National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS 368: Reinforcement Learning

Lab 01: Introduction to NumPy

Date: 13 September 2024

Time: 02:00 PM-5:00 PM

Instructor: Dr. Zuhair Zafar

Lab 01: Introduction to NumPy

Introduction
This lab is designed to develop a basic understanding of Google Colab and NumPy.

Objectives
Understand the workplace of Google Colab and basic working of NumPy library.

Tools/Software Requirement
Google Colab

Description
Follow the lab manual step by step.

1. Open a new notebook on Google Colab:

a) Open Your Web Browser and type "Google Colab" into your search engine or directly visit the Colab website by going to https://colab.research.google.com/.

b) Click the "Sign In" button located in the upper right corner of the Colab homepage to sign in from your google account. You'll only need to sign in with your Google account if you aren't already signed in.

c) Once you're signed in, you'll be on the Colab landing page. To create a new notebook, Click the "File" menu and select "New Notebook" from the dropdown menu.

d) After creating a new notebook, you'll be presented with a new tab. You are now ready to write and execute Python code in your Colab notebook.

2. Naming your new notebook:

By default, your new notebook will have an auto-generated name like "UntitledX.ipynb" where X is a number. To provide a name to your notebook, click on the "UntitledX" text at the top of the notebook. This will open a text input field. You can enter your desired name for the notebook in this text field and then press enter to save the name.

3. Add a code cell:

To write and execute code in your notebook, you need to add a code cell. You can add a code cell by clicking the "+ CODE" button (located below the menu).

4. Write code for printing a message:

a) In the newly added code cell, you can write Python code. To print "Hello, World!", type the following code in the cell:

```
print("Hello, World!")
```

b) To execute the code you've written, you can do one of the following:
   i. Click the "Play" button (a triangle icon) located to the left of the code cell.
   ii. Use the keyboard shortcut Shift+Enter (press Shift and Enter simultaneously).

c) After running the code cell, you will see the output displayed below the cell. It should show "Hello, World!" as the output.

5. Save your work:

If you want to save your notebook at this point, you can do so by clicking on "File" and selecting "Save" or using the keyboard shortcut Ctrl+S. Colab automatically saves your work in Google Drive.

6. Numpy

NumPy is the main package for scientific computing in Python. It is maintained by a large community (www.numpy.org). It provides a high-performance multidimensional array object, and tools for working with these arrays. In this exercise you will learn several key NumPy functions that are useful in machine learning and deep learning.

NumPy Arrays
A NumPy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize NumPy arrays from nested Python lists, and access elements using square brackets:

```
import NumPy as np

a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a))           # Prints "<class 'numpy.ndarray'>"
print(a.shape)           # Prints "(3,)"
print(a[0], a[1], a[2])  # Prints "1 2 3"
a[0] = 5                 # Change an element of the array
print(a)                 # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

NumPy also provides many functions to create arrays:

```
import NumPy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
```

```
                    #          [ 0.  0.]]"
b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"
d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"
e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

NumPy offers several ways to index into arrays. Like Python lists, NumPy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import NumPy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array:

```
import NumPy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

Integer array indexing: When you index into NumPy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
import NumPy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
import NumPy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
import NumPy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a NumPy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"
```

```
# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Every NumPy array is a grid of elements of the same type. NumPy provides a large set of numeric datatypes that you can use to construct arrays. NumPy tries to guess a datatype when you create an array, but functions that construct arrays usually it also include an optional argument to explicitly specify the datatype. Here is an example:

```
import NumPy as np

x = np.array([1, 2])   # Let NumPy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let NumPy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the NumPy module:

```
import NumPy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))
```

```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Note that `*` is elementwise multiplication, not matrix multiplication. We instead use the `dot` function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. `dot` is available both as a function in the NumPy module and as an instance method of array objects:

```
import NumPy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

NumPy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

```
import NumPy as np

x = np.array([[1,2],[3,4]])
```

```
print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the `T` attribute of an array object:

```
import NumPy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

Broadcasting is a powerful mechanism that allows NumPy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array. For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
import NumPy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector `v` to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this approach like this:

```
import NumPy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

NumPy broadcasting allows us to perform this computation without actually creating multiple copies of `v`. Consider this version, using broadcasting:

```
import NumPy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if `v` actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

Here are some applications of broadcasting:

```
import NumPy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
# Multiply a matrix by a constant:
# x has shape (2, 3). NumPy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

Tasks:

6. **Reshaping arrays with NumPy:**
   Create a function that takes a NumPy array of shape (length,width,height) and converts it in to a vector of shape (length*width*height,1). Use the function array.reshape() for this.

7. **Building a basic mathematical function with NumPy:**
   a) Write a function that returns the sigmoid of a real number x. Use math.exp(x) for the exponential function. Sigmoid(x)=1/(1+exp(-x)).
   b) Now create a list of 5 values and call your sigmoid function with the list as input. You will get an error because math.exp() only works when input is a real number. It does not work with vectors and matrices. Now create a new function for sigmoid but this time use np.exp() instead of math.exp(). Np.exp() works with all types of inputs including real numbers, vectors and matrices. In deep learning we mostly use matrices and vectors. This is why NumPy is more useful. Call your new function with a vector created by np.array() function.

8. **Implementing a function on a matrix:**
   Create a function that takes a matrix as input and returns the softmax (by row) of matrix. The softmax function is defined like this:

$$softmax(\boldsymbol{x}) = softmax([x_1\ x_2\ \dots\ x_n]) = [\frac{e^{x_1}}{\sum_j e^{x_j}}\ \frac{e^{x_2}}{\sum_j e^{x_j}}\ \dots\ \frac{e^{x_n}}{\sum_j e^{x_j}}]$$

   Check if your function is working correctly by using suitable inputs.

9. **Finding dot product using NumPy:**
   a. Create a function that implements dot product of two vectors. The input to the function should be two standard python lists. Identify the time taken to evaluate the dot product using a particular example of your choice.
   b. Now create another function that implements dot product of two vectors using np.dot() function. Identify the time taken to evaluate this dot product and compare it with the time taken in part a.

10. Finding outer product using NumPy:

a) Create a function that implements outer product of two vectors. The input to the function should be two standard python lists. Identify the time taken to evaluate the outer product using a particular example of your choice.

b) Now create another function that implements outer product of two vectors using np.outer() function. Identify the time taken to evaluate this dot product and compare it with the time taken in part a.

11. Defining Loss Functions:

a) L1 loss function can be defined as follows:

$$L_1(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=0}^{m} \left| y^{(i)} - \hat{y}^{(i)} \right|$$

Create a function that takes two vectors in the form of standard python lists and returns the L1 loss according to the above formula.

b) Now create another function that returns L1 loss but uses NumPy arrays instead of standard python list. Compare the two approaches.

c) L2 loss function can be defined as follows:

$$L_2(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=0}^{m} \left(y^{(i)} - \hat{y}^{(i)} \right)^2$$

Create a function that takes two vectors in the form of standard python lists and returns the L2 loss according to the above formula.

d) Now create another function that returns L2 loss but uses NumPy arrays instead of standard python list. Compare the two approaches.

12. Perform Matrix and Matrix Addition:

a) Create a function that performs matrix and matrix addition by using standard python data structures only.

b) Create a function that performs matrix and matrix addition by using NumPy arrays.

13. Perform Matrix and Vector Multiplication:

c) Create a function that performs matrix and vector multiplication by using standard python data structures only.

d) Create a function that performs matrix and vector multiplication by using NumPy arrays.

14. Perform Matrix and Matrix Multiplication:

e) Create a function that performs matrix and matrix multiplication by using standard python data structures only.

f) Create a function that performs matrix and matrix multiplication by using NumPy arrays.

Deliverable:

Please submit your notebook on LMS before deadline

Lab Rubrics

Assessment	**Does not meet expectation** **(1/2 marks)**	**Meets expectation** **(3/4 marks)**	**Exceeds expectation** **(5 marks)**
Software Problem Realization **(CLO1, PLO1)**	The student struggles to apply the given problem and does not identify an appropriate technique to solve it. There is a lack of demonstration of the problem's data requirements and no attempt to preprocess or explore the data effectively.	The student applies the given problem with some guidance, identifies a suitable technique with hints, and shows basic visualizations. However, the approach might not be fully optimized or lacks a thorough justification.	The student independently applies the given problem, selects the most appropriate technique without guidance, and effectively visualizes and explores the data. The choice of algorithm and data handling demonstrates a deep understanding of the problem's context and data characteristics.
Software Tool Usage **(CLO4, PLO5)**	Code has syntax errors, and the implementation of the functions using NumPy library is incorrect or incomplete. The code is not modular and lacks comments for readability and reuse. The student shows no ability to use NumPy library where required.	The student has implemented the functions using NumPy library correctly with minor mistakes. The code is mostly correct in terms of syntax and functionality but might not be optimized or well-structured for reuse. Some documentation is provided. The student also shows working knowledge of relevant visualization libraries where required.	The student has implemented the functions using NumPy library efficiently and correctly. The code is clean, modular, well-documented, and follows best practices for reproducibility and reuse. The student demonstrates full command of NumPy library and tools.