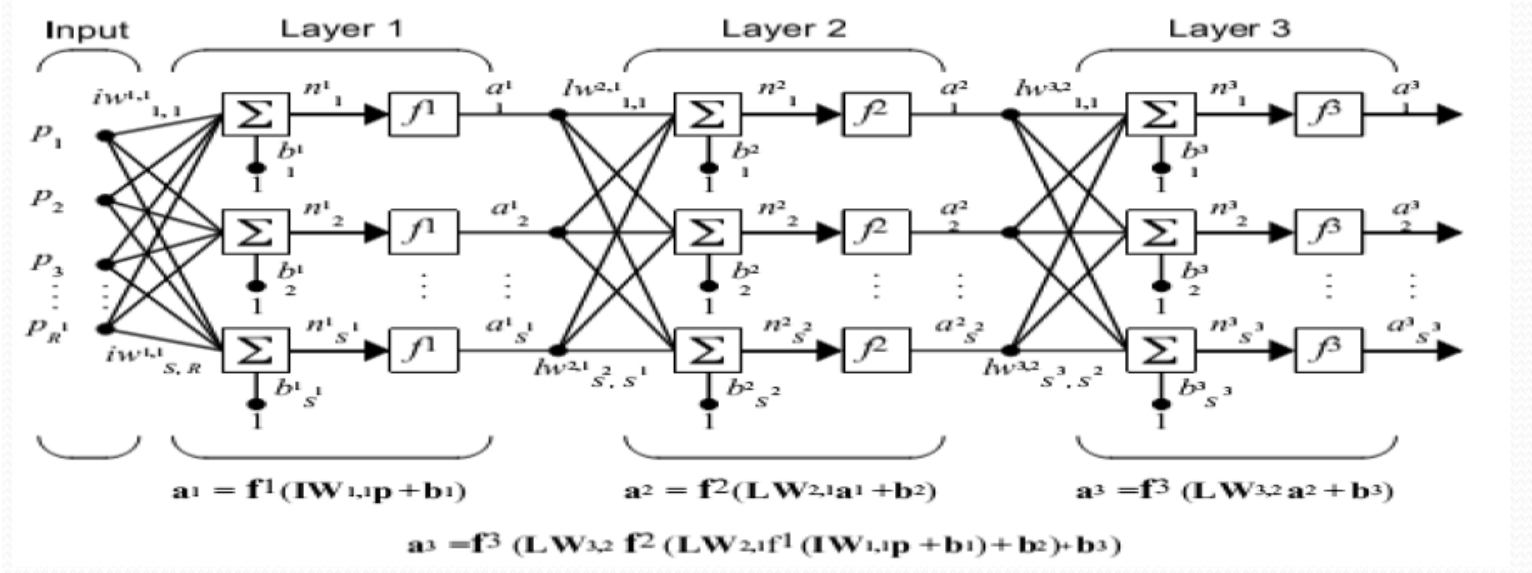


Dr. Konstantinos Karampidis

Network's architecture

When referring to the network's architecture we mean:

- The number of network's layers
- The number of neurons per layer
- How the neurons are connected (full interconnection or not)
- The activation functions



NeuroLab

- **NeuroLab** - a library of basic neural networks algorithms with flexible network configurations and learning algorithms for Python. Its interface is similar to the package of Neural Network Toolbox (NNT) of MATLAB© .

- The library is based on the package numpy (http://numpy.scipy.org) and some learning algorithms used in scipy.optimize (http://scipy.org).

Features

- Pure python + numpy
- API like Neural Network Toolbox (NNT) from MATLAB
- Interface to use trained algorithms from scipy.optimize
- Flexible network configurations and learning algorithms. You may change train, error, initialization and activation functions
- Unlimited number of neural layers and number of neurons in layers
- Variety of supported types of Artificial Neural Network and learning algorithms

Supported neural networks types

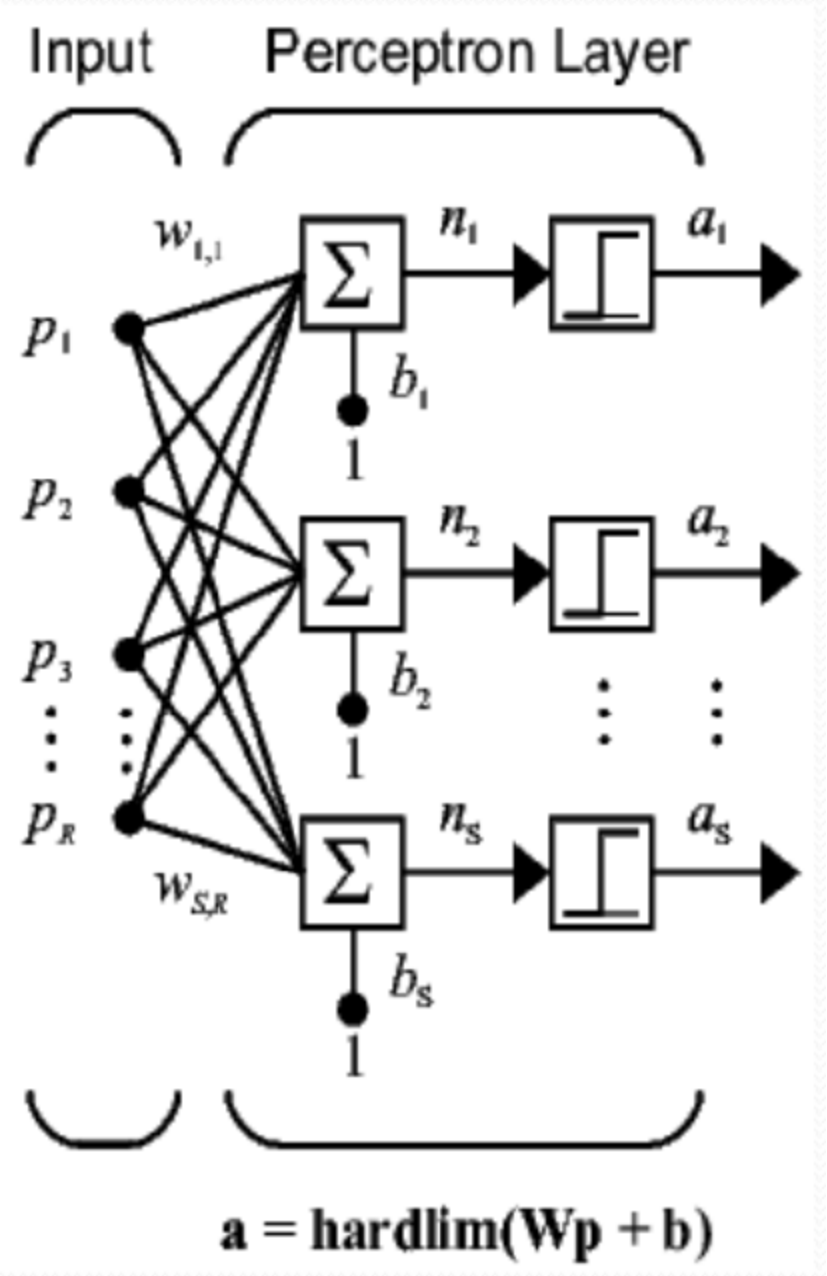
- Single layer perceptron
- Multilayer feed forward perceptron
- Competing layer (Kohonen Layer)
- Learning Vector Quantization (LVQ)
- Elman Recurrent network
- Hopfield Recurrent network
- Hemming Recurrent network

Network Type	Function	Count of layers	Support train fcn	Error fcn
Single-layer perceptron	newp	1	train_delta	SSE
Multi-layer perceptron	newff	>=1	train_gd, train_gdm, train_gda, train_gdx, train_rprop, train_bfgs*, train_cg	SSE
Competitive layer	newc	1	train_wta, train_cwta*	SAE
LVQ	newlvq	2	train_lvq	MSE
Elman	newelm	>=1	train_gdx	MSE
Hopield	newhop	1	None	None
Hemming	newhem	2	None	None

Note: * - default function

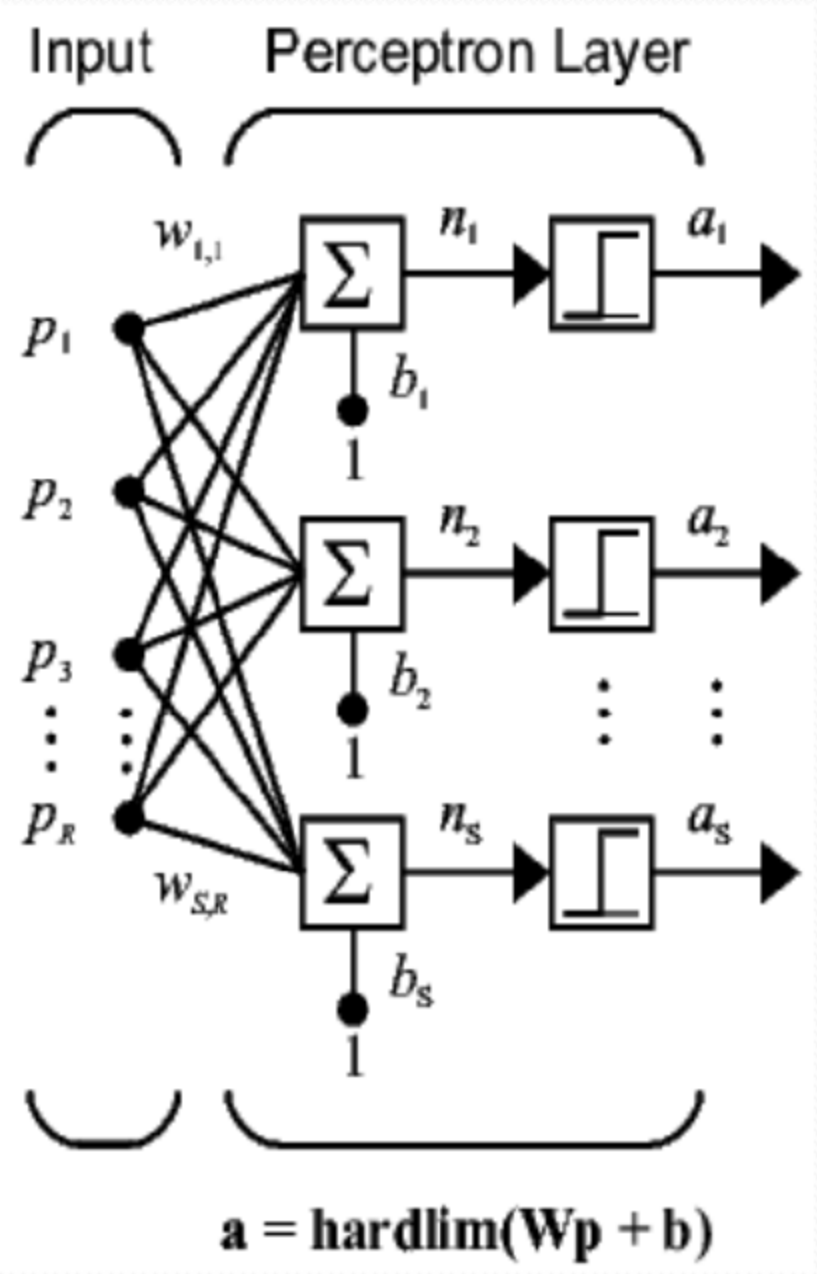
Network's Architecture

The **Perceptron** network consists of a single layer (theoretically with any number of neurons)



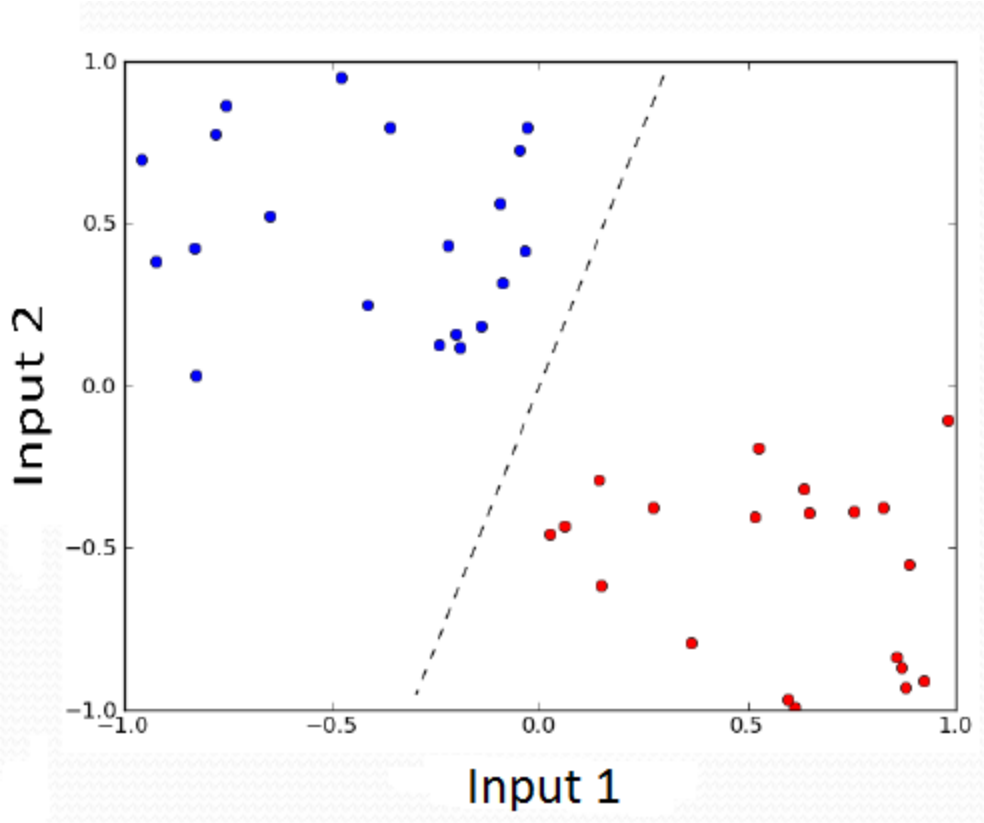
Network's Architecture

- Each neuron implements the hard limit activation function.
- Recall that the hard limit function's output is zero (0) when the input is negative or one (1) when the input is zero or a positive number.
- Between the neurons layer and the input vector there is a full connection with the corresponding weights.
- Each neuron has also as an input, a bias.



Linear Separable Problems

- Because the network consists of only one layer, it's not applicable to a wide range of pattern recognition problems.
- More specifically, the network is capable of dealing with linearly separable problems.
- Practically, this means that if for example we place all the possible inputs in a two-dimensional space, we can linearly separate the inputs into the corresponding outputs (patterns).



Perceptron Networks (single layer)

- They allow us to "predict" results for phenomena that we do not know exactly "how they work".
- We have set up a large set of data (training patterns) that can train a neural network in order to examine the behavior of the phenomenon we want to study
- Thus, the neural network responds (with relative precision) to the effect that the phenomenon would have on an unseen input.

Function newp

To create a single layer perceptron we use the function **newp** from neurolab library.

Syntax:

```
neurolab.net.newp(minmax, cn, transf=<neurolab.trans.HardLim)
```

Parameters:

- **minmax:** list of list, the outer list is the number of input neurons, inner lists must contain 2 elements: min and maxRange of each input value
- **cn:** integer, number of output neurons
- **transf:** activation function (default HardLim)

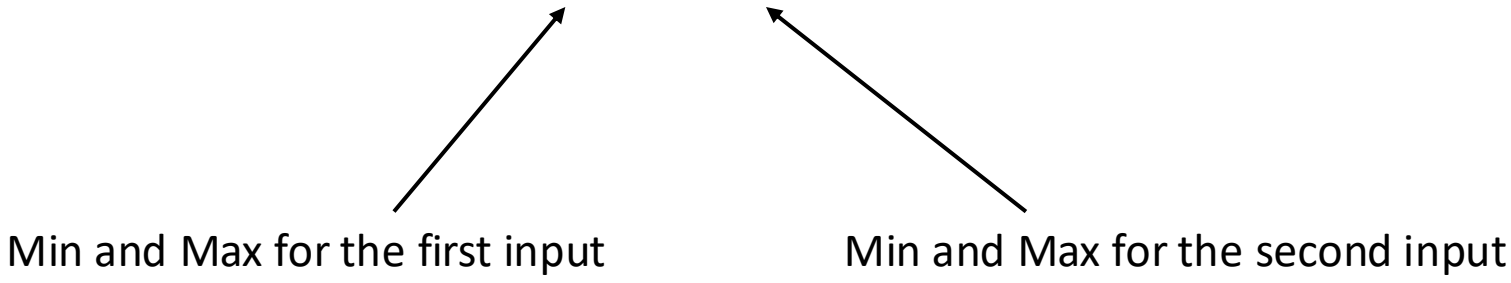
Returns:

- **net:** Net

Example 1

- We want to create a network with 2 inputs and 10 neurons.
- each input values are between -1 and 1

```
import neurolab as nl
net = nl.net.newp([[-1, 1], [-1, 1]], 10)
```



Functions

In order to create a network that can learn from the data and can achieve classification, we use some additional functions.

- **init():** initialization layers **e.g.** net.init()

- **reset():** clear of delay **e.g.** net.reset()

- **save(fname):** save network on file **e.g.** net.save('mynet')
  parameters: fname: file name

- **sim(input):** simulate a neural network **e.g.** net.sim(input)
  parameters: input: array like
  Returns: outputs: array like

- **train(input, target, epochs, show, lr):** train network **e.g.** net.train(input, target, epochs=1000, show=1, lr=0.1)

  input: the input data
  target: the target data (et. the categories)
  epochs: the number of presentations of the training set to network
  show: the number of epochs show
  lr: the learning rate

  Returns:
  errors: the errors from the training

Train

- The term network training, refers to the process of redefining weights and biases in order the network to be able to recognize input vectors in relation to the category in which they belong (output).

Learning rate

0,05 is a typical value. We can choose any other value we want.

Epochs

Batch learning algorithms take batches of training data to train a model. The batch algorithm keeps the system weights constant while computing the error associated with each sample in the input. The network error decreases if the vectors pass several times (epochs).

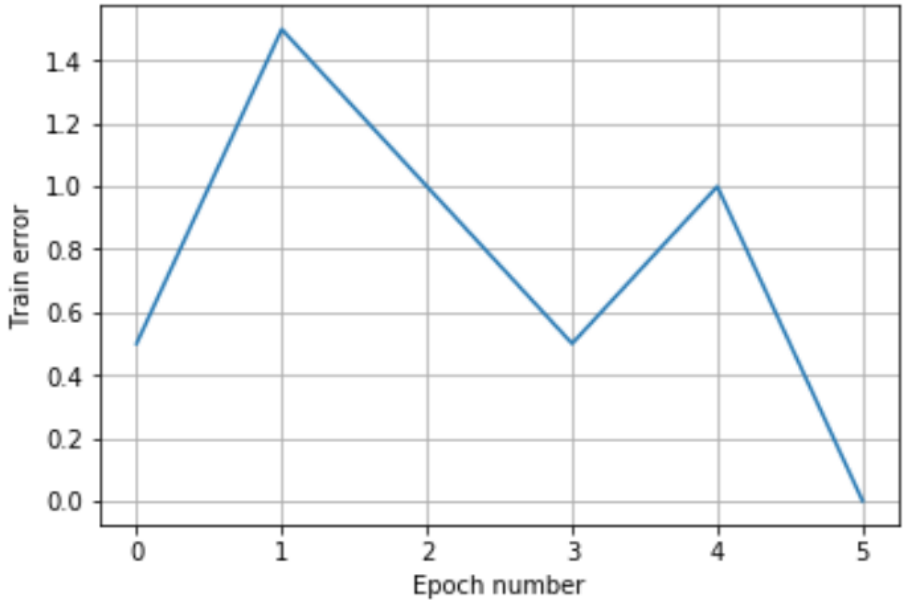
One epoch consists of one full training cycle on the training set. Once every sample in the dataset is presented to the network, we start again marking the beginning of the next epoch. The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset.
One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters.
An epoch is comprised of one or more batches.

Example 2

- We want to create a network with 4 inputs (patterns) from 2 categories.
- There is 1 neuron.

```
import neurolab as nl
import matplotlib.pyplot as plt
input = [[0, 0], [0, 1], [1, 0], [1, 1]]
target = [[0], [0], [0], [1]]

# Create net with 2 inputs and 1 neuron
net = nl.net.newp(nl.tool.minmax(input), 1)

# train with delta rule (default)
error = net.train(input, target,epochs=1000, show=2, lr=0.1)

# Plot results
plt.plot(error)
plt.xlabel('Epoch number')
plt.ylabel('Train error')
plt.grid()
plt.show();
```

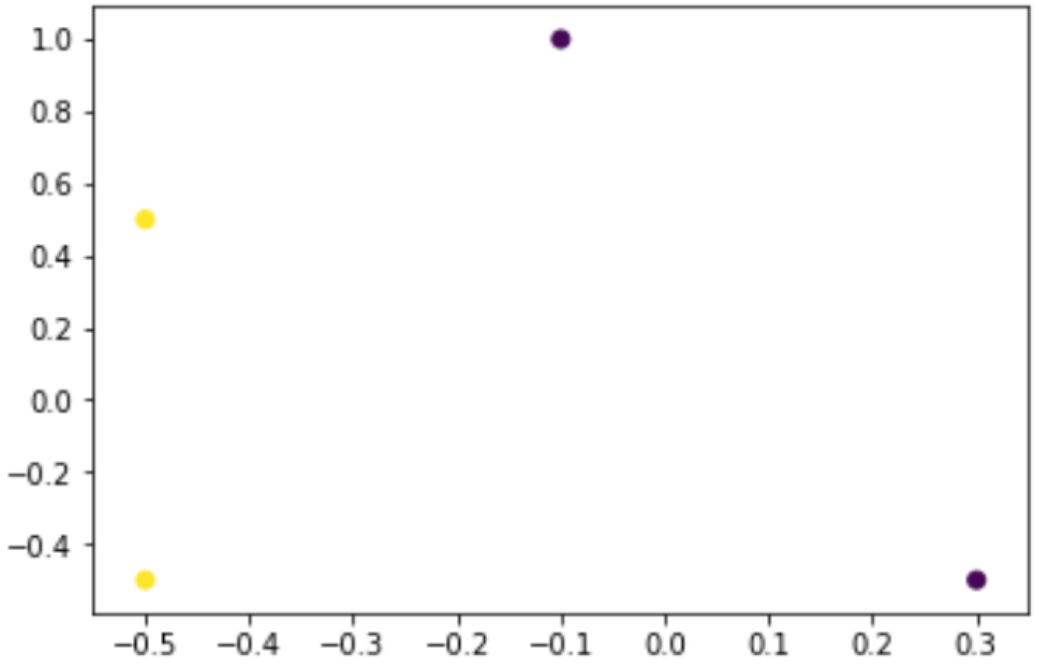
```
Epoch: 2; Error: 1.5;
Epoch: 4; Error: 0.5;
Epoch: 6; Error: 0.0;
The goal of learning is reached
```



Example 3

Suppose we want to build a perceptron network consisting of 2 inputs (features), and we want to be trained to classify 4 training patterns into two categories.

```
import neurolab as nl
import matplotlib.pyplot as plt
import numpy as np

X = np.array([[-0.5, -0.5, 0.3, -0.1], [-0.5, 0.5, -0.5, 1.0]]);
T = np.array([[1,1], [1,1], [0,0], [0,0]])

plt.scatter(X[0], X[1], c=T[:,0]);
```

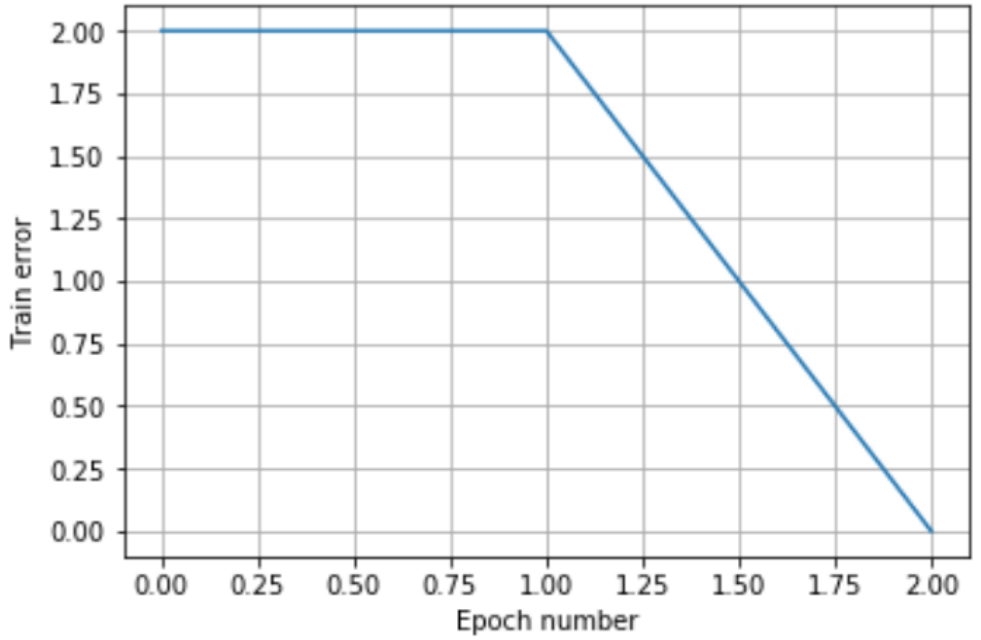


Example 3 cont

Suppose we want to build a perceptron network consisting of 2 inputs (features), and we want to train it, to classify 4 training patterns into two categories.

```
X = X.T #we traspose the input data

net   = nl.net.newp(nl.tool.minmax(X), 2)
error = net.train(X, T, epochs=100, show=1, lr=0.1)

# Plot results

plt.plot(error)
plt.xlabel('Epoch number')
plt.ylabel('Train error')
plt.grid()
plt.show();
net.save('my_net')
net.sim(X)
```

```
Epoch: 1; Error: 2.0;
Epoch: 2; Error: 2.0;
Epoch: 3; Error: 0.0;
The goal of learning is reached
```



We have used 2 neurons to represent each target

Multi Layer Perceptron

- The perceptron neural networks we have examined so far, have the disadvantage that they can not deal with problems requiring non-linear separation of the patterns we are giving as input.
- The backpropagation networks due to their many layers and especially the non-linear activation functions of their neurons come to solve this problem.

Backpropagation Networks

- The number of problems that these networks can face is wide.
- They are suitable for correlating the input vectors of a neural network with the output vectors. Therefore, we can use them:

  - To assign vectors in predefined classes
  - To forecast values

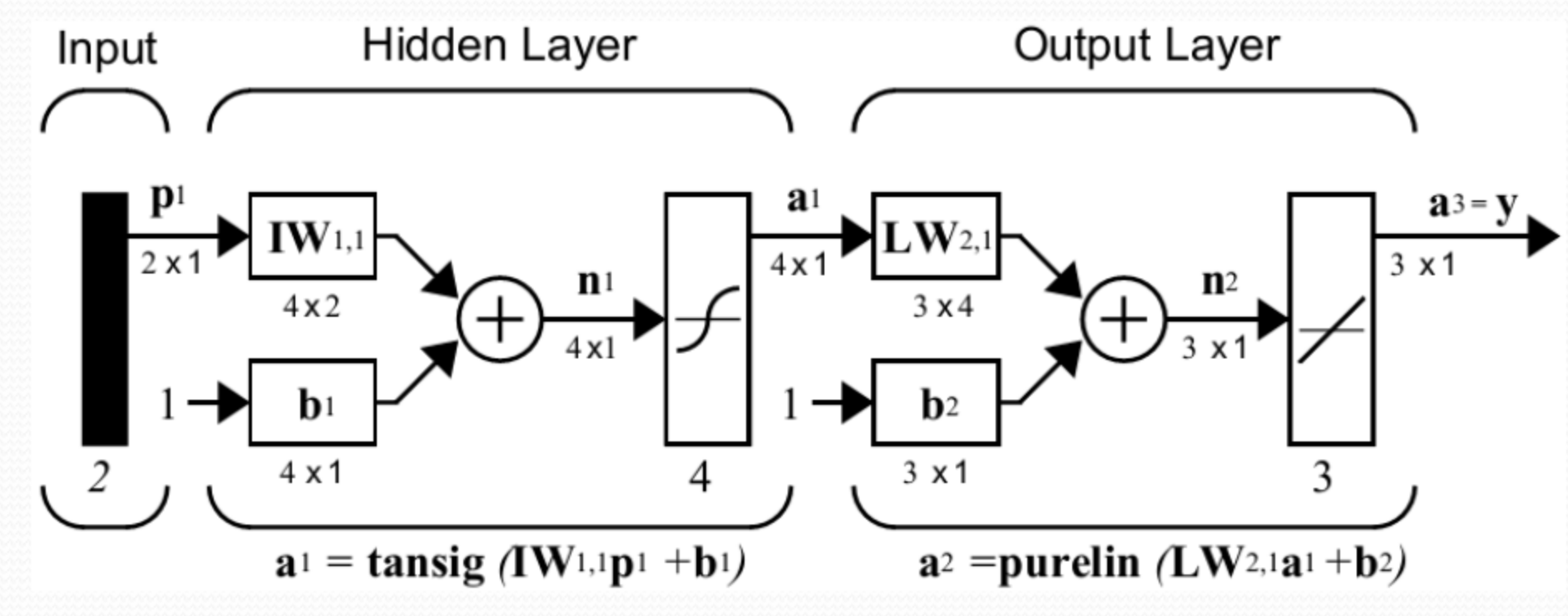
- We will build a multi-layer backpropagation neural network and then train it in order to use it in pattern recognition applications.

Procedure

- Choose dataset
- Preprocess the data
- Define the architecture of the neural network
- Train the neural network
- Test and optimize the neural network

Multilayer feedforward backpropagation networks consist of fully interconnected layers of neurons.

Network Architecture

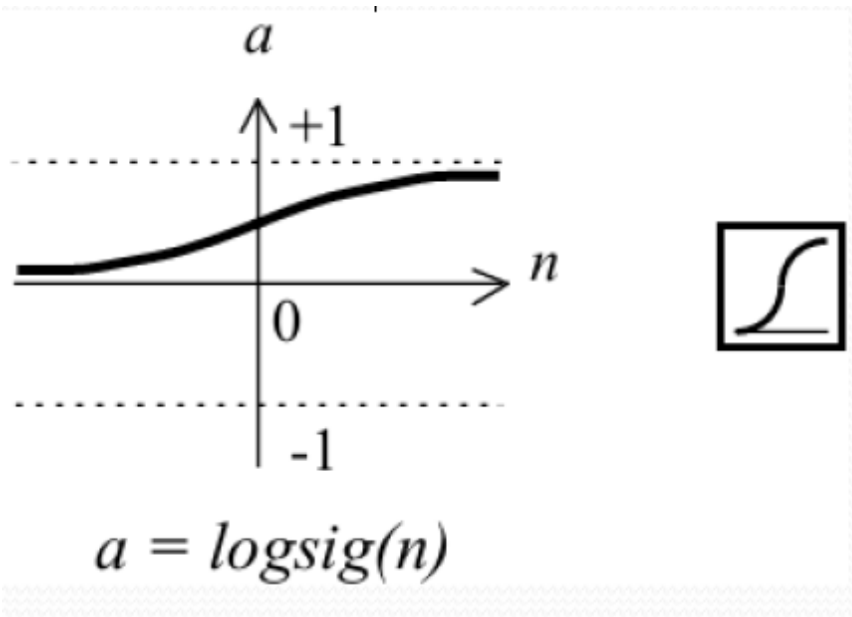


From the above example we conclude that:

- Two-dimensional weight tables are created between the layers.
- Each layer has a one-dimensional matrix with the biases of each neuron.

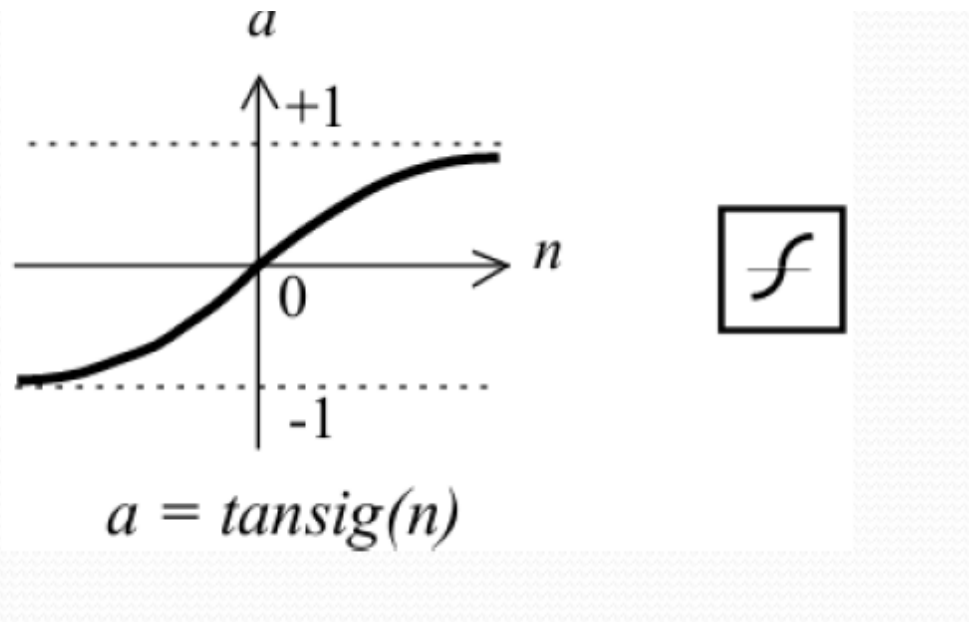
Activation Functions

- Due to the fact that we want the network to solve non-linear problems, the activation function to be implemented by each neuron should be non-linear.



Sigmoid activation function

Returns (output) any number between 0 and 1



Tan-sigmoid activation function

Returns (output) any number between -1 and 1

Activation Functions cont

- As we can see, the previous activation functions limit the range we have at the output of the network.
- If an application requires to have as output values that are not in the range of 0 to 1 or -1 to 1, then we can use at the last layer of the network the linear activation function which can provide us with any output value.

Function newff

To create a feed forward network we use the function **newff** from neurolab library.

Syntax:

```
neurolab.net.newff(minmax, size, transf=None)
```

Parameters:

- **minmax:** list of list, the outer list is the number of input neurons, inner lists must contain 2 elements: min and maxRange of each input value
- **size:** the length of list equal to the number of layers except input layer, the element of the list is the neuron number for corresponding layer contains the number of neurons for each layer
- **transf:** list (default TanSig) List of activation function for each layer

Returns:

- **net:** Net

Example 1

- We want to create a network with 2 inputs.
- Input range for each input is [-0.5, 0.5].
- 3 neurons for hidden layer, 1 neuron for output
- 2 layers including hidden layer and output layer

```
import neurolab as nl
net = nl.net.newff([[-0.5, 0.5], [-0.5, 0.5]], [3, 1])

len(net.layers) #the  number of layers
2
```

Train Functions

The train functions that apply Backpropagation technique:

Function	Algorithm
train_gd	Gradient descent
train_gdm	Gradient descent with momentum
train_gda	Gradient descent with adaptive learning rate
train_gdx	Gradient descent with momentum and adaptive lr
train_rprop	Resilient Backpropagation
train_bfgs	Broyden Fletcer Goldfarb Shasannp (BFGS) method using scipy.optimize.fmin_bfgs
train_cg	Newton-CG method Using scipy.optimize.fmin_ncg

Network Training

- We will deal with supervised learning.
- The function we use for network training is train () and its syntax is:

e.g. net.trainf = nl.train.train_gd(net,X, T, epochs=a, show=b, lr=c, goal=d)

where,

net: the initialized network
X: the input vector
T: the target vector
a: the max number of epochs we want to train the network
b: the number of epochs' shows
c: the desired learning rate
d: the minimum number of error we want to achieve

Backpropagation Learning

If, for example, we have a neural network where the last layer has one neuron and the actual output value is α while the desired value is t, then the resulting output error is equal to:
$e = t-\alpha$

Due to the many input values we use to train the network, many error values result, which ultimately define a function with discrete values.

Backpropagation starts from the end of the network and adjusts weights and biases in order to reduce the error.

Backpropagation Learning

That is, it adapts the weights and biases in the direction of the negative derivative of the error function.

The general formula for adjusting is:

$x_{k+1} = x_k - lr * gk$

where,
x could be any value of weight or bias
x_k is its initial value
x_{k+1} is the value after adjusting the training

The value of **gk** refers to the corresponding error derivative while the **lr** (learning rate) determines the magnitude of gk's influence on the readjustment.
A large value of lr (learning rate) leads to a large readjustment in weight or bias values. It is a parameter that is determined depending on the needs of the training. 0,05 is a typical value. We can choose any other value we want.

Setting up the parameters for the training cont

- Desired Learning Error

10^{-5} is a typical value we can choose.

- This parameter specifies the limit in which if the error occurs we consider the network trained and stop the training process.
- The best case is, of course, to set a zero error as a target. The problem is that this value is not being managed by any neural network for the overwhelming majority of the problems we have to deal with.

Training Performance

To calculate the network's performance, we must find the percentage of correct classifications in unknown patterns.

That is, we test the ability of the network to categorize vectors that have not been used to train it.

This ability of a neural network is called ***Generalization***.

We usually use **70-80%** for training and the remaining **30-20%** to determine network's performance.

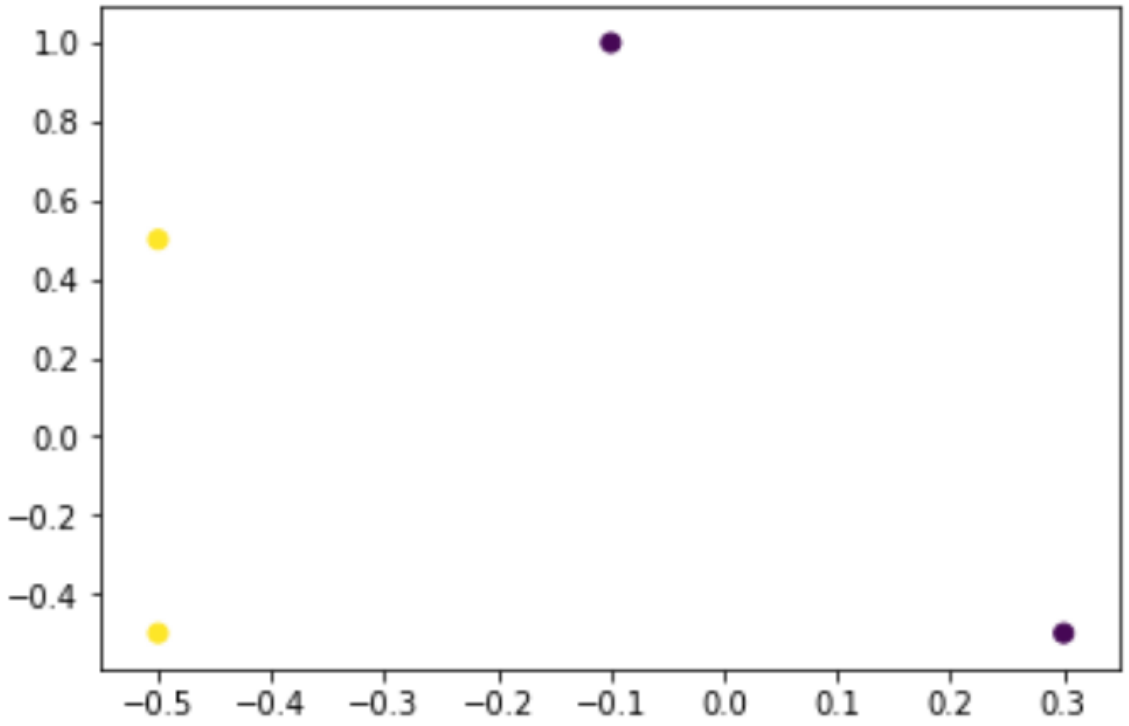
Modify network's properties

```
>>> # Create network
>>> net = nl.net.newff([[-1, 1]], [5, 1])
>>> # Default train function (train_gdx)
>>> print net.trainf
Trainer(TrainGDX)
>>> # Change train function
>>> net.trainf = nl.train.train_bfgs
>>> # Change init function
>>> for l in net.layers:
...     l.initf = nl.init.InitRand([-2., 2.], 'wb')
>>> # new initialized
>>> net.init()
>>> # Change error function
>>> net.errorf = nl.error.MSE()
>>> # Change weight of input layer
>>> net.layers[0].np['w'][:] = 0.0
>>> net.layers[0].np['w']
>>> array([[ 0.],
          [ 0.],
          [ 0.],
          [ 0.],
          [ 0.]])
>>> # Change bias of input layer
>>> net.layers[0].np['b'][:] = 1.0
>>> net.layers[0].np['b']
array([ 1.,  1.,  1.,  1.,  1.])
>>> # Save network in file
>>> net.save('test.net')
>>> # Load network
>>> net = nl.load('test.net')
```

Example 2

- We want to create a network with **4** input patterns each one having 2 features and belonging in one of **2** categories.
- **5** neurons for hidden layer, **2** neuron for output
- **2** layers including hidden layer and output layer
- **LogSig** activation function for all layers
- **rprop** train function

```
import neurolab as nl
import matplotlib.pyplot as plt
import numpy as np

X = np.array([[-0.5, -0.5, 0.3, -0.1], [-0.5, 0.5, -0.5, 1.0]]);
T = np.array([[1,1], [1,1], [0,0], [0,0]])
plt.scatter(X[0], X[1], c=T[:,0]);
```

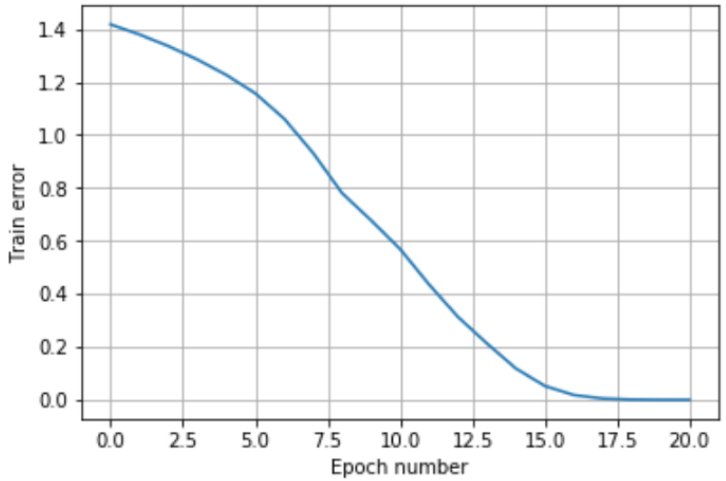


Example 2 cont

```
X = X.T #we transpose the input data

net   = nl.net.newff(nl.tool.minmax(X), [5,5,2])

#change the activation function for the output layer
net.layers[-1].transf = nl.trans.LogSig()  #output layer
net.layers[0].transf  = nl.trans.LogSig()  #hidden layer
net.layers[1].transf  = nl.trans.LogSig()  #hidden layer

# train the network
error = net.trainf = nl.train.train_rprop(net,X, T, epochs=100, show=1, lr =0.05 ,goal=0.0001)

# Plot results
plt.plot(error)
plt.xlabel('Epoch number')
plt.ylabel('Train error')
plt.grid()
plt.show();
```

Example 2 cont

```
Epoch: 1; Error: 1.4170430074593903;
Epoch: 2; Error: 1.3787900986143176;
Epoch: 3; Error: 1.3349225098315336;
Epoch: 4; Error: 1.2843814342258322;
Epoch: 5; Error: 1.2260592907160348;
Epoch: 6; Error: 1.1568276817488972;
Epoch: 7; Error: 1.0607697631098567;
Epoch: 8; Error: 0.9311370742332911;
Epoch: 9; Error: 0.779783718196264;
Epoch: 10; Error: 0.6775816803296378;
Epoch: 11; Error: 0.5694006896224931;
Epoch: 12; Error: 0.4357060877558401;
Epoch: 13; Error: 0.3130939335645598;
Epoch: 14; Error: 0.2130000821752202;
Epoch: 15; Error: 0.11855150561351777;
Epoch: 16; Error: 0.052558510922637296;
Epoch: 17; Error: 0.018069460857830132;
Epoch: 18; Error: 0.0048540426372644906;
Epoch: 19; Error: 0.0009734209410503212;
Epoch: 20; Error: 0.0001544657102132055;
Epoch: 21; Error: 1.7748731155529953e-05;
The goal of learning is reached
```




```
error[-1] #the error from the last epoch
```

```
1.7748731155529953e-05
```

Example 3

We will use the wine dataset. There are 2 tables:

- The wine_data matrix, in which there are:
  - **178** wines – training patterns (rows) , each having **13** features (columns)

- The wine_target matrix, in which there are:
  - **3** winemakers (columns) for each training pattern. There are 59,71 and 48 wines for the first, the second and the third winemaker respectively

Dataset's Features

- 1. Alcohol
- 2. Malic acid
- 3. Ash
- 4. Alcalinity of ash
- 5. Magnesium
- 6. Total phenols
- 7. Flavanoids
- 8. Nonflavanoid phenols
- 9. Proanthocyanins
- 10. Color intensity
- 11. Hue
- 12. OD280/OD315 of diluted wines
- 13. Proline

Data Input

Load the data:

```
import numpy as np
from pandas import *
import matplotlib.pyplot as plt
import neurolab as nl
from sklearn.datasets import load_wine

wine = load_wine()
X = wine.data
T = wine.target
```

Data Input

To see the first 5 patterns along with the header of the dataset (features names) issue:

```
df = DataFrame(wine.data, columns=wine.feature_names)
df.head()
```

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280/od315
0	14.23	1.71	2.43	15.6	127.0	2.80	3.06	0.28	2.29	5.64	1.04	
1	13.20	1.78	2.14	11.2	100.0	2.65	2.76	0.26	1.28	4.38	1.05	
2	13.16	2.36	2.67	18.6	101.0	2.80	3.24	0.30	2.81	5.68	1.03	
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	0.24	2.18	7.80	0.86	
4	13.24	2.59	2.87	21.0	118.0	2.80	2.69	0.39	1.82	4.32	1.04	

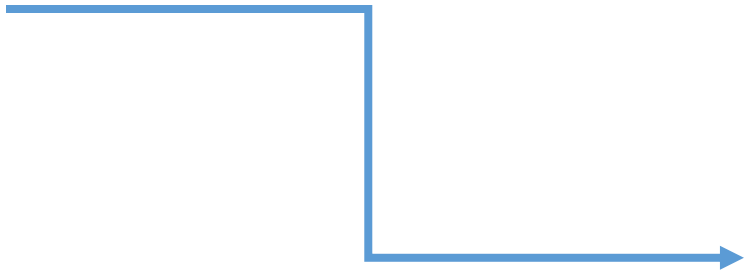
Targets Transform

The targets have the values: [0,1,2]
0 for the first winemaker
1 for the second winemaker
2 for the third winemaker

We need to transform the targets.

That means:

0 becomes **000**
1 becomes **001**
2 becomes **010**



```
T = np.empty([178, 3])
T[:,0]          = 0
T[0:131,1]      = 0
T[130:,1]       = 1
T[0:59,2]       = 0
T[59:130,2]     = 1
T[130:,2]       = 0
```

Data Split

We will divide the data into 2 sets. The first set will be used in network training (70%) and the second set will be used in network performance testing.

```
from sklearn.model_selection import train_test_split

 X_train, X_test, y_train, y_test = train_test_split(X, T,
test_size=0.30,random_state=42)
```

Preprocessing the data

Training a neural network can be more effective if we perform some preprocessing steps on the inputs and targets of network training data.

When your data is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes to all have the same scale. Therefore, it is often useful before training a neural network to change the scale to inputs and targets so that they always are within a defined value range.

We will use the MinMaxScaler to transform our data to a range [0, 1]

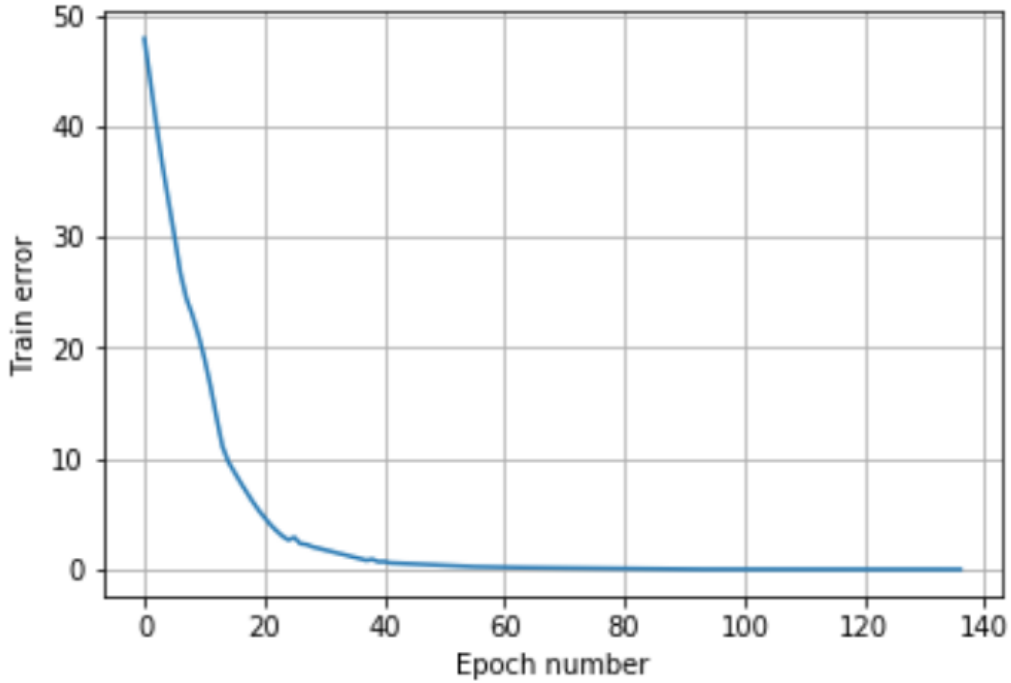
```
from sklearn import preprocessing
min_max_scaler = preprocessing.MinMaxScaler()
X_train = min_max_scaler.fit_transform(X_train)
X_test  = min_max_scaler.fit_transform(X_test)
```

Network Training

```
np.random.seed(42)
net   = nl.net.newff(nl.tool.minmax(X), [5,5,3])

#change the activation function for the output layer
net.layers[-1].transf = nl.trans.LogSig()  #output layer
net.layers[0].transf  = nl.trans.LogSig()  #hidden layer
net.layers[1].transf  = nl.trans.LogSig()  #hidden layer

# train the network
error = net.trainf = nl.train.train_rprop(net,X_train, y_train, epochs=10000, show=10, lr =0.05 ,goal=0.00005)

# Plot results
plt.plot(error)
plt.xlabel('Epoch number')
plt.ylabel('Train error')
plt.grid()
plt.show();
```

Network Training cont

```
Epoch: 10; Error: 21.173599284892553;
Epoch: 20; Error: 5.374859386899595;
Epoch: 30; Error: 1.9036313526349216;
Epoch: 40; Error: 0.6924395661549272;
Epoch: 50; Error: 0.38393916534348005;
Epoch: 60; Error: 0.19435566958305817;
Epoch: 70; Error: 0.1192397563126912;
Epoch: 80; Error: 0.0555628091359866;
Epoch: 90; Error: 0.011641779498293196;
Epoch: 100; Error: 0.0084111212329365628;
Epoch: 110; Error: 0.006007199354756838;
Epoch: 120; Error: 0.0039326165813679;
Epoch: 130; Error: 0.0014905892198120 98;
The goal of learning is reached
```



Error from the last epoch

```
error[-1]
```

```
0.0004675424879430433
```

Simulate network

```
out = net.sim(X_test)
out = np.around(out)
```

Accuracy

```
correct=(out == y_test).all(axis=1)
acc=(np.sum(correct)/len(out))*100
```

```
# A different approach to calculate accuracy
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, out)
print ("Accuracy: %.2f%%" % (accuracy * 100.0))
```

Network Output

list: error

index	values
0	70.1287161073
1	66.0390230354
2	61.057940602
3	55.0698772598
4	47.9590902432
5	39.9680090849
6	32.2745910208
7	27.7424876094
8	26.0508061625
9	24.9532788442
10	23.4556775001
11	21.7016219335

ndarray: out

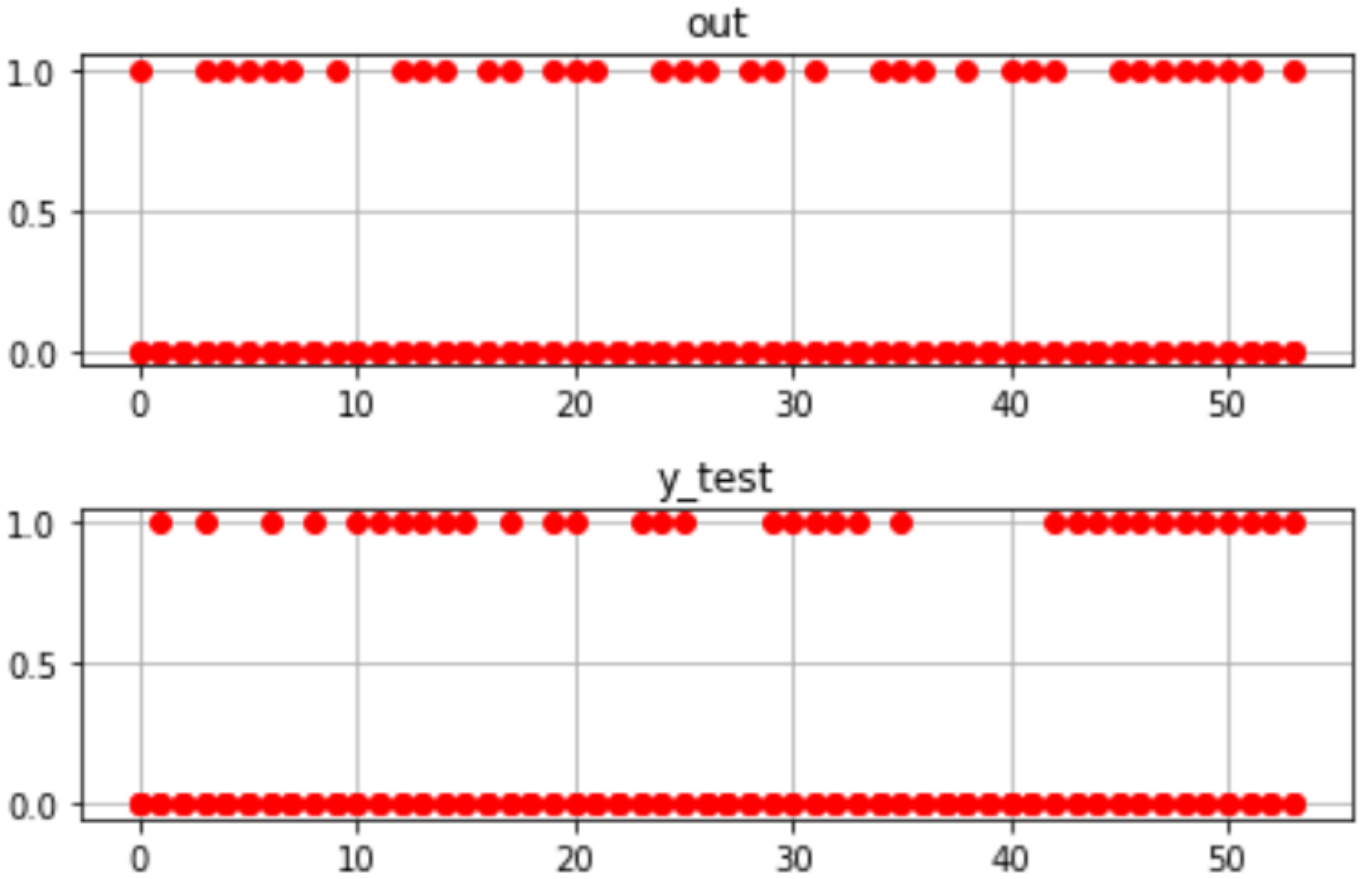
index	0	1	2
0	0	0	0
1	0	1	0
2	0	0	1
3	0	0	0
4	0	1	0
5	0	0	0
6	0	1	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	1	0
11	0	1	0

Compare the results

```
np.array_equal(out, y_test)
```

False

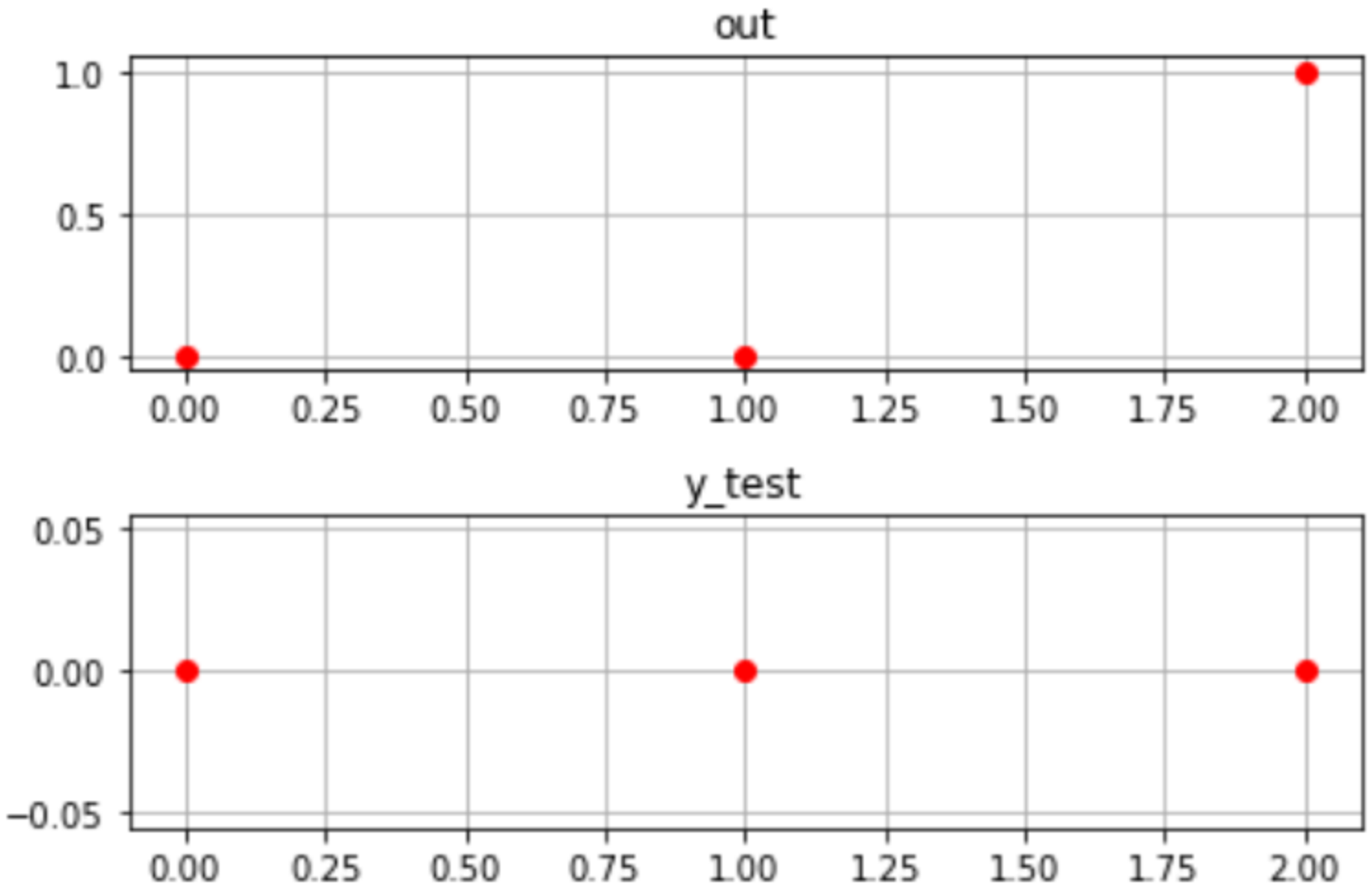
```
plt.figure(1)
plt.subplot(2,1,1)
plt.plot(out,'ro')
plt.title('out')
plt.grid(True)

plt.subplot(2,1,2)
plt.plot(y_test,'ro')
plt.title('y_test')
plt.grid(True)

plt.tight_layout()
plt.show()
```



Compare the results cont

We can also compare specific patterns

```
plt.figure(1)
plt.subplot(2,1,1)
plt.plot(out[34],'ro')
plt.title('out')
plt.grid(True)

plt.subplot(2,1,2)
plt.plot(y_test[34],'ro')
plt.title('y_test')
plt.grid(True)

plt.tight_layout()
plt.show()
```



Thank you !!!