Single-Layer and Multi-Layer Perceptrons in Python

Jessica Cervi

Activity Overview

A neural network (NN) is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. The basic unit is the perceptron. Each perceptron has connections, like the synapses in a biological brain, that can compute and transmit a signal to other units (neurons).

In this notebook, we first build a very simple version of a single perceptron by hand, and use it to make a prediction on the Titanic dataset. Then, you'll learn how to handle the implementation of a perceptron using `sklearn`.

This activity is designed to help you apply the machine learning algorithms you have learned using the packages in `Python`. `Python` concepts, instructions, and starter code are embedded within this Jupyter Notebook to help guide you as you progress through the activity. Remember to run the code of each code cell prior to submitting the assignment. Upon completing the activity, we encourage you to compare your work against the solution file to perform a self-assessment.

Index:

Basics of Artificial Neural Networks

Artificial neural networks (ANNs or simply NNs) are computing systems that are inspired by, but not identical to, biological neural networks that constitute animal brains. Such systems learn to perform tasks by considering examples, generally without being programmed with task-specific rules.

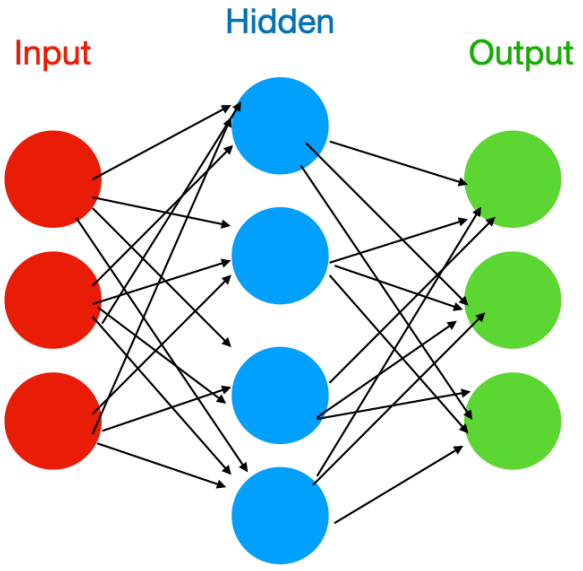
An NN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. The basic unit is the perceptron. Each perceptron has connections, like the synapses in a biological brain, that can compute and transmit a signal to other units (neurons).

In ANN implementations, the "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called *edges*. Neurons and edges typically have a weight that adjusts as the learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a strong signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

Single-Layer and Multi-Layer Perceptrons

A single-layer perceptron (SLP) is a feed-forward network based on a threshold transfer function. SLP is the simplest type of ANN. Even though simple, an SLP can be used to classify linearly separable cases with a binary target (1, 0).

A multi-layer perceptron (MLP) has one or more hidden layers, each having a structure the same as or similar to a single-layer perceptron. A backpropagation algorithm is used to train or learn the weights in an MLP. The backpropagation algorithm consists of two phases: the forward phase, where the activations are propagated from the input to the output layer, and the backward phase, where the error between the observed actual and requested nominal value in the output layer is propagated backwards in order to modify the weights and bias values.



About the Dataset

Reading the Data

The dataset used here is from Kaggle *Titanic: Machine Learning from Disaster (https://www.kaggle.com/c/titanic/data)*.

The training set will be used to build your machine learning models. For the training set, we provide the outcome (also known as the “ground truth”) for each passenger. Our model will be based on “features” like the passengers’ gender and class, and we'll use feature engineering to create new features. We'll use the training set to learn weights in the model.

The test set will be used to see how well your model performs on unseen data. For the test set, we don't provide the ground truth for each passenger. It's our job to predict these outcomes. For each passenger in the test set, use the model you trained to predict whether they survived the sinking of the Titanic.

In the code cell below, we have imported the necessary libraries `NumPy`, `pandas` and the visualization libraries `seaborn` and `matplotlib`. We also have set a random seed for reproducibility of the results.

Next, we read the training and testing datasets and assign them to the dataframe `data_train`.

In [1]:
```
import numpy as np # linear algebra
np.random.seed(10)
import pandas as pd # data processing, CSV file I/O (e.g., pd.read_csv
)
import matplotlib.pyplot as plt
import seaborn as sns

# Required magic to display matplotlib plots in notebooks
%matplotlib inline

#reading the files
data_train = pd.read_csv('./data/train.csv')
```

In the code cell below, we used the function `head()` with argument 4 to display the first four rows of the dataset. You should adjust the code below to display eight rows in the dataset.

In [2]:
```
data_train.head(8)
```

Out[2]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000

Back to top

Problem Setup

In this short activity, we'll manually build a very basic multi-layer perceptron to predict if a set of passengers has survived.

In the code below, we have defined a dictonary `dict_live`, which maps the entries in the column `Survived` from 0 and 1 to `Perished` and `Survived`, respectively.

In [3]:
```
# We define a dictionary to transform the 0,1 values in the
# labels to a String that defines the fate of the passenger
dict_live = {
    0 : 'Perished',
    1 : 'Survived'
}
```

In the code below, fill the ellipsis in the definition of the dictonary `dict_sex` to map the entries in the column `Sex` from `male` and `female` to `0` and `1`, respectively.

In [4]:
```
# We define a dictionary to assign labels 0 or 1
# corresponding to the sex
dict_sex = {
    'male' : 0,
    'female' : 1
}
```

Run the code cell below, to create the column `BSex` with the values in `dict_sex` and append it to the dataframe. After running, you should see this new data as the right-most column in the dataframe.

In [5]:
```
#We apply the dictionary using a lambda function and the pandas .apply
() module
data_train['BSex'] = data_train['Sex'].apply(lambda x: dict_sex[x])

#This line displasy the dataframe with the new column
data_train.head()
```

Out[5]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500

Before implementing the perpeptron in `Python` , we need to define the feature that we are going to use for our prediction.

In the code cell below, fill the ellipsis with `Pclass` and `BSex` to indicate a list with these two values as our two-dimensional feature vector. Note that we used the function `to_numpy()` to convert our features to `NumPy` arrays.

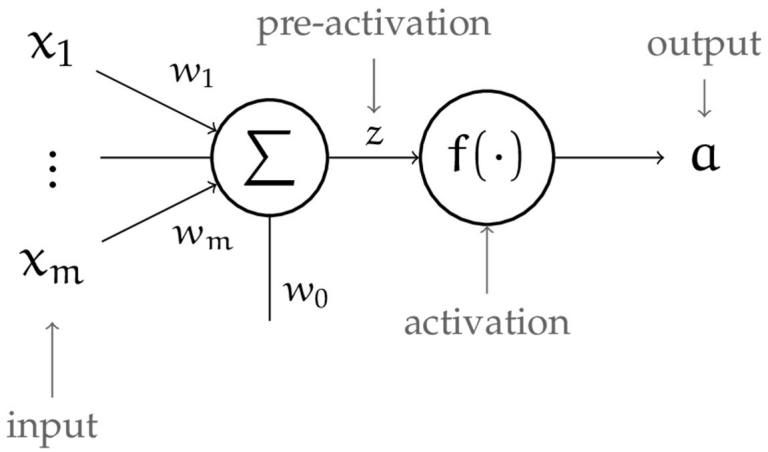
In [6]:
```
# Now the features are a 2 column matrix whose entries are
# the Class (1,2,3) and the Sex (0,1) of the passengers
features = data_train[['Pclass', 'BSex']].to_numpy()
print(features)
```

```
[[3 0]
 [1 1]
 [3 1]
 ...
 [3 1]
 [1 0]
 [3 0]]
```

Back to top

Perceptron

The perceptron is a basic unit or function that mimics the human neuron. It receives a vector (i.e., array) x_i of signals, where i stands for the i-th input; then weighs each of them by a vector of weights w_i. It also adds a *bias*, w_0, to shift the decision boundary away from the origin as needed.



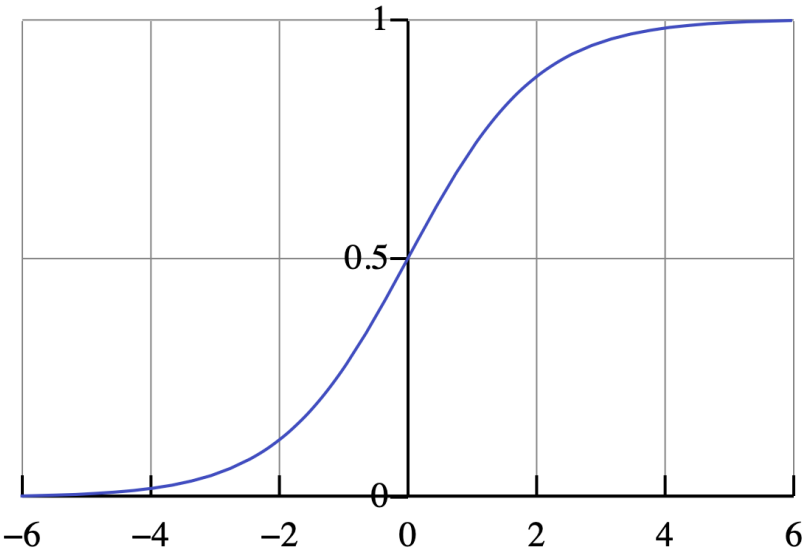
Thus, the intermediate value z is given by $z = w_0 + \sum_{i=1}^{m} w_i x_i = w_0 + \mathbf{w}^T \cdot \mathbf{x}$.

Activation Functions

The perceptron next ignites an output through an activation function acting on the intermediate value z (note: z is sometimes called the *pre-activation* value, since it is fed to the activation function). Activation functions can vary, but the ones that we will consider here are:

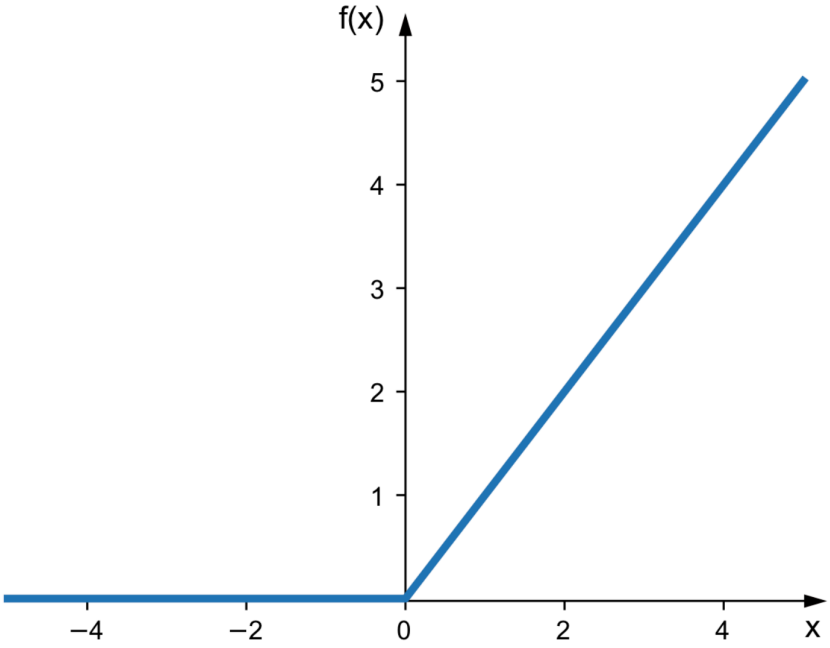
The *sigmoid* Function

$$\phi(z) = \frac{1}{1 + e^{-z}},$$



The Rectified Linear Unit (ReLU)

$$\phi(z) = \max(0, z)\,,$$



Output

Combining the weighted linear combination (that calculates z) with the activation function ϕ, we see that the output a of the perceptron is given by

$$a = \phi(z) = \phi\left(\sum_{i=1}^{n} w_i\, x_i + w_0\right),$$

or, in vector representation:

$$a = \phi(z) = \phi\left(\mathbf{w}^T \cdot \mathbf{x} + w_0\right)\,.$$

Back to top

Implementing the Perceptron

Below, we present you with a simple implementation of the perceptron.

First, we need to define two functions in `Python`, `sigmoid_act` and `ReLu_act`, for the activations.

`sigmoid_act`

The function `sigmoid_act` takes as argument a vector `z` and returns the output `a`.

In [7]:
```
# Define the sigmoid activator
def sigmoid_act(z):
    # sigmoid
    a = 1/(1+ np.exp(-z))
    return a
```

`ReLU_act`

The function `ReLU_act` takes as argument a vector `z` and returns the output `a`.

In [8]:
```
# We may employ the Rectifier Linear Unit (ReLU)
def ReLU_act(z):
    return max(0, z)
```

Now, we are ready to define the perceptron.

In the code cell below, we have defined a function, `perceptron`, that takes as input an array `X` representing the features in the dataframe that we wish to use and the argument `act` selecting the activation function.

Run the code cell below.

In [9]:
```
def perceptron(X, act):
    np.random.seed(1) #seed for reproducibilty
    shapes = X.shape #get the number of (rows, columns)
    n = shapes[0] + shapes[1] #adding the number of rows and colums
    # Generating random weights and bias
    w = 2*np.random.random(shapes) - 0.5 #we want initial w between -1
and 1
    w_0 = np.random.random(1)

    # Initialize the function
    f = w_0[0]
    for i in range(0, X.shape[0]-1): #run over column elements
        for j in range(0, X.shape[1]-1): #run over rows elements
            f += w[i, j]*X[i,j]/n #adding each component of input mult
iplied by corresponding weight
    # Pass it to the activation function and return it as an output
    if act == 'Sigmoid':
        output = sigmoid_act(f)
    elif act == "ReLU":
        output = ReLU_act(f)
    return output
```

An example of an output of the perceptron with the sigmoid activation is given below:

In [10]:
```
print('Output with sigmoid activator: ', perceptron(features, act = 'S
igmoid'))
```

```
Output with sigmoid activator:  0.8331095577932985
```

In the code cell below, try to generate the output by using the ReLU function. Simply observe the code above and fill - in the ellippsis with the name of the function that defined the desired activation.

In [11]:
```
print('Output with ReLU activator: ', perceptron(features, act = "ReLU
"))
```

```
Output with ReLU activator:  1.607827593037905
```

Question

The outputs we have obtained are different for the two activation functions. Can you explain why?

CLICK ON THIS CELL TO TYPE YOUR ANSWER

(Correct answer, not to be shown to the students): The same value z is calculated in both cases, but depending on the activation function, the output a will be different. In the case of ReLU, the z is greater than zero, so the output of the activation function is just the value z. In the case of sigmoid, the value z is mapped to the range $(0, 1)$, so the output a is different.

Back to top

Single-Layer Perceptron in `sklearn`

The library `sklearn` offers a built-in implementation of a single-layer perceptron. Even better, the implementation also knows how to calculate gradients of all parts of the perceptron, and use that information to implement back-propagation.

In this next example, we use a perceptron learner to classify the famous iris dataset (https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html) offered by the sklearn library.

Run the code cell below to import the necessary libraries.

In [12]:
```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

In the next cell, we import the iris dataset from `sklearn`.

In [13]:
```
# Load the iris dataset
iris = datasets.load_iris()

# Create our X and y data
X = iris.data
y = iris.target
```

Run the code bell below to visualize the first five rows of the array `X`.

In [14]:
```
X[:5]
```

Out[14]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

Question

How many features (independent variables) does `X` have?

CLICK ON THIS CELL TO TYPE YOUR ANSWER

(Correct answer, not to be shown to the students): 4

Run the code cell below to visualize the values of `y` for all of our data. We see that some have class label 0, some label 1, and some label 2. Thus we have three different kinds of irises that we are trying to predict. The first 5 of these are the labels corresponding to the first 5 rows of our `X` data above.

In [15]:
```
# View the first 100 observations of our y data
y
```

Out[15]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2
, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Split The Iris Data into Training and Test

Next, we need to split `X` and `y` into a training and a testing set.

This can be achieved by using the function `train_test_split` from `sklearn`.

In the code cell below, fill in the ellipsis by setting the argument `test_size = 0.25`

In [16]:
```
# Split the data into 75% training data and 25% test data


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
25)
X_train.shape, X_test.shape
```

Out[16]:
```
((112, 4), (38, 4))
```

Preprocess the `x` Data by Scaling

It is often standard practice to standardize all the features to have mean equal to zero and unit variance. Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g., Gaussian with 0 mean and unit variance).

We can accomplish this by calling the function `StandardScaler` and by fitting it over the `X_train` set.

Run the code cell below:

In [17]:
```
sc = StandardScaler()
sc.fit(X_train)
```

Out[17]:
```
StandardScaler()
```

Finally, to complete this, we need to apply the scaler to both the `X_train` set and the `X_test` set.

Run the code cell by filling the ellipsis with the `X_test` set.

In [18]:
```
# Apply the scaler to the X training data
X_train_std = sc.transform(X_train)

X_test_std = sc.transform(X_test)
```

Train a Single-Layer Perceptron Model

Next, we need to train the `Perceptron` classifier on the scaled `X_train` data with the corresponding labels in `Y_train` .

Run the code cell below. Inside the classifier `Perceptron` , set the learning rate `eta0` equal to 0.1 and the `random_state` equal to 1. Note that by default, the sklearn `Perceptron` uses a ReLU activation function.

In [19]:
```
# Create a perceptron object with the parameters:max_ieter (epochs) eq
ual to 40 learning rate of 0.1

ppn = Perceptron(max_iter = 1000, eta0 = 0.1, random_state = 0)

# Train the perceptron
ppn.fit(X_train_std, y_train)
```

Out[19]:
```
Perceptron(eta0=0.1)
```

Now we can use this trained perceptron, `ppn`, to predict the results for the **scaled** test data.

In the code cell below, fill in the ellipsis with the name of the variable corresponding to the set above.

In [20]:
```
# Apply the trained perceptron on the X data to make predicts for the
y test dat

y_pred = ppn.predict(X_test_std)
y_pred
```

Out[20]:
```
array([1, 2, 2, 1, 2, 1, 1, 2, 0, 2, 2, 0, 1, 2, 0, 2, 2, 1, 1, 2, 1
, 0,
       2, 1, 0, 0, 2, 1, 1, 1, 2, 0, 0, 0, 1, 0, 1, 1])
```

We can compare the predicted `y` with the true `y`. Run the code cells below to examine these.

In [21]:
```
y_pred
```

Out[21]:
```
array([1, 2, 2, 1, 2, 1, 1, 2, 0, 2, 2, 0, 1, 2, 0, 2, 2, 1, 1, 2, 1
, 0,
       2, 1, 0, 0, 2, 1, 1, 1, 2, 0, 0, 0, 1, 0, 1, 1])
```

In [22]:
```
y_test
```

Out[22]:
```
array([1, 2, 2, 2, 2, 1, 1, 2, 0, 2, 2, 0, 1, 1, 0, 2, 2, 1, 1, 1, 1
, 0,
       2, 1, 0, 0, 2, 1, 1, 1, 2, 0, 0, 0, 1, 0, 1, 1])
```

Finally, we can evaluate the accuracy of our model by using the function `accuracy_score` from `sklearn`, which uses the formula:

$$\text{accuracy} = 1 - \frac{\text{observations predicted wrong}}{\text{total observations}}$$

Run the code below to evaluate the accuracy of our model.

In [23]:
```
# View the accuracy of the model, which is: 1 - (observations predicte
d wrong / total observations)
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.92
```

Train a Multilayer Perceptron (MLP) Model

Finally, we'll train a multilayer perceptron (MLP) using the same training and test data. Run the cell below to train our `mlp` model, with three hidden layers each having 10 hidden nodes.

In [24]:
```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
mlp.fit(X_train_std, y_train)
```

Out[24]:
```
MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
```

In [25]:
```
mlp_y_pred = mlp.predict(X_test_std)
mlp_y_pred
```

Out[25]:
```
array([1, 2, 2, 2, 2, 1, 1, 2, 0, 2, 2, 0, 1, 1, 0, 2, 2, 1, 1, 1, 1
, 0,
       2, 1, 0, 0, 2, 1, 1, 1, 2, 0, 0, 0, 1, 0, 1, 1])
```

In [26]:
```
# View the accuracy of the model, which is: 1 - (observations predicte
d wrong / total observations)
print('Accuracy: %.2f' % accuracy_score(y_test, mlp_y_pred))
```

```
Accuracy: 1.00
```

Which model, the single-layer perceptron or the multi-layer perceptron, performed better on test data? Why?

In [ ]:
```

```