Artificial Neural Network: Introduction to ANN

AAA-Python Edition

Plan

- **1-** Introduction to ANN
- **2-** Perceptron
- **3-** Single Layer Perceptron
- **4-** ANN topologies
- **5-** Multi Layer Perceptron
- **6-** ANN topologies

1- Introduction to ANN

Concept

- The idea of an **Artificial Neural Network** (ANN) is to build a model based on the way the human brain **learns new** things**.**

- It can be used in any type of **machine learning.** It learns by extracting the different underlying patterns in a given data.

- This extraction is performed by stages, called **layers.** Each layer, composed by a set of **neurons,** will identify a certain pattern. The following layer, will identify another **more complex** pattern, from its previous layer.

- The most common architecture is as follow: a first layer, has the training data as input. It is called the **input layer.** In the last one, the output of the neurons are the final output. It is called the **output** layer. The layers in between, are called **hidden layers.**
- From now, the term **neural network** will mean **artificial neural network.**

[By Amina Delali]

1- Introduction to ANN

Neurolab

- **Neurolab** is Neural Network library for Python. It supports several types of neural networks.

- For installation:

```
# installation of neurolab
!pip install neurolab
```

- Like other machine learning techniques, a neural network need to be **trained**. Can be **tested**. And will be used to **predict** results.

- Here is an example of how to use neurolab to create a neural network, and how to perform the fore-mentioned tasks:

```
import numpy as np
import neurolab as nl
# Create data
myInput = np.random.uniform(-0.5, 0.5, (10, 2))
# the labels correspond to the sum of two values
myLabels = (myInput[:, 0] + myInput[:, 1]).reshape(10, 1)
# Create network with 2 inputs, 5 neurons in hidden layer and 1 in output layer
myNN = nl.net.newff([[-0.5, 0.5], [-0.5, 0.5]], [5, 1])
# Train process
myErr = myNN.train(myInput,myLabels, show=15)
```

The samples are uniformally distributed from -0.5 (included) to 0.5 (excluded)

Length of the outer list Equals to the number of Neurons of the input layer

Input layer with **2** neurons, hidden layer with **5** neurons, and output layer with **1** neuron

[By Amina Delali]

1- Introduction to ANN

Neurolab example details

- Data: **10** samples described by **2** features. The labels are the **sum** of the **2** features. In fact, the NN tries to model the **sum** function for values ranging from **-0.5** to **0.5**

- After creating the data, the steps were:
  - Create an instance of a neural network with specified number of layers and neurons (**nl.net.newff)**
  - Train the neural network (**myNN.train**)
  - Predict the output for the value [0.2,0.1] (**myNN.sim**)
  - Compute the test error (the true label is known: 0.2+0.1)

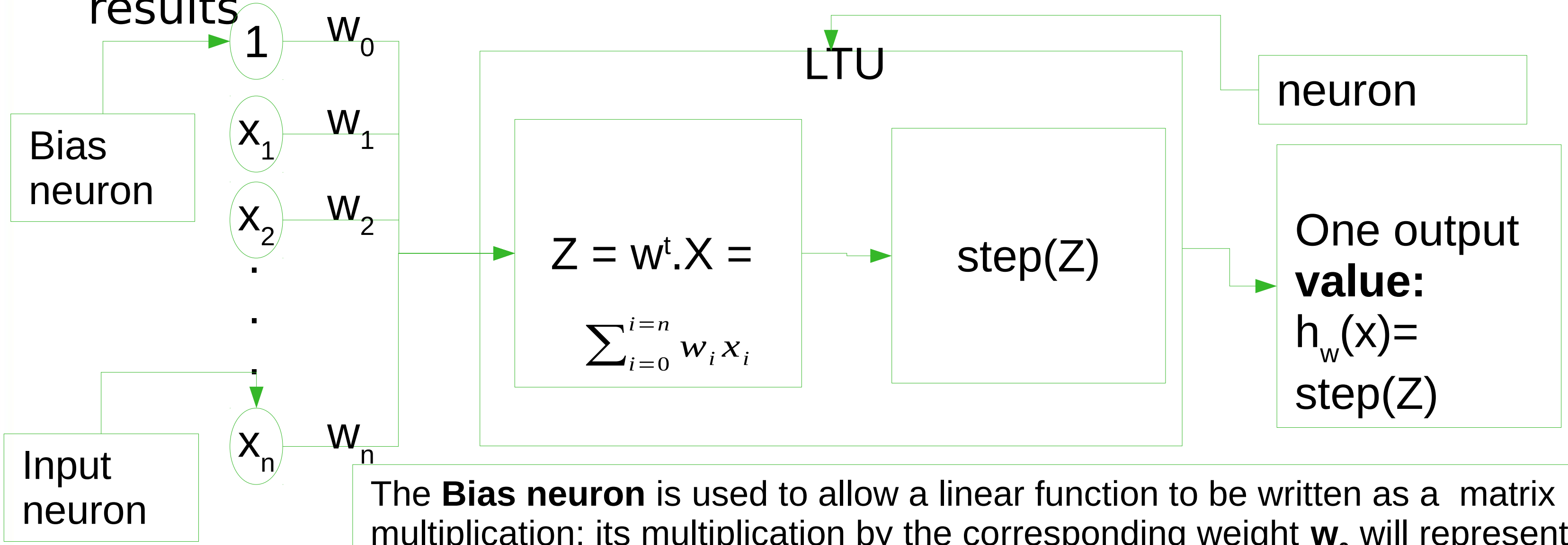
```
# Test and prediction process
pred= myNN.sim([[0.2, 0.1]])
# the result should be 0.3
testErr= np.abs(0.3-pred)
print ("Prediction=",pred)
print ("Test error = ",testErr)
```

```
The goal of learning is reached
Prediction= [[0.29308242]]
Test error =  [[0.00691758]]
```

2- Perceptron

Definition

- The term **Perceptron** refers to an input layer of data features values, with forward weighted connections to an output **layer** of **one single neuron** , or of **multiple neurons.**
- One of the simplest form of a neuron is an **LTU.**
- **LTU,** for **L**inear **T**hreshold **U**nit, it is a component (neuron) that:
  - Computes a **weighted sum** of its inputs: a linear function
  - Applies a **step** function to the resulting sum, and **outputs** the results



The **Bias neuron** is used to allow a linear function to be written as a matrix multiplication: its multiplication by the corresponding weight $\mathbf{w_0}$ will represent the **intercept** value **b**: ($\mathbf{a_1x_1 + a_2x_2 + \dots + a_nx_n + b}$).

[By Amina Delali]

2- Perceptron

The functions

- The weighted sum function, is also called the **Propagation** function.
- The step function, can be :
  - A **non-linear** function, in this case, it will be called the **threshold activation function (**this is the case in an **LTU**)**.** For example:
    - Heaviside step function:

      $$\text{heaviside}(z)=\begin{cases} 0 & if \quad z<0 \\ 1 & if \quad z\geq 0 \end{cases}$$

    - Sign function:

      $$\text{sgn}(z)=\begin{cases} -1 & if \quad z<0 \\ 0 & if \quad z=0 \\ 1 & if \quad z>0 \end{cases}$$

  - A **linear** function, simply called **activation function.** For example:
    - The **identity function**: which means that the value computed by the **propagation** function, **is** the **output** value of the neuron.
  - A **semi-linear** function, that is **monotonous** and **differentiable.** Also called **activation function.**

[By Amina Delali]

Single Layer Perceptron

- A **S**ingle **L**ayer **P**erceptron **(SLP),** is simply a Perceptron, with only one layer (**without** counting the **input layer**) .

- So, it is composed of an input layer and an output layer. The later one can have one ore more outputs. So, it can be used for binary and for multi-output classification

- Considering ANN in general, a **Perceptron** is considered as a **feedforward** neural network. We are going to talk about it in the next section.

- An **SLP** an apply **2** different kind for **rules** to learn: **the perceptron** rule, or the **delta rule.** Each of the rules is associated with a certain type of activation function. To apply the delta rule, we need the activation function to be differentiable.

SLP Learning

SLP learning

- With Perceptron rule (variant of **Hebb’s rule**)
  - Uses a threshold activation function
    - Learning algorithm has the same behavior of a **stochastic gradient descent** algorithm
- With the delta rule (**Widrow Hoff** rule)
  - Uses a **linear** or **semi-linear** activation function
    - **Offline** training: **batch gradient descent.**
    - **Online** (or **sequential**) training: **Stochastic Gradient descent algorithm**

[By Amina Delali]

SLP Learning: Perceptron Rule

- To update the weights, the following formula is used:

$$w_{i,j}^{(next\ \ step)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

The learning rate

The **true** value at the node **j,** for the current training sample.

True value the node **j** for the current training sample.

i^th input value (**i^th feature**) for the current training sample.

The weight of the **connection** between the **input** node **i** and the **output** node **j,** for the **next training sample.**

The weight of the **connection** between the **input** node **i** and the **output** node **j**, for the **current training sample**

- The concept is that each **wrong** prediction **reinforces** the **weight** corresponding to the **feature** that would **contributed** to the **correct prediction.** The computation of the weights is repeated until the samples are classified correctly.

Gradient Descent

- The concept of the **gradient descent** is:
  - With **initial parameters** of a model, predict an **output value**
  - Compute the gradient of the "**error**" ("loss") function **(**function of the **parameters** of the learning model**)** at a certain point= the slope of the surface of that function at that point calculated by its **derivative** at that point.
  - Update the parameters in order to find the **local minima** by a step **proportional** to **the negative** of that **gradient**. (opposite direction ==> toward the local minima of the function). In the case of :
    - **A stochastic gradient descent**: with **one sample**, **predict** → **update** the parameters for the **next sample** → predict with the next sample with the new parameters
    - **A Batch gradient descent:** predict for all samples ==**1 epoch** → **update** the parameters → predict again with the new parameters
  - Repeat the process in order to **minimize** the error.

3- Single Layer Perceptron

[By Amina Delali]

SLP Learning: Delta Rule

- With the activation rule being **linear** or **semi-linear but differentiable,** the gradient descent is used to update the weights.
- The weights are updated as follow: $w_{i,j}^{next} = w_{i,j} + \Delta w_{i,j}$
  - In general: $\Delta w = \frac{-\eta \cdot \partial E}{\partial w}$
  - In a case of **a linear activation function, and a Sum-Squared** error function
    - **In Online training:** for a given sample: $\Delta w_{i,j} = \eta \cdot x_i \cdot (y_j - \hat{y}_j) = \eta \cdot \delta_j$

This is why it is called the **delta rule**

The difference here with the perceptron rule is the type of activation function used

- **In Offline training:**

$$\Delta w_{i,j} = \eta \cdot \sum_{s \in X} x_i^s \cdot (y_j^s - \hat{y}_j^s) = \eta \cdot \sum_{s \in X} x_i^s \cdot \delta_j^s$$

Predicted label for output neuron **j,** for the sample **s**

The **input feature i** value of the sample **s**

The **true** label that should have the output neuron **j**, for the sample **s**

Learning rate

[By Amina Delali]

3- Single Layer Perceptron

With sklearn: perceptron rule

```
#SLP using Perceptron class

from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
#load iris datasets
X, y = load_iris(return_X_y=True)
# instance of a perceptron: SLP with perceptron rule
# learning rate =0.1, maximum of epoch = 5000, without shuffle after
# each iteration
mySLP = Perceptron(eta0=0.1, max_iter=5000,tol=1e-3,shuffle=False)
mySLP.fit(X, y)
print ("The score of the classification (without shuffle) = ", mySLP.score(X, y) )

# instance of an other perceptron: SLP with perceptron rule
# learning rate =0.1, maximum of epoch = 20, with shuffle after
# each iteration
mySLPShuf = Perceptron(eta0=0.1, max_iter=20,tol=1e-3,shuffle=True)
mySLPShuf.fit(X, y)
print ("The score of the classification (with shuffle) = ", mySLPShuf.score(X, y) )
```

The fact that after each weight update the samples were **shuffled**, only **20** iterations were sufficient to reach convergence.

```
The score of the classification (without shuffle) =  0.6666666666666666
The score of the classification (with shuffle) =  0.82
```

Sklearn uses **Hinge loss function** with threshol = **0** to compute the error of the prediction

- Eta0==**0.1** → Learning rate = **0.1**
- max_iter== **5000** → number of iterations if there is no convergence
- Shuffle == **False** → there is no shuffle after a weight is updated for the next sample

[By Amina Delali]

With sklearn: perceptron rule

```
from sklearn.linear_model import SGDClassifier
#SLP using SGDClassifier class
mySGDCl = SGDClassifier(loss="perceptron", learning_rate="constant", eta0=0.1,penalty = "none",
                        max_iter=20, shuffle= True,tol=1e-3, verbose=2,random_state=0
                       )
mySGDCl.fit(X, y)

print ("The score of the classification (with shuffle) = ", mySGDCl.score(X, y) )

print("The number of classes = ", len(mySGDCl.classes_))
```

Use of stochastic gradient descent classifier class

Since there is **3** classes, and since the classification strategy used is **one vs all**, the classifier will run on **3 times.** Each time with **max epochs** == **20**

```
-- Epoch 1
Norm: 0.83, NNZs: 4, Bias: 0.100000, T: 150, Avg. loss: 0.065987
Total training time: 0.00 seconds.
-- Epoch 2
Norm: 0.83, NNZs: 4, Bias: 0.100000, T: 300, Avg. loss: 0.000000
```

```
-- Epoch 1
Norm: 1.27, NNZs: 4, Bias: 0.000000, T: 150, Avg. loss: 1.399133
Total training time: 0.00 seconds.
-- Epoch 2
```

```
-- Epoch 1
Norm: 2.02, NNZs: 4, Bias: -0.400000, T: 150, Avg. loss: 1.023480
Total training time: 0.00 seconds.
```

```
The score of the classification (with shuffle) =  0.82
The number of classes =  3
```

[By Amina Delali]

Example with neurolab: delta rule

```
The number of samples=  150
The number of Features =  4
```

```python
f1_mi= X[:,0].min()
f1_ma= X[:,0].max()
f2_mi= X[:,1].min()
f2_ma= X[:,1].max()
f3_mi=X[:,2].min()
f3_ma= X[:,2].max()
f4_mi=X[:,3].min()
f4_ma= X[:,3].max()
# instantiate an SLP with: 4 input nuerons for th 4 features
# with 3 output neurons : to represent the 3 classes
#  The learning rule is the Delta rule, the activation function is the
# SoftMax function
mySLPDelta = nl.net.newp([[f1_mi,f1_ma],[f2_mi,f2_ma],[f3_mi,f3_ma],[f4_mi,f4_ma]],3,
                         transf=nl.trans.SoftMax())
```

Since we have **4** features → we need an **input layer** with **4 neurons.** And since we have **3 classes,** and since we will use **SoftMax** activation function (ideal for multi-class classification), we will need **3** output neuron:

Class 0 coded as 1 0 0

Class 1 coded as 0 1 0

Class 2 coded as 0 0 1

NB: the classes are **not coded** in a **binary code.**

```
The number of classes =  3
```

The corresponding formula of **SoftMax** function is :

$$f(x_i) = \frac{e_i^x}{\sum_{j=0}^{N} e_j^x}$$

N is the Number of samples

[By Amina Delali]

Example with neurolab: delta rule

4- Single Layer Perceptron examples

```
# separate the data into test and train samples
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X,y,test_size=0.2)
```

```
array([2, 2, 0])
```

```
array([[0, 0, 1],
       [0, 0, 1],
       [1, 0, 0]])
```

Third class → third column == 1

```
# we have to construct the target values (true labels)
# we will convert each of the labels vectors into a 3 columns
# vector
from sklearn.preprocessing import LabelBinarizer
from sklearn.utils import shuffle
myLB = LabelBinarizer()
y3_train= myLB.fit_transform(y_train)
y3_test= myLB.fit_transform(y_test)
```

```
# train the SLP
myErr2 = mySLPDelta.train(x_train,y3_train, epochs=1000, show=100, lr=0.01)
```

This implementa-tion in **neurolab** uses the **sum squared error** as cost function. And it **doesn’t use** the **derivative** of the activation function used

```
# define a prediction function
def Predict(x, Net):
  if np.ndim(x)==1:
    x=[x]
  res = Net.sim(x)
  return np.argmax(res,axis=1)
```

Select the index corresponding to the maximum **probability** as the **predicted class label**

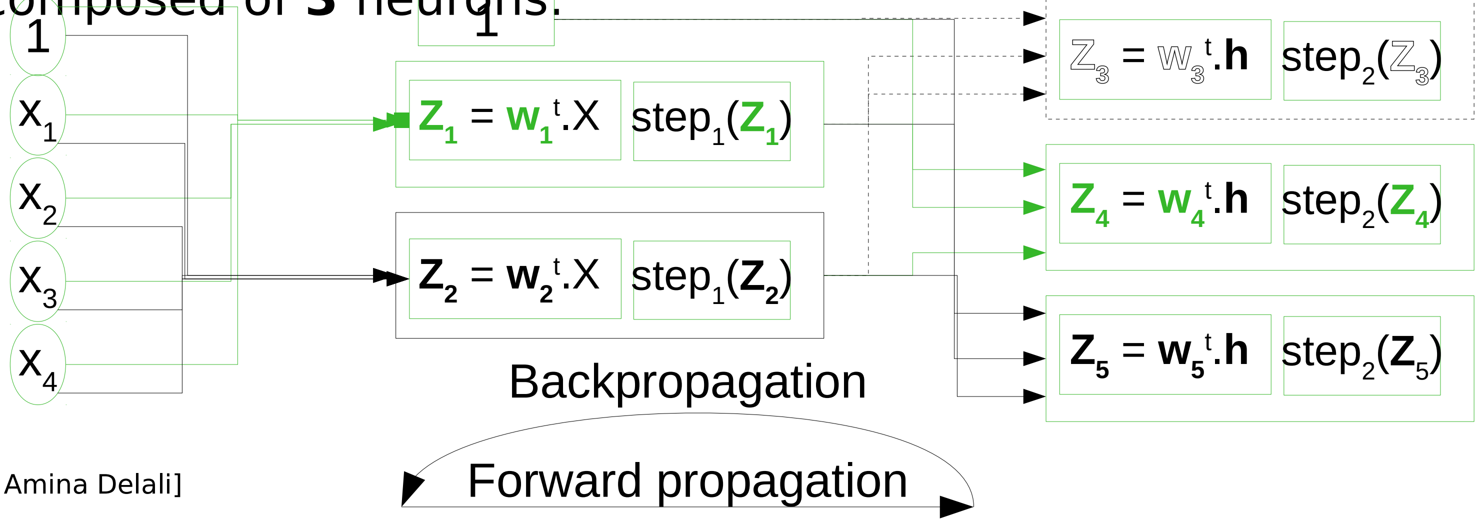
```
Accuracy of the training =  0.97
Accuracy of the testing =  0.97
```

```
# compute the accuracy of the SLP predictions
# for the training and testing data

yPred = Predict(x_train,mySLPDelta)
accuracy =np.count_nonzero(yPred== y_train) / y_train.shape[0]
print ("Accuracy of the training = ",np.round(accuracy,2))

yP_test = Predict(x_test,mySLPDelta)
accuracy2 =np.count_nonzero(yP_test== y_test) / y_test.shape[0]
print ("Accuracy of the testing = ",np.round(accuracy2,2))
```

[By Amina Delali]

Multi-Layer Perceptron

- An **MLP** (**M**ulti-**L**ayer **P**erceptron) is a **Percetron** with **one** or **more hidden layers.**
- It is another **Feed Forwad Artificial neural network.** Each of the layers (**except** the **output** layer) includes a **bias** neuron.
- An **ANN** with more than **one hidden** layer is a **Deep Neural Network** (**DNN**).
- Lets consider this MLP: input layer with **4** neurons, second(**hidden**) layer with **2** neurons (for both first and second layer the bias neurons are not counted). The last layer is composed of **3** neurons:



[By Amina Delali]

5- Multi Layer Perceptron

Backpropagation

- It is a generalization of the **delta rule.** After a **forward pass,** a **backward** pass is applied to **update** the **weights** to back propagate the **errors,** using **gradient descent** procedure.
- This forward/backward passes are repeated until the error function is minimized
- The formula (of the generalized delta rule) is:

$$\Delta_{w_{k,h}} = \eta o_k \delta_h$$

The update value for the weights of the connection between the neuron k and neuron h

Output of the neuron k

$$\delta_h = \begin{cases} f'_{act}(net_h)\cdot(y_h - \hat{y}_h) & (h \quad is \quad an \quad output \quad neuron) \\ f'_{act}(net_h)\cdot\sum_{l\in L}\delta_{w_{h,l}} & (h \quad is \quad a \quad neuron \quad of \quad a \quad hidden \quad layer) \end{cases}$$

The derivative of the activation function of the neuron h

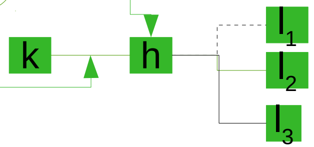
Inputs of neuron h: result of propagation function at h

True label

Predicted label

The connection with the weights to update

The connection with the weights to update



[By Amina Delali]

5- Multi Layer Perceptron

Example

```
# an MLP with an input layer with 4 neurons, a hidden layer with
# 2 neurons and an output layer with 3 neurons
myMLP = nl.net.newff([[f1_mi,f1_ma],[f2_mi,f2_ma],[f3_mi,f3_ma],[f4_mi,f4_ma]],[2, 3],
                          [nl.trans.TanSig(), nl.trans.SoftMax()])
```

Activation function for the second (first hidden) layer

Activation function for the third (output) layer

```
# train the MLP
myErr2 = myMLP.train(x_train,y3_train, epochs=10000, show=100)
```

```
The goal of learning is reached
```

TanSig formula:

$$y=\frac{2}{1+e^{-2n}}-1$$

```
# compute the accuracy of the MLP predictions
# for the training and testing data

yPred = Predict(x_train,myMLP)
accuracy =np.count_nonzero(yPred == y_train) / y_train.shape[0]
print ("Accuracy of the training = ",np.round(accuracy,2))

yP_test = Predict(x_test,myMLP)
accuracy2 =np.count_nonzero(yP_test== y_test) / y_test.shape[0]
print ("Accuracy of the testing = ",np.round(accuracy2,2))
```

```
Accuracy of the training =  1.0
Accuracy of the testing =  0.93
```

The accuracy of training is equal = 1.0 (100%) but with test data we got less good results as with the SLP ==> over-fitting

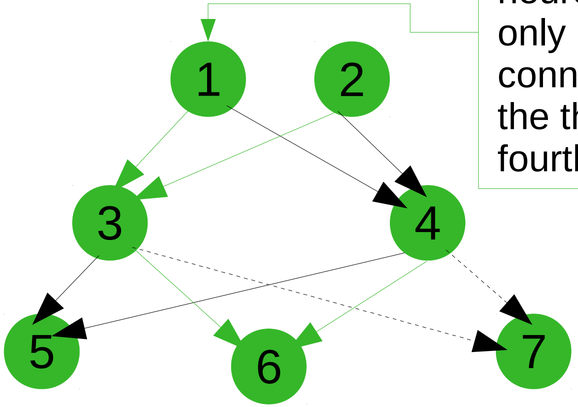
[By Amina Delali]

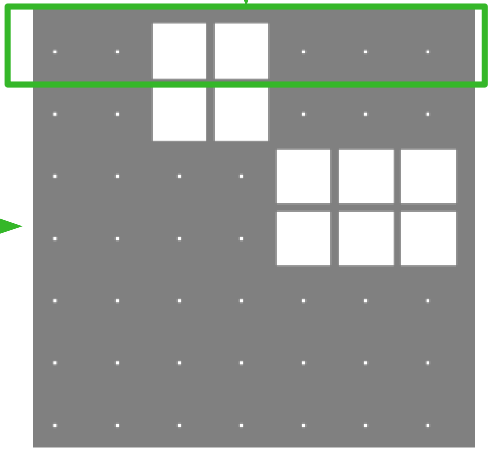
FeedForward Neural Networks

- We have already seen **feedforward neural networks** (SLP and MLP)
  - One input layer + n hidden layers + one output layer (n>=1)
  - Connection are only allowed to neurons of the following layers
  - They can have **shortcut connections:** the connection are not set to the following layers but to subsequent layers

Connections from the First neuron: it has only connections to the third and fourth neuron

The white squares will correspond to the existing connections





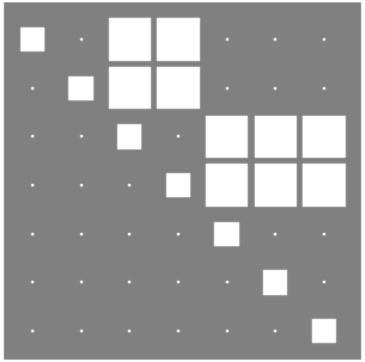
The corresponding Hinton Diagram

[By Amina Delali]

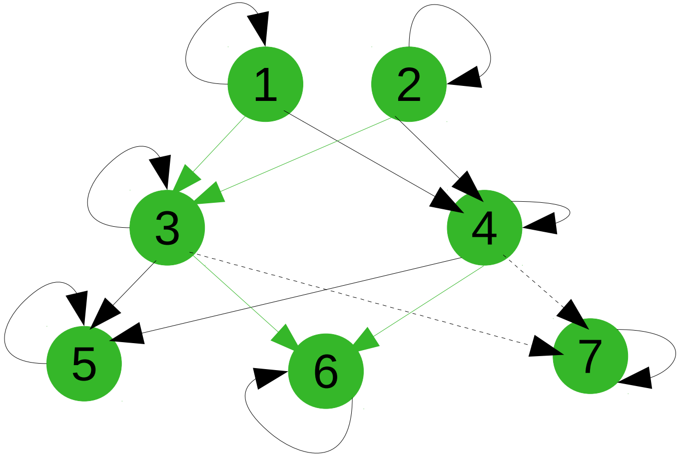
Recurrent Networks

Direct Recurrence

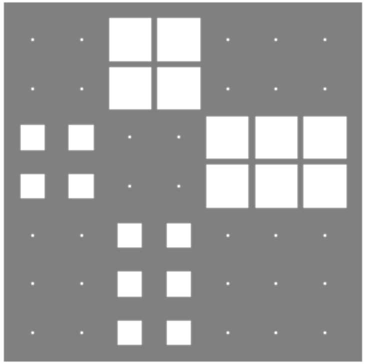
- Multiple layers with connections allowed to neurons of the following layers
- A neuron can also be connected to itself

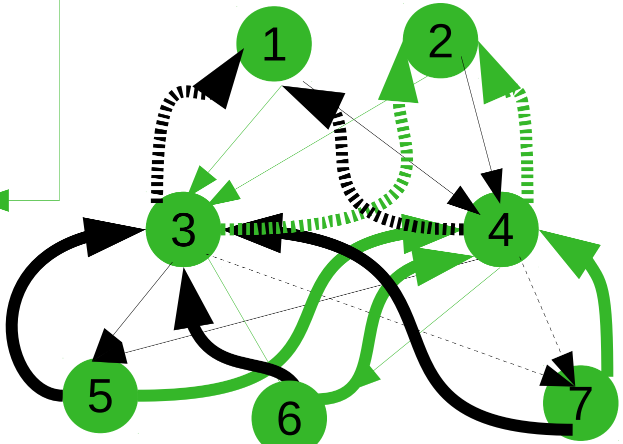


For visualization purposes the recurrent connections are represented by smaller squares



Indirect Recurrence





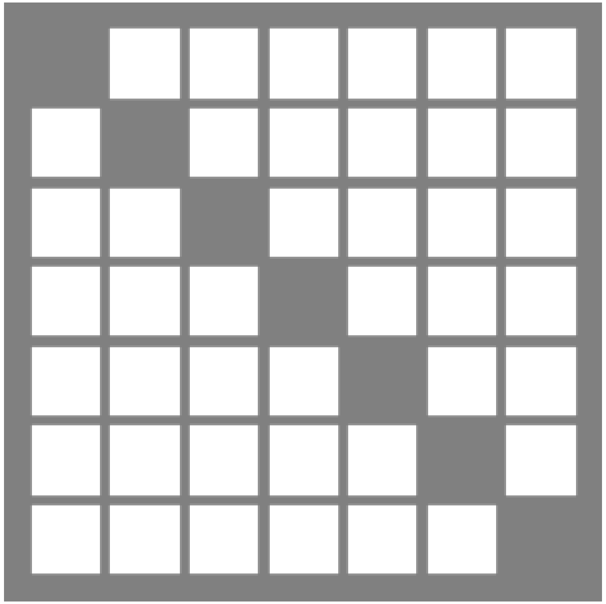
- Multiple layers with connections allowed to neurons of the following layers
- Connections are also allowed between neurons and preceding layer

[By Amina Delali]

6- ANN topologies

Fully connected Neural Network

- Multiple layers with connections allowed from any neuron to any other neuron
- Direct recurrence is not allowed
- Connections must be symmetric



```
myFC = 49 *[3]
myFC = np.array(myFC).reshape(7,7)
for i in range(7):
  myFC[i,i] = 0.01
#print(myFC)
plots.hinton(myFC,add_legend=False)
```

[By Amina Delali]

References

- Joshi Prateek. Artificial intelligence with Python. Packt Publishing, 2017.
- Jake VanderPlas. Python data science handbook: essential tools for working with data. O’Reilly Media, Inc, 2017.
- Neural Networks – II, Version 2 CSE IIT, Kharagpur, available at: https://nptel.ac.in/courses/106105078/pdf/Lesson%2038.pdf
- Isaac Changhau, Loss Functions in Neural Networks, 2017, available at: https://isaacchanghau.github.io/post/loss_functions/
- Sebastian Seung, The delta rule, MIT Department of Brain and Cognitive Sciences , Introduction to Neural Networks, 2005, https://ocw.mit.edu/courses/brain-and-cognitive-sciences/9-641j-introduction-to-neural-networks-spring-2005/lecture-notes/lec19_delta.pdf
- NeuPy, Neural Networks in Python,http://neupy.com/pages/home.html

Thank you!

FOR ALL YOUR TIME