

Neural Network Basics

TUYEN NGOC LE

Outline

1. Artificial Neural Networks and their relation to biology
2. The seminal Perceptron algorithm
3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently
4. How to train neural networks using the Keras library

Types of Neural Networks

There are many types of neural networks available or that might be in the development stage. They can be classified depending on their: Structure, Data flow, Neurons used and their density, Layers and their depth activation filters etc.

A. Perceptron

B. Feed Forward Neural Networks

C. Multilayer Perceptron

D. Convolutional Neural Network

E. Radial Basis Function Neural Networks

F. Recurrent Neural Networks

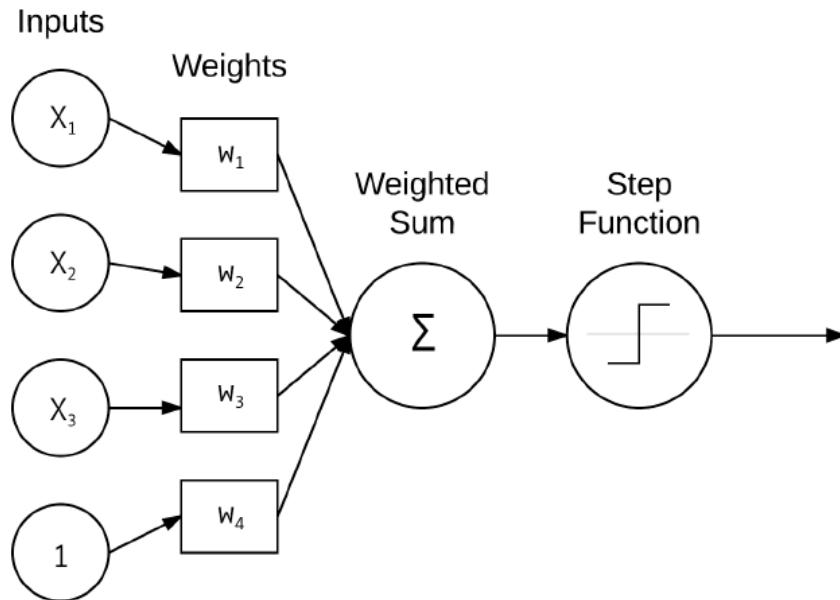
G. Sequence to sequence models

H. Modular Neural Network

Outline

- 1. Artificial Neural Networks and their relation to biology**
2. The seminal Perceptron algorithm
3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently
4. How to train neural networks using the Keras library

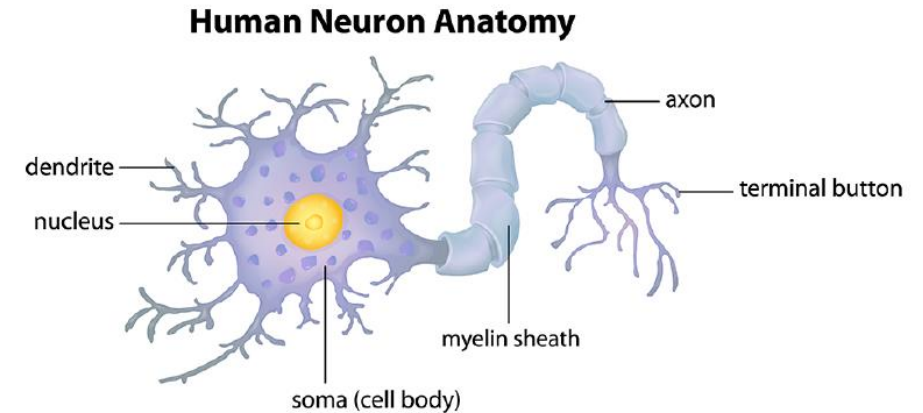
1. Artificial Neural Networks and their relation to biology



A simple NN that takes the weighted sum of the input x and weights w . This weighted sum is then passed through the activation function to determine if the neuron fires

- $f(w_1x_1 + w_2x_2 + \dots + w_nx_n)$
- $f(\sum_{i=1}^n w_ix_i)$
- Or simply, $f(net)$, where $net = \sum_{i=1}^n w_ix_i$

Relation to Biology



The structure of a biological neuron. Neurons are connected to other neurons through their dendrites and enurons.

Activation Functions

$$f(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

1. Artificial Neural Networks and their relation to biology

Activation Functions

1. step:
$$f(\text{net}) = \begin{cases} 1 & \text{if } \text{net} > 0 \\ 0 & \text{otherwise} \end{cases}$$

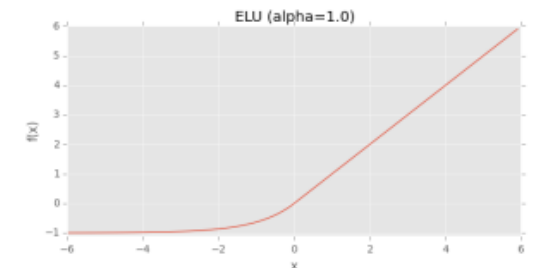
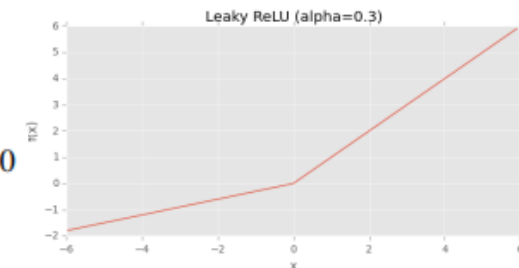
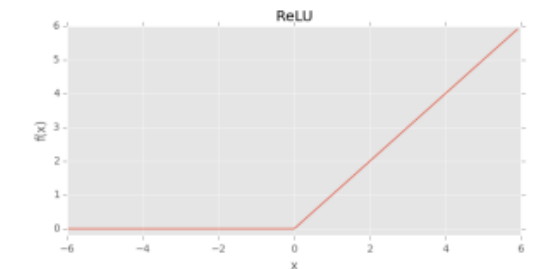
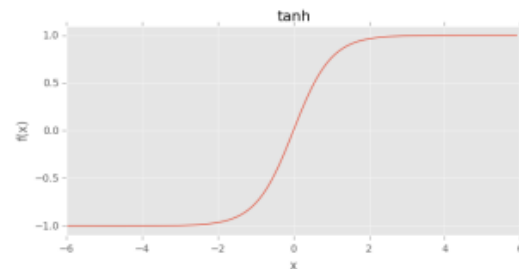
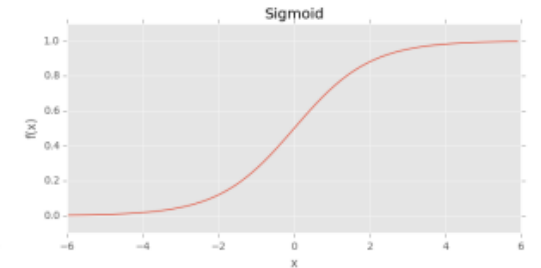
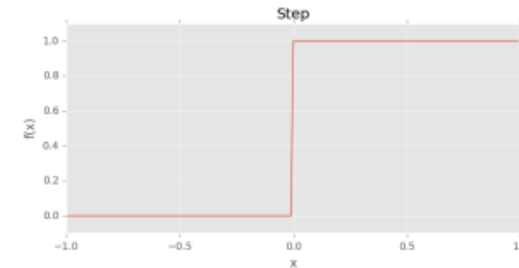
2. sigmoid:
$$t = \sum_{i=1}^n w_i x_i \quad s(t) = 1/(1 + e^{-t})$$

3. tanh:
$$f(z) = \tanh(z) = (e^z - e^{-z}) / (e^z + e^{-z})$$

4. Rectified Linear Unit (ReLU)
$$f(x) = \max(0, x)$$

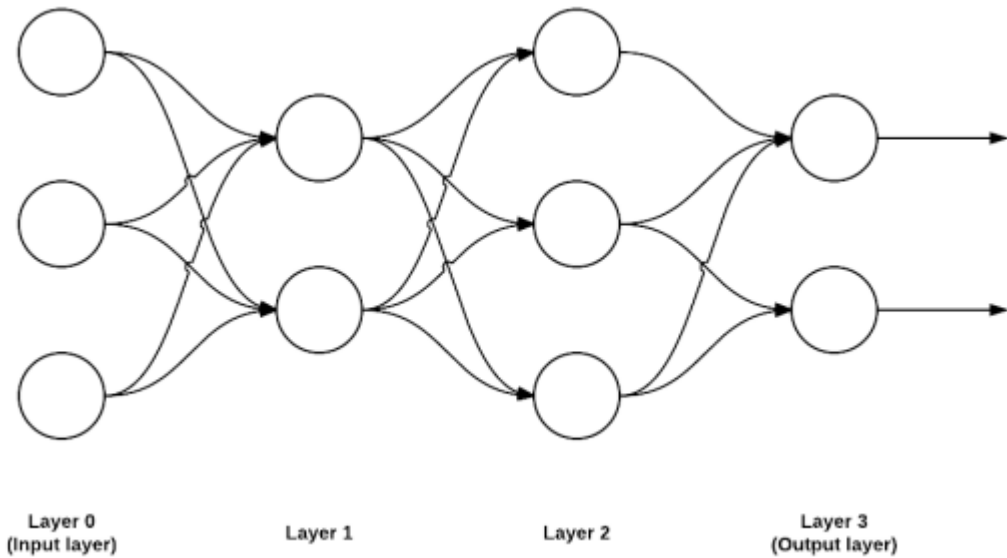
5. Leaky ReLUs
$$f(\text{net}) = \begin{cases} \text{net} & \text{if } \text{net} \geq 0 \\ \alpha \times \text{net} & \text{otherwise} \end{cases}$$

6. Exponential Linear Units (ELUs)
$$f(\text{net}) = \begin{cases} \text{net} & \text{if } \text{net} \geq 0 \\ \alpha \times (\exp(\text{net}) - 1) & \text{otherwise} \end{cases}$$

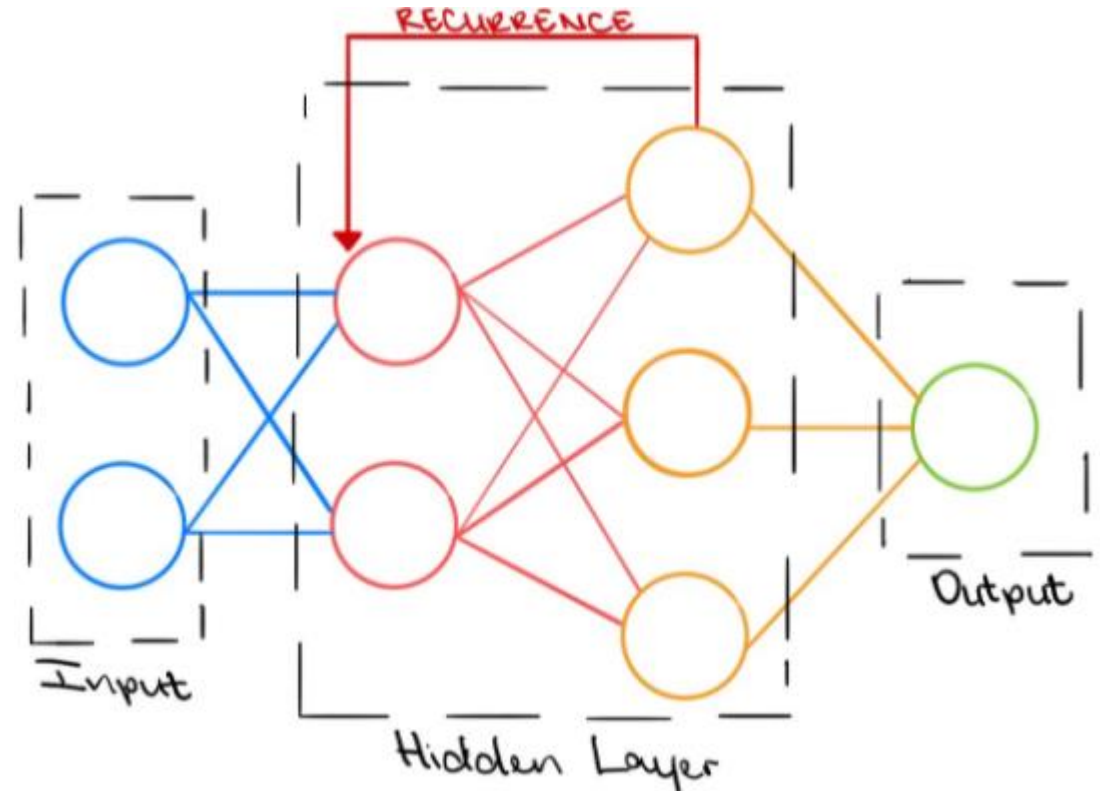


1. Artificial Neural Networks and their relation to biology

Feedforward Network Architectures



Recurrent neural networks



Outline

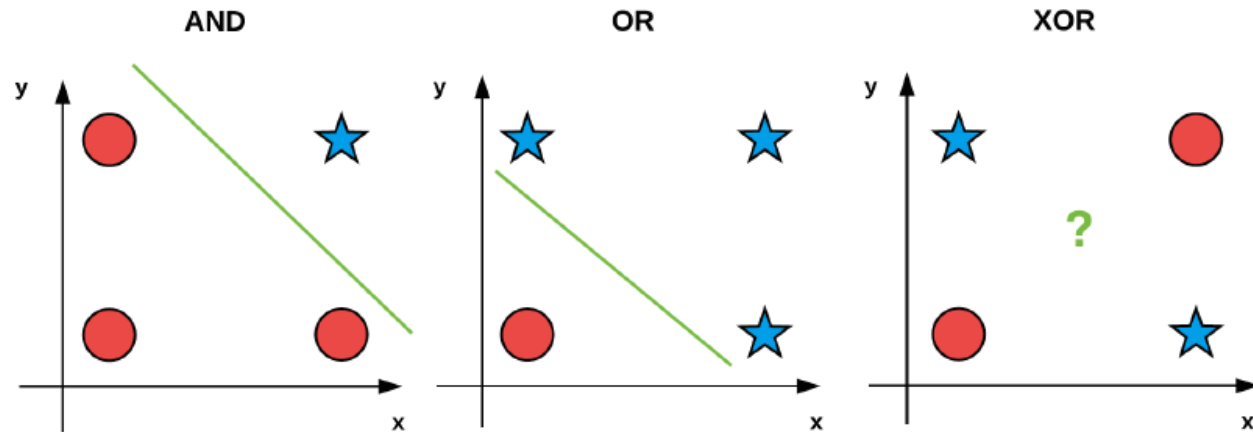
1. Artificial Neural Networks and their relation to biology
- 2. The seminal Perceptron algorithm**
3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently
4. How to train neural networks using the Keras library

2. The Perceptron algorithm

AND, OR, and XOR Datasets

x_0	x_1	$x_0 \& x_1$	x_0	x_1	$x_0 x_1$	x_0	x_1	$x_0 \wedge x_1$
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

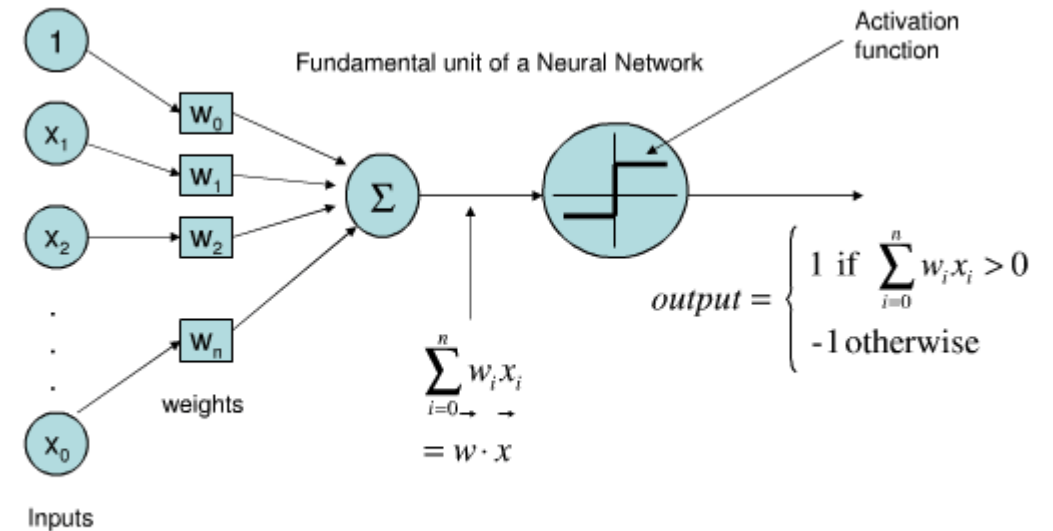
- The AND and OR bitwise datasets are **linearly separable** (we can draw a single line (green) that separates the two classes)
- For XOR bitwise dataset → it is impossible to draw a single line that separates the two classes (**nonlinearly separable dataset**).



- Perceptron algorithm can correctly classify the AND and OR functions but fails to classify the XOR data

2. The Perceptron algorithm

Training a Perceptron is a fairly straightforward operation. Our goal is to obtain a set of weights \mathbf{w} that accurately classifies each instance in our training set. **In order to train our Perceptron, we iteratively feed the network our training data multiple times. Each time the network has seen the full set of training data, we say an epoch has passed.** It normally takes many epochs until a weight vector \mathbf{w} can be learned to linearly separate our two classes of data.



Source: stackexchange.com

Architecture of the Perceptron network

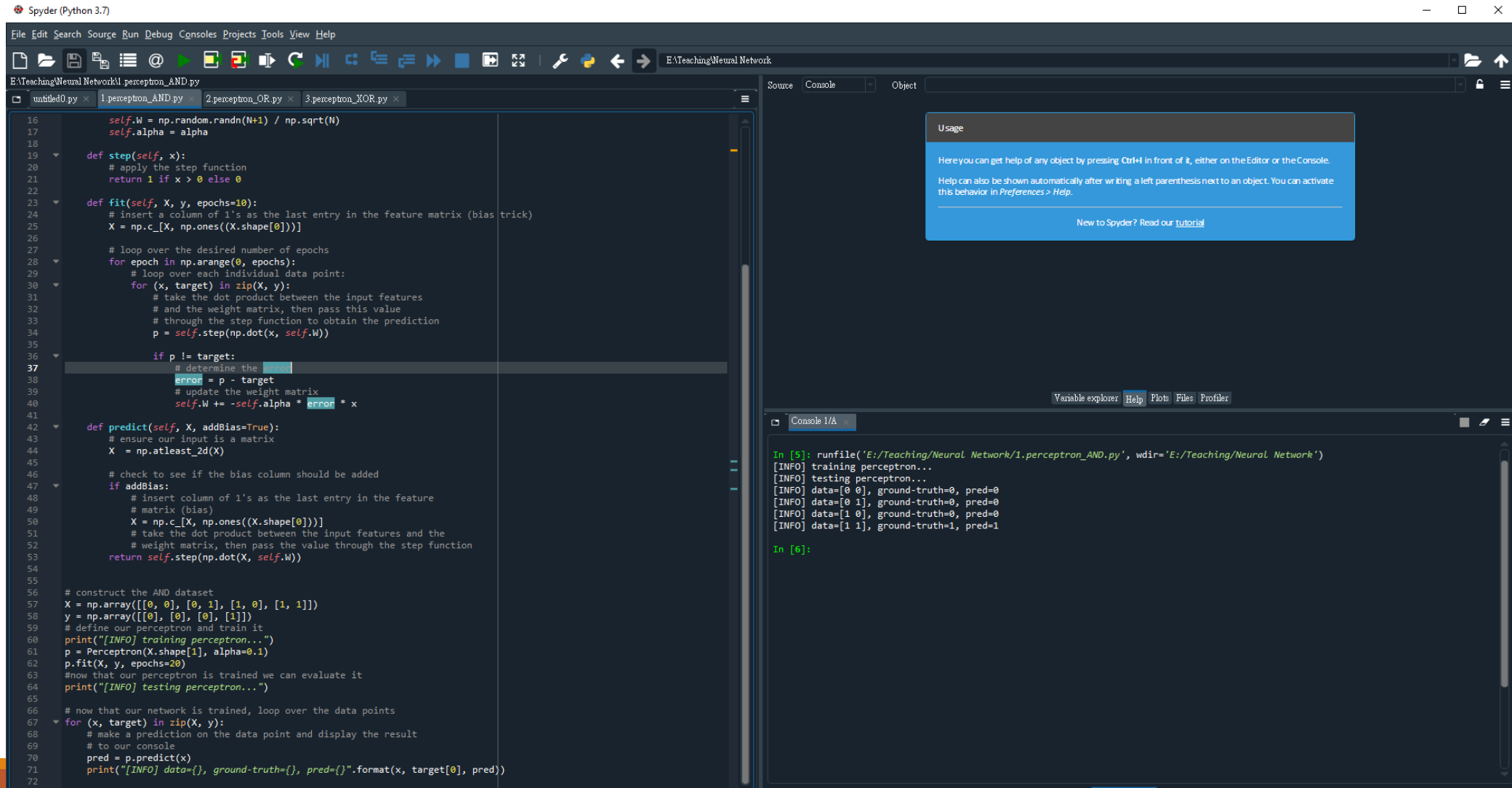
2. The Perceptron algorithm

Perceptron Training Procedure and the Delta Rule

1. Initialize our weight vector \mathbf{w} with small random values
2. Until Perceptron converges:
 - (a) Loop over each feature vector \mathbf{x}_j and true class label d_j in our training set D
 - (b) Take \mathbf{x} and pass it through the network, calculating the output value: $y_j = f(\mathbf{w}(t) \cdot \mathbf{x}_j)$
 - (c) Update the weights \mathbf{w} : $w_i(t+1) = w_i(t) + \eta(d_j - y_j)x_{j,i}$ for all features $0 \leq i \leq n$

η : learning rate

2. The Perceptron algorithm



The screenshot shows the Spyder Python IDE with the following components:

- Editor:** Contains the implementation of the Perceptron algorithm. The code defines a `Perceptron` class with methods `step`, `fit`, and `predict`. It then constructs an AND dataset and trains the perceptron for 20 epochs.
- Console:** Displays the output of the code execution. It shows the training process and the results of testing the trained perceptron on the AND dataset.

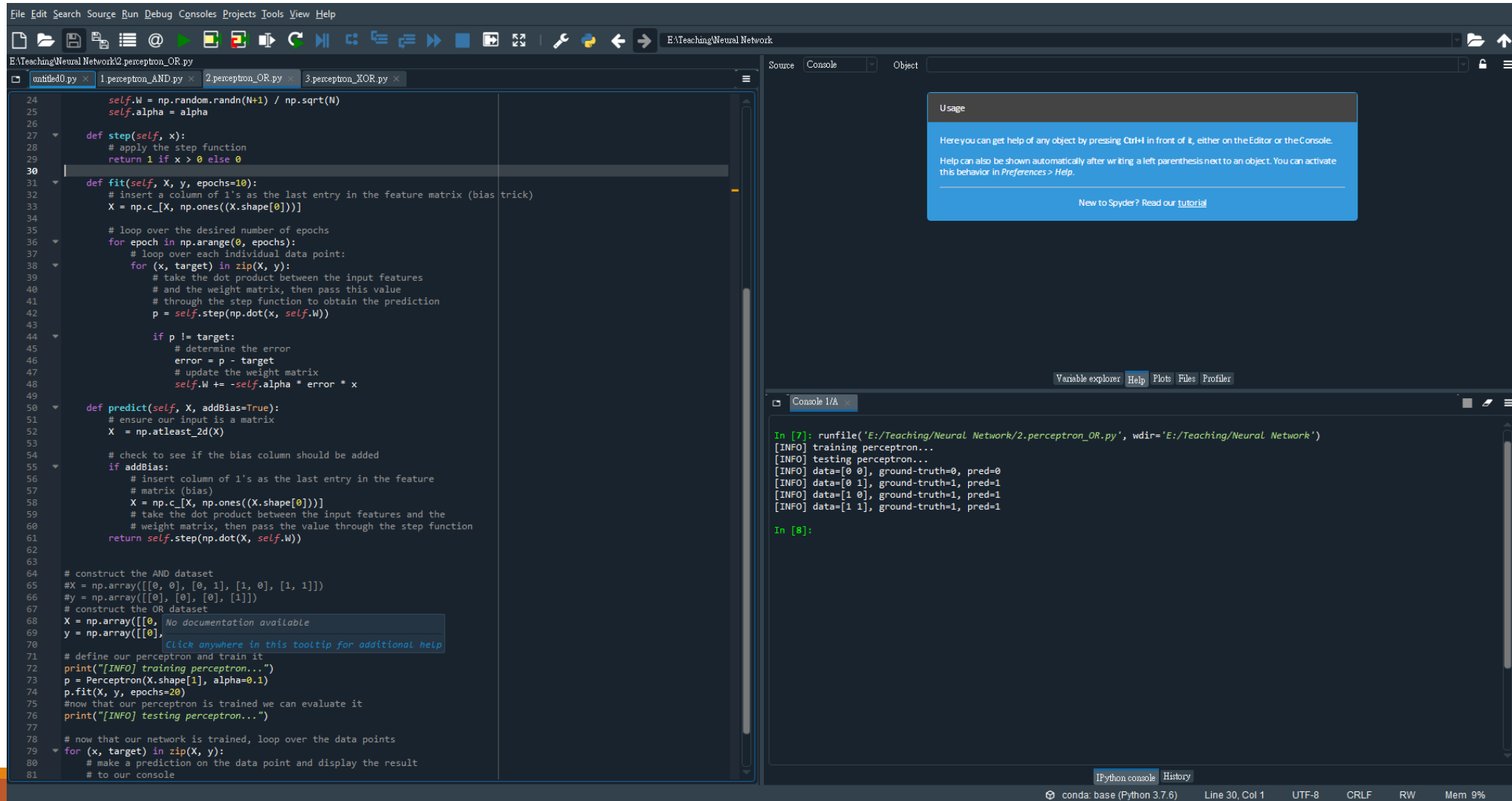
```
16 self.W = np.random.randn(N+1) / np.sqrt(N)
17 self.alpha = alpha
18
19 def step(self, x):
20     # apply the step function
21     return 1 if x > 0 else 0
22
23 def fit(self, X, y, epochs=10):
24     # insert a column of 1's as the last entry in the feature matrix (bias trick)
25     X = np.c_[X, np.ones((X.shape[0]))]
26
27     # loop over the desired number of epochs
28     for epoch in np.arange(0, epochs):
29         # loop over each individual data point:
30         for (x, target) in zip(X, y):
31             # take the dot product between the input features
32             # and the weight matrix, then pass this value
33             # through the step function to obtain the prediction
34             p = self.step(np.dot(x, self.W))
35
36             if p != target:
37                 # determine the error
38                 error = p - target
39                 # update the weight matrix
40                 self.W += -self.alpha * error * x
41
42 def predict(self, X, addBias=True):
43     # ensure our input is a matrix
44     X = np.atleast_2d(X)
45
46     # check to see if the bias column should be added
47     if addBias:
48         # insert column of 1's as the last entry in the feature
49         # matrix (bias)
50         X = np.c_[X, np.ones((X.shape[0]))]
51     # take the dot product between the input features and the
52     # weight matrix, then pass the value through the step function
53     return self.step(np.dot(X, self.W))
54
55
56 # construct the AND dataset
57 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
58 y = np.array([0, 0, 1, 1])
59 # define our perceptron and train it
60 print("[INFO] training perceptron...")
61 p = Perceptron(X.shape[1], alpha=0.1)
62 p.fit(X, y, epochs=20)
63 #now that our perceptron is trained we can evaluate it
64 print("[INFO] testing perceptron...")
65
66 # now that our network is trained, loop over the data points
67 for (x, target) in zip(X, y):
68     # make a prediction on the data point and display the result
69     # to our console
70     pred = p.predict(x)
71     print("[INFO] data={}, ground-truth={}, pred={}".format(x, target[0], pred))
72
```

Console Output:

```
In [5]: runfile('E:/Teaching/Neural Network/1.perceptron_AND.py', wdir='E:/Teaching/Neural Network')
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=0
[INFO] data=[0 1], ground-truth=0, pred=0
[INFO] data=[1 0], ground-truth=0, pred=0
[INFO] data=[1 1], ground-truth=1, pred=1

In [6]:
```

2. The Perceptron algorithm



```
File Edit Search Source Run Debug Consoles Projects Tools View Help
E:\Teaching\Neural Network\2.perceptron_OR.py
untitled0.py 1.perceptron_AND.py 2.perceptron_OR.py 3.perceptron_XOR.py

24 self.W = np.random.randn(N+1) / np.sqrt(N)
25 self.alpha = alpha
26
27 def step(self, x):
28     # apply the step function
29     return 1 if x > 0 else 0
30
31 def fit(self, X, y, epochs=10):
32     # insert a column of 1's as the last entry in the feature matrix (bias trick)
33     X = np.c_[X, np.ones((X.shape[0]))]
34
35     # loop over the desired number of epochs
36     for epoch in np.arange(0, epochs):
37         # loop over each individual data point:
38         for (x, target) in zip(X, y):
39             # take the dot product between the input features
40             # and the weight matrix, then pass this value
41             # through the step function to obtain the prediction
42             p = self.step(np.dot(x, self.W))
43
44             if p != target:
45                 # determine the error
46                 error = p - target
47                 # update the weight matrix
48                 self.W += -self.alpha * error * x
49
50 def predict(self, X, addBias=True):
51     # ensure our input is a matrix
52     X = np.atleast_2d(X)
53
54     # check to see if the bias column should be added
55     if addBias:
56         # insert column of 1's as the last entry in the feature
57         # matrix (bias)
58         X = np.c_[X, np.ones((X.shape[0]))]
59     # take the dot product between the input features and the
60     # weight matrix, then pass the value through the step function
61     return self.step(np.dot(X, self.W))
62
63
64 # construct the AND dataset
65 #X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
66 #y = np.array([0, 0, 1, 1])
67 # construct the OR dataset
68 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
69 y = np.array([0, 1, 1, 1])
70
71 # define our perceptron and train it
72 print("[INFO] training perceptron...")
73 p = Perceptron(X.shape[1], alpha=0.1)
74 p.fit(X, y, epochs=20)
75 #now that our perceptron is trained we can evaluate it
76 print("[INFO] testing perceptron...")
77
78 # now that our network is trained, loop over the data points
79 for (x, target) in zip(X, y):
80     # make a prediction on the data point and display the result
81     # to our console
```

Usage

Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our [tutorial](#)

Variable explorer Help Plots Files Profiler

Console 1/A

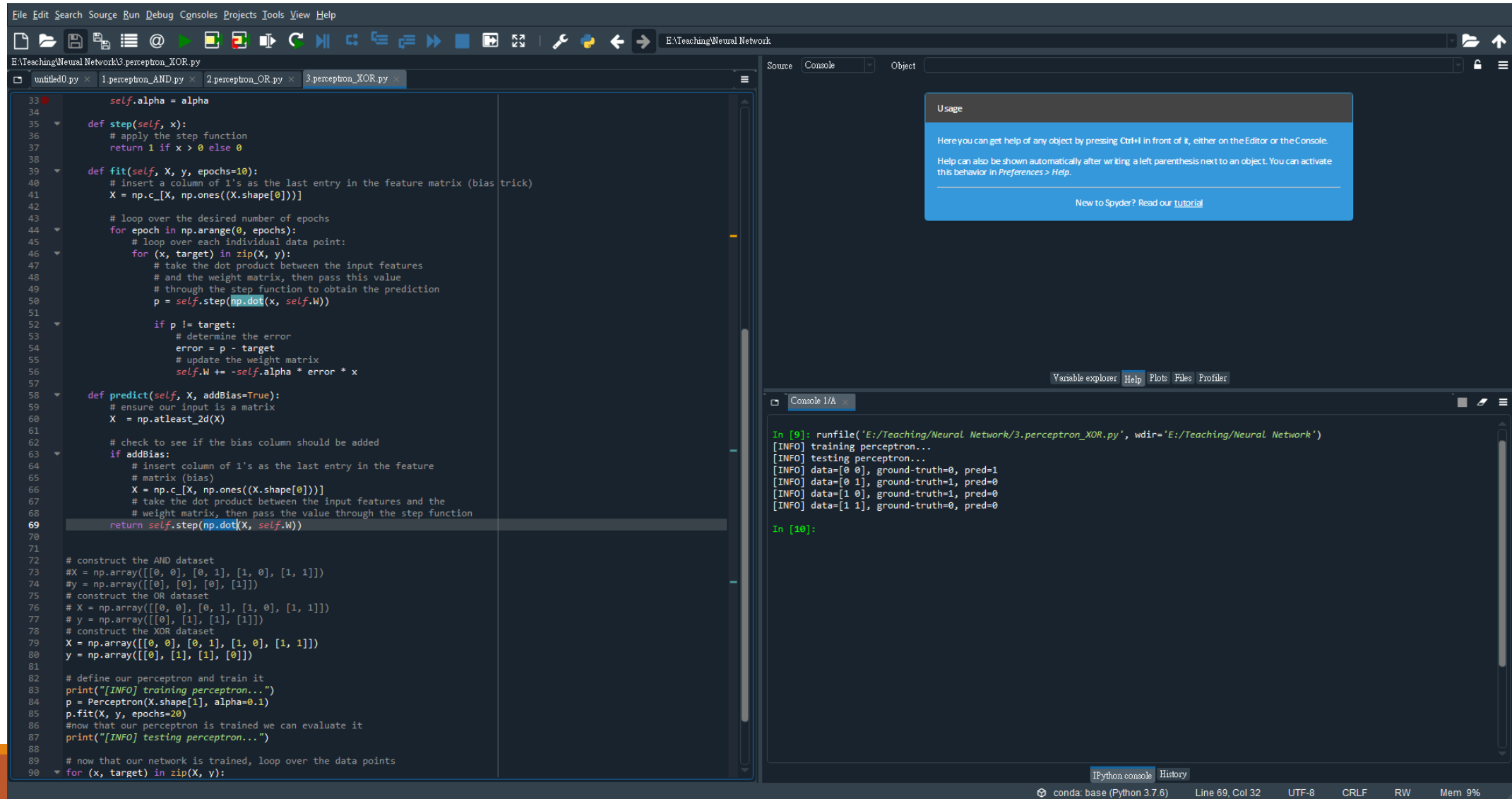
```
In [7]: runfile('E:/Teaching/Neural Network/2.perceptron_OR.py', wdir='E:/Teaching/Neural Network')
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=0
[INFO] data=[0 1], ground-truth=1, pred=1
[INFO] data=[1 0], ground-truth=1, pred=1
[INFO] data=[1 1], ground-truth=1, pred=1

In [8]:
```

Python console History

conda: base (Python 3.7.6) Line 30, Col 1 UTF-8 CRLF RW Mem 9%

2. The Perceptron algorithm



```
File Edit Search Source Run Debug Consoles Projects Tools View Help
untitled0.py 1.perceptron_AND.py 2.perceptron_OR.py 3.perceptron_XOR.py

33 self.alpha = alpha
34
35 def step(self, x):
36     # apply the step function
37     return 1 if x > 0 else 0
38
39 def fit(self, X, y, epochs=10):
40     # insert a column of 1's as the last entry in the feature matrix (bias trick)
41     X = np.c_[X, np.ones((X.shape[0]))]
42
43     # loop over the desired number of epochs
44     for epoch in np.arange(0, epochs):
45         # loop over each individual data point:
46         for (x, target) in zip(X, y):
47             # take the dot product between the input features
48             # and the weight matrix, then pass this value
49             # through the step function to obtain the prediction
50             p = self.step(np.dot(x, self.W))
51
52             if p != target:
53                 # determine the error
54                 error = p - target
55                 # update the weight matrix
56                 self.W += -self.alpha * error * x
57
58 def predict(self, X, addBias=True):
59     # ensure our input is a matrix
60     X = np.atleast_2d(X)
61
62     # check to see if the bias column should be added
63     if addBias:
64         # insert column of 1's as the last entry in the feature
65         # matrix (bias)
66         X = np.c_[X, np.ones((X.shape[0]))]
67     # take the dot product between the input features and the
68     # weight matrix, then pass the value through the step function
69     return self.step(np.dot(X, self.W))
70
71
72 # construct the AND dataset
73 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
74 y = np.array([0, 0, 1, 1])
75 # construct the OR dataset
76 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
77 y = np.array([0, 1, 1, 1])
78 # construct the XOR dataset
79 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
80 y = np.array([0, 1, 1, 0])
81
82 # define our perceptron and train it
83 print("[INFO] training perceptron...")
84 p = Perceptron(X.shape[1], alpha=0.1)
85 p.fit(X, y, epochs=20)
86 #now that our perceptron is trained we can evaluate it
87 print("[INFO] testing perceptron...")
88
89 # now that our network is trained, loop over the data points
90 for (x, target) in zip(X, y):
```

Usage

Here you can get help of any object by pressing Ctrl+H in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in [Preferences > Help](#).

[New to Spyder? Read our tutorial](#)

Variable explorer Help Plots Files Profiler

Console 1/1

```
In [9]: runfile('E:/Teaching/Neural Network/3.perceptron_XOR.py', wdir='E:/Teaching/Neural Network')
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=1
[INFO] data=[0 1], ground-truth=1, pred=0
[INFO] data=[1 0], ground-truth=1, pred=0
[INFO] data=[1 1], ground-truth=0, pred=0

In [10]:
```

Python console History

conda: base (Python 3.7.6) Line 69, Col 32 UTF-8 CRLF RW Mem 9%

2. The Perceptron algorithm

What is a function in Python?

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

Syntax of Function

```
def function_name(parameters):  
    """docstring"""  
    statement(s)
```

Above shown is a function definition that consists of the following components.

1. Keyword `def` that marks the start of the function header.
2. A function name to uniquely identify the function. Function naming follows the same [rules of writing identifiers in Python](#).
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (`:`) to mark the end of the function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
7. An optional `return` statement to return a value from the function.

What is the use of Self in Python?

The self is used to represent the [instance](#) of the class. With this keyword, you can access the attributes and methods of the [class in python](#). It binds the attributes with the given arguments. The reason why we use self is that Python does not use the '@' syntax to refer to instance attributes. Join our [Master Python programming](#) course to know more. In Python, we have methods that make the instance to be passed automatically, but not received automatically.

Example:

```
1  class food():  
2  
3  # init method or constructor  
4  def __init__(self, fruit, color):  
5      self.fruit = fruit  
6      self.color = color  
7  
8  def show(self):  
9      print("fruit is", self.fruit)  
10     print("color is", self.color )  
11  
12     apple = food("apple", "red")  
13     grapes = food("grapes", "green")  
14  
15     apple.show()  
16     grapes.show()
```

<https://www.edureka.co/blog/self-in-python/>

2. The Perceptron algorithm

Python zip()

In this tutorial, we will learn about the Python zip() function with the help of examples.

The `zip()` function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.

Example

```
languages = ['Java', 'Python', 'JavaScript']
versions = [14, 3, 6]

result = zip(languages, versions)
print(list(result))

# Output: [('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

Syntax of zip()

The syntax of the `zip()` function is:

```
zip(*iterables)
```

<https://www.programiz.com/python-programming/methods/built-in/zip>

numpy.dot

`numpy.dot(a, b, out=None)`

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using `matmul` or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to `multiply` and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where $M \geq 2$), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

$$\text{dot}(a, b)[i,j,k,m] = \text{sum}(a[i,j,:]) * b[k,:,m]$$

Parameters: *a* : *array_like*

First argument.

b : *array_like*

Second argument.

out : *ndarray, optional*

Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for `dot(a,b)`. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns: *output* : *ndarray*

Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned.

Raises: *ValueError*

If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

<https://numpy.org/doc/stable/reference/generated/numpy.dot.html>

Outline

1. Artificial Neural Networks and their relation to biology
2. The seminal Perceptron algorithm
- 3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently**
4. How to train neural networks using the Keras library

3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently

Backpropagation can be considered the cornerstone of modern neural networks and deep learning.

The backpropagation algorithm consists of two phases:

1. The forward pass where our inputs are passed through the network and output predictions obtained (also known as the propagation phase).
2. The backward pass where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule to update the weights in our network (also known as the weight update phase).

3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently

The Forward Pass

Let's compute the inputs to the three nodes in the hidden layers:

1. $\sigma((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$
2. $\sigma((0 \times -0.097) + (1 \times -0.165) + (1 \times 0.542)) = 0.593$
3. $\sigma((0 \times 0.457) + (1 \times -0.165) + (1 \times -0.331)) = 0.378$

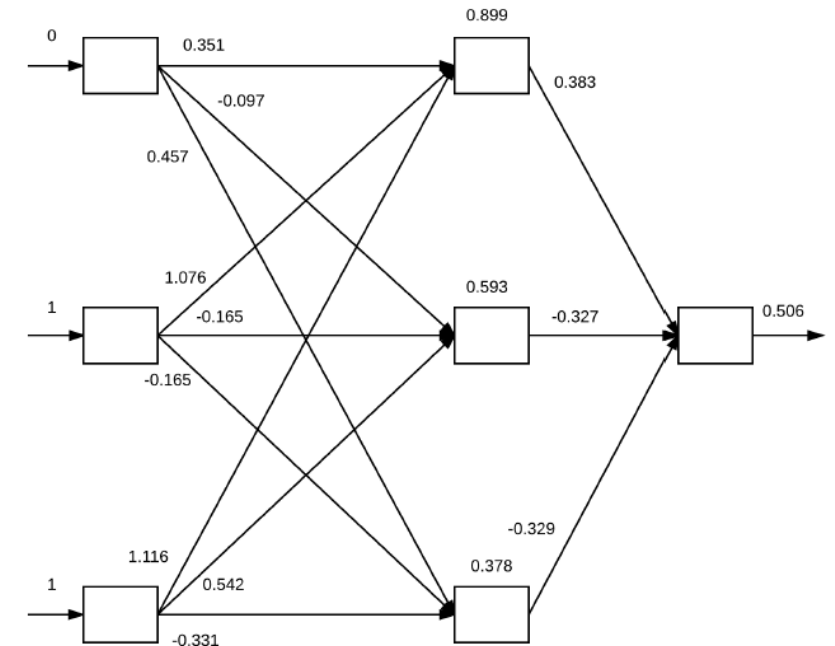
$$\sigma((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$$

$$f(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

The Backward Pass

In order to apply the backpropagation algorithm, our activation function must be **differentiable** so that we can compute the **partial derivative** of the error with respect to a given weight w_{0j} , $loss(E)$, node output o_j , and network output net_j .

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}}$$



An example of the forward propagation pass. The input vector $[0,1,1]$ is presented to the network. The dot product between the inputs and weights are taken, followed by applying the sigmoid activation function to obtain the values in the hidden layer (0.899, 0.593, and 0.378, respectively). Finally, the dot product and sigmoid activation function is computed for the final layer, yielding an output of 0.506. Applying the step function to 0.506 yields 1, which is indeed the correct target class label.

3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently

The screenshot displays an IDE with a Python script for a neural network and its execution output in the console.

Python Script (backpropagation_XOR.py):

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Dec  9 02:31:12 2021
4
5  @author: user
6  """
7
8
9  import numpy as np
10 # import matplotlib.pyplot as plt
11
12 class NeuralNetwork:
13
14     def __init__(self, layers, alpha=0.1):
15         self.W = [] # list W is empty and initialize later
16
17         self.layers = layers
18         self.alpha = alpha
19         # layers: A list of integers which represents the actual architecture of the
20         # feedforward network. For example, a value of [2, 2, 1] would imply
21         # that our first layer has 2 nodes, our hidden layer has 2 nodes, and our
22         # final output layer has one node.
23         # alpha: specify the learning rate of our neural network.
24         # This value is applied during the weight update phase
25         # start looping from the index of the first layers
26         # but stop before we reach the last 2 layers
27
28         # initialize W
29         # start looping over the number of layers in the network (i.e., len(layers)),
30         # but we stop before the final two layer
31         for i in np.arange(0, len(layers) - 2):
32             # randomly initialize a weight matrix connecting the number of nodes
33             # in each respective layer together, adding an extra node for the
34             # bias.
35             w = np.random.randn(layers[i] + 1, layers[i + 1] + 1)
36             # Each layer in the network is randomly initialized by constructing an MxN weight matrix by
37             # sampling values from a standard, normal distribution. The matrix is MxN since we wish
38             # to connect every node in current layer to every node in the next layer.
39             self.W.append(w / np.sqrt(layers[i]))
40             # scale W by dividing by the square root of the number of nodes in the current layer, thereby
41             # normalizing the variance of each neuron's output
42
43         # the last 2 layers are a special case where the input connections need
44         # a bias term but the output does not
45         w = np.random.randn(layers[-2] + 1, layers[-1])
46         self.W.append(w / np.sqrt(layers[-2]))
47
48         # this function is useful for debugging:
49         def __repr__(self):
50             # construct and return a string that represents the network
51             # architecture
52             return "NeuralNetwork: {}".format("-".join(str(l) for l in self.layers))
53
54         def sigmoid(self, x):
55             # compute and return the sigmoid activation value for a given inputs
56             return 1.0 / (1 + np.exp(-x))
```

Console Output:

```
In [28]: runfile('E:/Teaching/Neural Network/4.backpropagation_XOR.py', wdir='E:/Teaching/Neural Network')
NeuralNetwork: 2-2-1
[INFO] epoch=1, loss=0.5124964
[INFO] epoch=100, loss=0.4986981
[INFO] epoch=200, loss=0.4933560
[INFO] epoch=300, loss=0.4697747
[INFO] epoch=400, loss=0.4195478
[INFO] epoch=500, loss=0.3628898
[INFO] epoch=600, loss=0.3127687
[INFO] epoch=700, loss=0.2696377
[INFO] epoch=800, loss=0.2328873
[INFO] epoch=900, loss=0.2045619
[INFO] epoch=1000, loss=0.1844625
[INFO] data=[0 0], ground-truth=0, pred=0.1082, step=0
[INFO] data=[0 1], ground-truth=1, pred=0.8381, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.8352, step=1
[INFO] data=[1 1], ground-truth=0, pred=0.5512, step=1

In [29]:
```

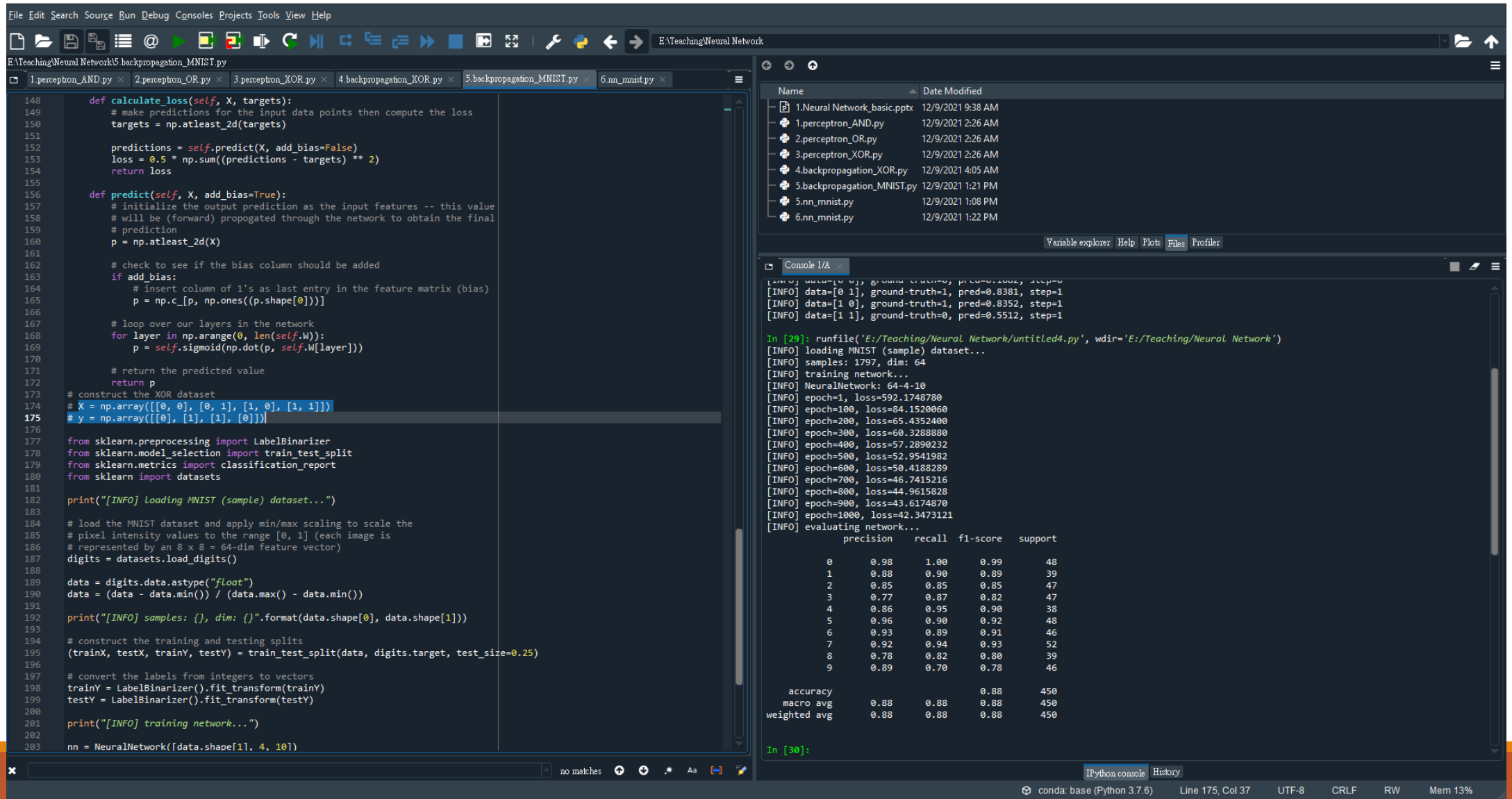
The IDE interface includes a menu bar (File, Edit, Search, Source, Run, Debug, Consoles, Projects, Tools, View, Help), a toolbar with icons for file operations and execution, and a status bar at the bottom showing 'conda: base (Python 3.7.6)', 'Line 199, Col 30', 'UTF-8', 'CRLF', 'RW', and 'Mem 13%'.

3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently



MNIST dataset: handwritten digit recognition. This subset of the MNIST dataset is built-into the scikit-learn library and includes 1,797 example digits, each of which are 88 grayscale images (the original images are 28x28). When flattened, these images are represented by an 88 = 64-dim vector.

3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently



The screenshot displays a Python script in an IDE, likely JupyterLab, with the following components:

- File Explorer:** Shows a project structure with files like `1.neural_network_basic.py`, `1.perceptron_AND.py`, `2.perceptron_OR.py`, `3.perceptron_XOR.py`, `4.backpropagation_XOR.py`, `5.backpropagation_MNIST.py`, and `6.nn_mnist.py`.
- Code Editor:** Contains the implementation of a neural network for MNIST digit recognition. The script includes:
 - `calculate_loss` function: Computes the loss based on predictions and targets.
 - `predict` function: Performs forward propagation to obtain the final prediction.
 - `train` function: Loads the MNIST dataset, scales it, and splits it into training and testing sets. It then trains the neural network for 1000 epochs.
- Console:** Displays the output of the script, including the training progress and a confusion matrix.

Console Output:

```
In [29]: runfile('E:/Teaching/Neural Network/untitled4.py', wdir='E:/Teaching/Neural Network')
[INFO] loading MNIST (sample) dataset...
[INFO] samples: 1797, dim: 64
[INFO] training network...
[INFO] NeuralNetwork: 64-4-10
[INFO] epoch=1, loss=592.1748780
[INFO] epoch=100, loss=84.1520060
[INFO] epoch=200, loss=65.4352400
[INFO] epoch=300, loss=60.3288880
[INFO] epoch=400, loss=57.2890232
[INFO] epoch=500, loss=52.9541982
[INFO] epoch=600, loss=50.4188289
[INFO] epoch=700, loss=46.7415216
[INFO] epoch=800, loss=44.9615828
[INFO] epoch=900, loss=43.6174870
[INFO] epoch=1000, loss=42.3473121
[INFO] evaluating network...
precision    recall  f1-score   support
0           0.98      1.00      0.99        48
1           0.88      0.90      0.89        39
2           0.85      0.85      0.85        47
3           0.77      0.87      0.82        47
4           0.86      0.95      0.90        38
5           0.96      0.90      0.92        48
6           0.93      0.89      0.91        46
7           0.92      0.94      0.93        52
8           0.78      0.82      0.80        39
9           0.89      0.70      0.78        46

accuracy          0.88      0.88      0.88        450
macro avg         0.88      0.88      0.88        450
weighted avg      0.88      0.88      0.88        450

In [30]:
```

The status bar at the bottom indicates the environment is `conda: base (Python 3.7.6)`, the cursor is at `Line 175, Col 37`, and the encoding is `UTF-8`.

Outline

1. Artificial Neural Networks and their relation to biology
2. The seminal Perceptron algorithm
3. The backpropagation algorithm and how it can be used to train multi-layer neural networks efficiently
- 4. How to train neural networks using the Keras library**

4. How to train neural networks using the Keras library

The screenshot displays a Jupyter Notebook interface with a Python script for training a neural network using Keras. The script is divided into several sections: data loading, preprocessing, model definition, training, and evaluation. The training process is monitored via a plot of loss and accuracy over 100 epochs. The console output shows the progress of the training, including loss and accuracy values for each epoch, and a final evaluation report.

```
17 # help="path to the output loss/accuracy plot")
18 # args = vars(ap.parse_args())
19
20 # grab the MNIST dataset (if this is your first time running this
21 # script, the download may take a minute -- the 55MB MNIST dataset
22 # will be downloaded)
23 print("[INFO] Loading MNIST (full) dataset...")
24 #dataset = datasets.fetch_mldata("MNIST Original")
25
26
27
28 dataset = fetch_openml('mnist_784')
29
30 # scale the raw pixel intensities to the range [0, 1.0], then
31 # construct the training and testing splits
32 data = dataset.data.astype("float") / 255.0
33 (trainX, testX, trainY, testY) = train_test_split(data,
34 dataset.target, test_size=0.25)
35 # convert the labels from integers to vectors
36 lb = LabelBinarizer()
37 trainY = lb.fit_transform(trainY)
38 testY = lb.transform(testY)
39 # define the 784-256-128-10 architecture using Keras
40 model = Sequential()
41 model.add(Dense(256, input_shape=(784,)), activation="sigmoid")
42 model.add(Dense(128, activation="sigmoid"))
43 model.add(Dense(10, activation="softmax"))
44
45 # train the model using SGD
46 print("[INFO] training network...")
47 sgd = SGD(0.01)
48 model.compile(loss="categorical_crossentropy", optimizer=sgd,
49 metrics=["accuracy"])
50 H = model.fit(trainX, trainY, validation_data=(testX, testY), epochs=100, batch_size=128)
51 # evaluate the network
52 print("[INFO] evaluating network...")
53 predictions = model.predict(testX, batch_size=128)
54 print(classification_report(testY.argmax(axis=1),
55 predictions.argmax(axis=1),
56 target_names=[str(x) for x in lb.classes_]))
57
58
59 # plot the training loss and accuracy
60 plt.style.use("ggplot")
61 plt.figure()
62 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
63 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
64 plt.plot(np.arange(0, 100), H.history["accuracy"], label="train_acc")
65 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
66 plt.title("Training Loss and Accuracy")
67 plt.xlabel("Epoch #")
68 plt.ylabel("Loss/Accuracy")
69 plt.legend()
70 #plt.savefig(args["output"])
71
```

The plot shows the training loss (red line) and validation loss (blue line) over 100 epochs. The training loss decreases from approximately 2.0 to 0.25, while the validation loss decreases from approximately 1.0 to 0.25. The training accuracy (green line) increases from approximately 0.5 to 0.92, and the validation accuracy (purple line) increases from approximately 0.5 to 0.92.

The console output shows the progress of the training, including loss and accuracy values for each epoch. The final evaluation report shows the following results:

	precision	recall	f1-score	support
0	0.95	0.97	0.96	1741
1	0.94	0.97	0.95	2002
2	0.92	0.89	0.90	1742
3	0.92	0.88	0.90	1839
4	0.92	0.93	0.92	1705
5	0.88	0.87	0.88	1537
6	0.93	0.95	0.94	1755
7	0.94	0.93	0.93	1815
8	0.88	0.88	0.88	1662
9	0.90	0.90	0.90	1702
accuracy			0.92	17500
macro avg	0.92	0.92	0.92	17500
weighted avg	0.92	0.92	0.92	17500

Traceback (most recent call last):
File "E:\Teaching\Neural Network\5.nn_mnist.py", line 65, in <module>

See you next week

