# Open Lab 1

## CSCI 4850-5850 - Neural Networks

### Due: Feb. 3 @ 11:00pm

IPython notebooks provide a natural way to develop and test code/scripts for use in machine learning applications due to cell-based modularization, integrated markdown with mathjax utilities, and inline display of plots/graphs/images.

For this assignment you will utilize the JupyterLab environment provided by the BioSim cluster machines (or any other installation of Jupyter notebook and all necessary Python packages) to create an IPython notebook which takes advantage of these capabilities.

JupyterHub on BioSim is accessible using this link: https://jupyterhub.cs.mtsu.edu/biosim/

Use your CS department credentials (Username will match your PipelineMT ID: get an account at https://mgt.cs.mtsu.edu/aru/ if you don't already have one) to log into your JupyterLab session. You **must** obtain a CS department account that matches your PipelineMT ID to turn in **all** assignments for this course, so be prepared in advance.

## Introduction to $\LaTeX$ Math Mode

Often in our neural net work (yes, I will use this joke far too frequently) we will need to express the mathematical underpinnings of our work in a convenient form. Jupyter notebooks integrate the MathJax Javascript library in order to render mathematical formulas and symbols in the same way as one would in $\LaTeX$: document preparation software often used to typeset textbooks, research papers, or other technical documents. We will look more into $\LaTeX$ later in the semester, and only focus on rendering mathematical expressions in JupyterLab for now.

First, we will take a look at a couple of rendered expressions and then examine the corresponding way to render these in the notebook. After that, we will perform some follow-up exercises which will help you become more familiar with these tools and their corresponding documentation. Learning to read and utilize documentation is **very** important anytime you are learning new skills, so do not excuse yourself to work from memory alone.

Let's begin by learning how to render a mathematical expression using **markdown cells** and $\LaTeX$ formattting syntax. A common expression used in neural networks is the *weighted sum* or *net input* of a neuron rendered as so:

$y = \sum_{i=1}^{N} w_i x_i + b$

where the variable $y$ will be the result of the *weighted sum* of the elements in the vector, $\boldsymbol{x}$. This means that each element of the vector, $x_i$, is multiplied by a corresponding weight, $w_i$. An additional scalar term, $b$, known as the *bias* is added to the overall result as well. This expression is more commonly written as:

$y = \boldsymbol{w}\boldsymbol{x} + b$

where $\boldsymbol{w}$ and $\boldsymbol{x}$ are both vectors of length $N$. Note the subtle difference in the notation where ***vectors*** are in bold italic, while *scalars* are only in italic.

These kinds of expressions can be rendered in your notebook by creating **markdown cells** and populating them with the proper expressions. Normally, a cell in a Jupyter notebook is for some code that you would like for the interpreter to execute, but there is a drop-down menu at the top of the current notebook which can change the mode of the current cell to either *Code*, *Markdown*, or *Raw* (it will say *Code* by default). We will rarely use *Raw* cells, but the *Code* and *Markdown* types are both quite useful.

To render both of the two expressions above, you will need to create a markdown cell, and then enter the following code into the cell:

```
$y = \sum_{i=1}^{N}{w_i x_i + b}$
$y = \boldsymbol{w}\boldsymbol{x}+b$
```

You should notice first that each expression is surrounded by a set of $ symbols. Any text that you type between two \$ symbols is rendered using $\LaTeX$ mathematics mode. $\LaTeX$ is a complete document preparation system that we will learn more about later in the semester. For now, the important thing to understand is that it has a special mode and markup language used to render mathematical expressions, and this markup language is supported in *markdown* cells in Jupyter notebooks.

Second, you can see that special mathematical symbols such as summation, $\sum$, can be rendered using the "sum" escape sequence (\sum) where \ is the math mode escape character. There are numerous different escape sequences that can be used in math mode, each representing a common mathematical symbol or operation.

Third, you can see that symbols can be attached to other symbols for rendering as sub- or super-scripts by using the _ and ^ operators, respectively. You can also use curly-braces (liberally) to group symbols together into these sub- or super-scripts and the curly-braces, themselves, will not be rendered in the equation. These curly-brace delimeters only help the math mode interpreter understand which symbols you would like grouped together, and won't be displayed unless escaped.

Finally, it is clear that many symbols are rendered in a way that makes intuitive sense. For example, the bias term, $b$, is simply provided with no markup. Any text **not** escaped or otherwise marked up will be rendered as a standard scalar (plain italic). However, the `\mathrm{}` sequence can be used to render standard (roman) text when required. For example:

`$a\ \mathrm{plus}\ b$`

$a\ \mathrm{plus}\ b$

Notice also how a backslash followed by a space will add a space between the words or symbols. Normally, when two scalars are presented, it is assumed they are being multiplied together, and are placed closely together to represent this fact. Therefore, since spaces normally would *not* be desired between symbols/variables in mathematical expressions, we must include them explicitly ourselves when using math mode.

When you have a new mathematical symbol that you need to use, going to the documentation will help you find it. There is link in the **Reference Material** section at the end of this lab to a comprehensive Here are a few other example expressions and their corresponding $\LaTeX$ code:

- The following is an example of a matrix decomposition (while bold lower-case variables indicate vectors, bold upper-case variables indicate matrices):

`$\boldsymbol{A}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$`

$\boldsymbol{A}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$

- Greek letters are also commonly used in mathematical expressions. For neural networks, many *hyperparameters* (values that are tuned by the programmer to improve network performance) are often rendered as such. All Greek letters can be rendered in either upper- or lower-case form:

`$\alpha \beta \Theta \Omega$`

$\alpha \beta \Theta \Omega$

- Many mathematical operators make special use of the sub- and super-script syntax:

`$\int_{-\pi}^{\pi} \sin{x}\ dx$`

$\int_{-\pi}^{\pi} \sin{x}\ dx$

`$\prod_{i=1}^{N}{(x_i+y_i)^2}$`

$\prod_{i=1}^{N}{(x_i+y_i)^2}$

- Rendering fractions requires an operator which accepts two arguments, each in a set of curly braces, to define the upper and lower parts of the expression, respectively:

`$f(x)=\frac{1}{x^2}$`

$f(x)=\frac{1}{x^2}$

$\frac{d}{dx}f(x) = -\frac{2}{x^3}$

`$\frac{d}{dx}f(x) = -\frac{2}{x^3}$`

## Introduction to Matplotlib

Matplotlib is a Python library that is very useful for visualizing data and results in graphical form. You will be using Matplotlib (and several other related libraries) to make plots and figures for your lab assignments and final project paper in this course, so it will serve you well to become familiar with its general capabilities for now.

Let's start with an example to illustrate the core concepts...

In [1]:
```
# Typical import for numpy
# We will use a utility function or two for now...
import numpy as np
```

In [2]:
```
#Typical import for Matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

Importing packages and functions in python work as expected. However, in the notebook you will sometimes need to run special commands to gain additional functionality. The `%matplotlib inline` command tells Jupyter to render your plots directly in your notebook as part of the output for a code cell. This will be the most common way to use plots, but we will explore other methods as well.

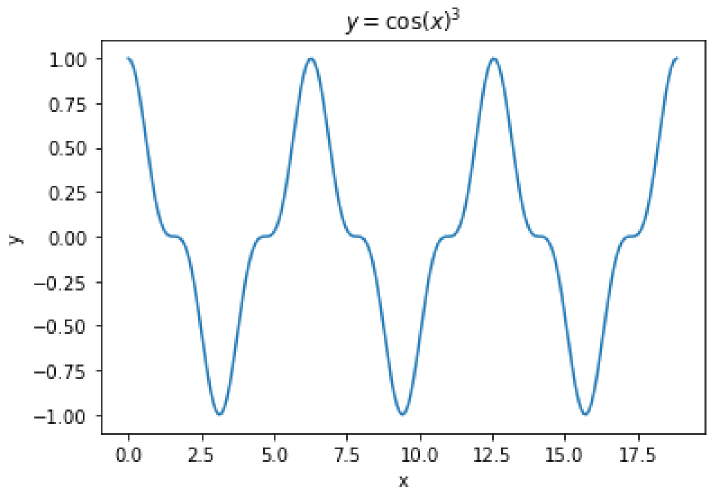
In [3]:
```
x = np.linspace(0,6*np.pi,201)
y = np.power(np.cos(x),3.0)
print('The length of x is %d'%(len(x)))
print('The length of y is %d'%(len(y)))
```

```
The length of x is 201
The length of y is 201
```

Generating a sequence of floating points numbers and storing them as a numpy array is the main use of the `numpy.linspace()` function. The sequence above goes from zero to $6\pi$, taking exactly 201 evenly-sized steps to get there. This is handy for generating a set of values that we would like to use to evalute (or sample) a mathematical function. We set these values to be the 'x' values we would like to evaluate our function at. Rather than passing a single value to a mathematical function, we can send an entire array and the function will be computed for all elements in the array. Here we use the `numpy.power()` and `numpy.cos()` functions to calculate function $y = \cos(x)^3$.

At this point, y is a numpy array that contains the value of the function at each of the 201 values in x, and hence are the same size. Because of this, we can pair them up and use them to make a plot using the matplotlib library.

In [4]:
```
plt.plot(x,y) # x and y must be vectors/lists of the -same length-
plt.title('$y=\cos(x)^{3}$')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



The `plot()` function will take a vector x and y coordinates and render a line plot, connecting points specified by array elements at the same index. You can add a label to the x and y axes using the `xlabel()` and `ylabel()` functions, and a title similarly with the `title()` function. Note that you can surround math elements with a \$ and get the same $\LaTeX$ rendering as you would in markdown cells in the notebook. Finally, the `show()` function tells the notebook to render the final product in the output cell.

## Introduction to Sympy

What if you would like to perform some calculus alongside your other computations? For that, you would need a computer algebra system (CAS). Luckily, the sympy package can provide you with the tools to perform symbolic computations and then can help you numerically evaluate those results.

Let's get started...

In [5]:
```
# Import sympy tools
import sympy as sp
sp.init_printing(use_latex=True)
```

Note that I imported sympy, but then also initialized the sympy printing facilities to utilize $\LaTeX$. The reason for this will be obvious soon, if you haven't already figured it out.

Rather than have x be a vector, I need it to be a symbol. In particular, I need it to be a symbol that one would like to manipulate similar to how one uses symbols in mathematical expressions. Unlike variables which point to specific data structures, a symbol is more like how a variable is used in mathematics, and can take on values at a later time.

In [6]:
```
# Make x a symbol
x = sp.symbols('x')
```

In [7]:
```
# Let's write an expression
y = sp.cos(x)
```

In [8]:
```
# Just provide the expression by itself,
# and it will be printed with LaTeX!
y
```

Out[8]: $\cos(x)$

So, you can define mathematical expressions in sympy, and they will be rendered using $\LaTeX$. Don't be confused by the naming conventions though. In this particular case, x is a standard python variable: it just so happens to reference the sympy *symbol* 'x'. We are naming the python variable the same as the intended mathematical symbol just for parsimony. Similarly, y is a standard python variable, but it now references a sympy expression (which itself is composed of a sympy function defined in terms of the symbol x):

In [9]:
```
type(x)
```

Out[9]:
```
sympy.core.symbol.Symbol
```

In [10]:
```
type(y)
```

Out[10]:
```
cos
```

In [11]:
```
type(type(y))
```

Out[11]:
```
sympy.core.function.FunctionClass
```

In [12]:
```
y.free_symbols
```

Out[12]: $\{x\}$

Additionally, we can perform symbolic math on that expression. For example, let's take the derivative with respect to x using the `diff()` function.

In [13]:
```
dydx = y.diff(x)
```

In [14]:
```
dydx
```

Out[14]: $-\sin(x)$

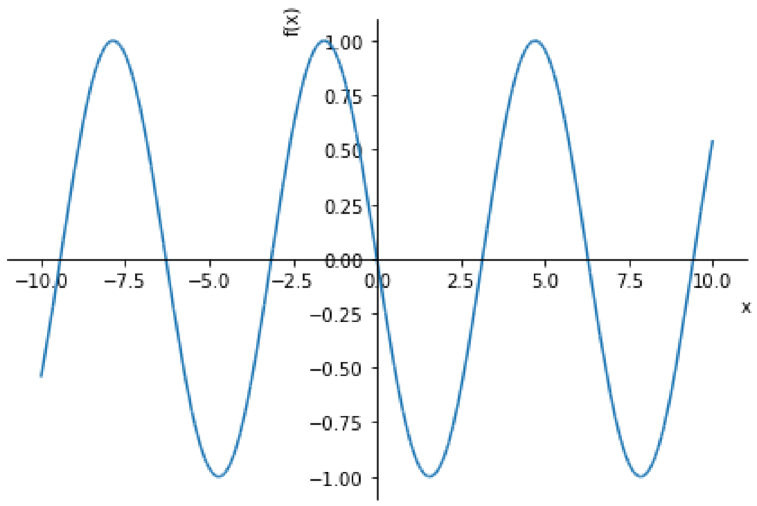
Now we have the derivative of the function with respect to x, and have solved it **analytically** (rather than numerically) using sympy! The `diff()` function is a member function of the function class, but you can also use the general `diff()` function if desired:

In [15]:
```
sp.diff(y,x)
```

Out[15]: $-\sin(x)$

Sympy has it's own builtin function which utilize matplotlib underneath for plotting expressions as well...

In [16]:
```
sp.plot(dydx)
```



```
Out[16]: <sympy.plotting.plot.Plot at 0x7f115ccd2b50>
```

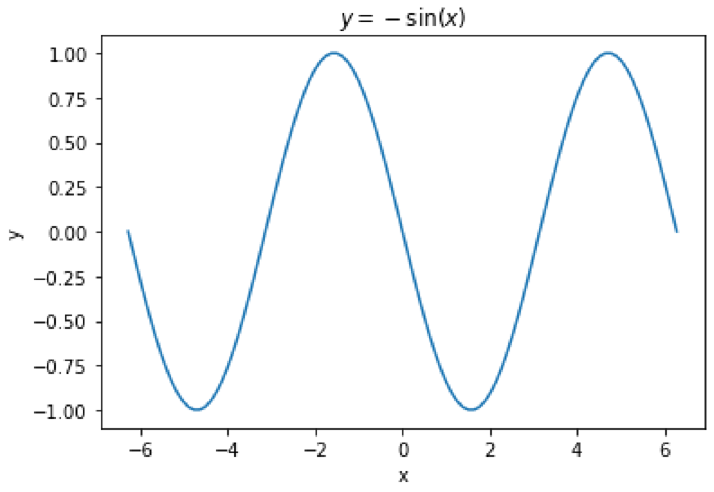
However, we may want more control over the x values sampled and plot itself. This can sometimes be better done by evaluating the function *numerically* at the intended points. Let's do that now...

```
In [17]: x_vals = np.linspace(-2*np.pi,2*np.pi,101)
         y_vals = np.array([dydx.evalf(subs=dict(x=x_val)) for x_val in x_vals])
         print('The length of x is %d'%(len(x_vals)))
         print('The length of y is %d'%(len(y_vals)))
```

```
The length of x is 101
The length of y is 101
```

Here we have used a python list comprehension to evaluate our derivative (dydx) at each of the 101 points in the $-2\pi$ to $2\pi$ range created by `linspace()`. The `evalf()` function allows us plug in specific numberic values for our symbols. In particular, we pass the subs= argument a python dictionary object which contains a mapping from a symbol to a particular value we would like to associate with that symbol. Multiple symbols can be passed into the function using the dictionary object, so that functions with more than one symbol can be evaluated numerically as well. We can now plot those specific values and maintain full control over the appearance of the plot itself since we are using matplotlib/pyplot directly:

```
In [18]: plt.plot(x_vals,y_vals)
         plt.title('$y=%s$'%(sp.latex(dydx)))
         plt.xlabel('x')
         plt.ylabel('y')
         plt.show()
```



You can also now see in the code above how the `sympy.latex()` function can be used to convert the expression we were storing in dydx into a string form recognized by $\LaTeX$ math mode, and therefore also the `pyplot.title()` function from matplotlib. It's usually much easier to control how you want your plots to look using numerical evaluation instead of using sympy's built-in plotting tools, so keep that in mind in the future.

## Reference Materials

Now that you have had the chance to examine $\LaTeX$ math mode, matplotlib, and sympy a little (as well as a little numpy, to be fair), here are some links to the documentation materials for those tools and a few others.

1. Python: https://www.cs.mtsu.edu/~jphillips/courses/CSCI4850-5850/public/python-tutorial.ipynb
2. JupyterLab: https://jupyterlab.readthedocs.io/en/stable/
3. $\LaTeX$ math mode: https://www.overleaf.com/learn/latex/Mathematical_expressions
4. Markdown: https://www.markdowntutorial.com/
5. Sympy: https://docs.sympy.org/latest/index.html
6. Numpy: https://numpy.org/doc/1.18/reference/index.html

**You will need to use the math mode reference and numpy reference** to find some of the symbols and functions used below (for example, sigma, $\sigma$, and numpy.exp(), $e$).

# Assignment

Now that you have some examples of how to use these tools in-hand, you will use your skills to create a notebook of your own that addresses the following three requirements:

1. Render the following 3 formulas (as shown) in a markdown cell using $\LaTeX$ math mode:
$\zeta(x) = \log(1 + e^x), \quad \sigma(x) = \frac{1}{1+e^{-x}}, \quad y_i = \frac{e^{y_i}}{\sum_{j=1}^{N} e^{y_j}}$
2. Generate a plot of the following function of x, evaluated in the range [-5,5]: $y = \frac{1}{1+e^{-x}}$
3. Use sympy to calculate the derivative of the equation from problem 2 with respect to x, and then also plot that function in the range [-5,5].

Once you have performed these three steps, clearly label your answers (Problems 1, Problem 2, and Problem 3) using additional markdown cells, save it, and then right-click on your notebook file from the "Files" browser on the left side of JupyterLab and select "Download" to download your notebook onto your local computer. Create a ZIP file containing only your IPython notebook file and submit it to the course assignment submission system by the due date at the top of this assignment. (Note that you can also ZIP the file from the command line in JLab, and then just download/submit the ZIP file to the submission system.)

CSCI 4850-5850 Assignment Submission System: https://4850.cs.mtsu.edu/